Изучение ACPI на примере разработки драйвера кнопки Power для ОС FreeBSD

Автор: Залата Руслан Николаевич

Дата: 22.10.2026

Ссылка: https://www.fabmicro.ru/pub/articles/FreeBSD_Power_Button_Driver.pdf

SPDX-License-Identifier: CC-BY-SA-4.0

Так получилось, что я уговорил одного своего коллегу установить на его рабочий ноутбук «HP Laptop 15s» операционную систему FreeBSD 15.0. Он уже был немного знаком с ОС FreeBSD в серверном варианте, но с этой ОС на десктопе ранее никогда не сталкивался, поэтому мы решили устроить на работе небольшой «сэйшен по установке Фрюхи» и выполнить установку под моим пристальным контролем. В целом, не учитывая экспериментов с разбивкой диска, совмещением и сожительством с ОС Windows, установки менеджера загрузки **rEFInd** и борьбы с глюками USB накопителей, установка ОС FreeBSD на отдельный раздел диска прошла удачно. Беспроводная WiFi карта и встроенная AMD графика заработали «из коробки». В систему был установлен X11 сервер Xorg, а к нему графическая оболочка Xfce4 и много прочего разного ПО для удобства выполнения повседневных задач.

Преисполненные чувством выполненного большого дела, мы решили разбегаться по домам, коллега нажал на ноутбуке кнопку выключения питания и обнаружил, что система никак не реагирует на это нажатие, хотя команда *shutdown* выполняется корректно - с выключением питания. Я тут же вспомнил, что в ОС FreeBSD поведение системы при нажатии на кнопку «Power» задается системной переменной **hw.acpi.power_button_state**, но установка её значения в **S5** («shutdown and power off») ни к чему не привела. Тогда я включил режим «многословности» для операционной системы и стал наблюдать сообщения от ядра ОС на консоли, одновременно многократно нажимая кнопку «Power». На удивление, ни одного сообщения от ACPI я не обнаружил, и это был очень нехороший знак — кнопка либо физически сломана, либо что-то конкретно «сломано» в ACPI в ядре ОС FreeBSD. Мы загрузились в Windows, убедились в том, что там с кнопкой «Power» всё в порядке, а значит остается одно — дебажить ядро Фрюхи и выяснять причину. Ведь не могу же я оставить коллегу «один на один» с нерабочей системой на его ноутбуке.

Далее я поведаю историю со счастливым концом о том, как в течении двух недель я, тайком от семьи, по ночам, разбирался с устройством «Advanced Configuration and Power Interface» (ACPI), дебажил ядро операционной системы FreeBSD и писал свой небольшой драйвер для кнопки «Power».

ACPI
FreeBSD

СОДЕРЖАНИЕ

1. Погружение в ACPI..... 5
1.1. Что такое ACPI ?..... 5
1.2. Аппаратные регистры ACPI..... 12
1.3. Поддержка произвольных источников событий (GPE)..... 16
1.4. Поддержка событий через GPIO..... 18
1.5. ACPI Namespace..... 19
1.5.1. Корневые пространства имен..... 20
1.5.2. Служебные объекты и их методы..... 21
1.5.3. Hardware ID (Plug-and-Play ID)..... 24
1.5.4. Уникальный идентификатор устройства (UUID)..... 25
1.6. Таблицы с «Firmware»..... 26
1.6.1. Таблица «Fixed ACPI Description Table» (FADT)..... 29
1.6.2. Таблица «Firmware ACPI Control Structure» (FACS)..... 46
1.6.3. Таблица «Differentiated System Description Table» (DSDT)..... 47
1.7. Язык «ACPI Machine Language» и его Си-подобное представление (ASL)..... 48
1.7.1. Типы данных в AML/ASL..... 48
1.7.2. Конструкции языка AML/ASL (операторы)..... 49
1.7.3. Оператор Notify() для отправки нотификаций в OSPM..... 52
1.7.4. Компиляция и декомпиляция ASL/AML..... 53
2. Анализируем таблицы ACPI «Firmware» от ноутбука «HP Laptop 15s»..... 57
2.1. Дамп таблиц и их декомпиляция..... 58
2.2. Дамп таблиц и их декомпиляция (BSD-версия)..... 63
2.3. Компиляция ASL кода и подстановка DSDT таблицы..... 63
2.4. Загрузка ядра ОС FreeBSD в режиме «многословности» (Verbose Mode)..... 66
2.5. Если что-то пошло не так..... 69
2.6. Вызов метода ACPI из командной строки..... 69
2.7. Информация о поддерживаемых ОС и её свойствах..... 70
2.7.1. Метод _OSI для запроса «Operating System Interfaces»..... 70
2.7.2. Метод _OS для запроса типа операционной системы..... 72
3. Поиск причины «неисправности» кнопки Power..... 75
3.1. Включение режима отладки для ACPICA..... 76
3.2. Установка исходных кодов ОС FreeBSD 15.1-RELEASE..... 77
3.3. Сборка отладочного ядра ОС FreeBSD..... 78
3.4. Получение отладочной информации от ACPICA..... 81
3.5. Анализ кода ПО ACPI и поиск причины отсутствия события PWRB..... 84
3.5.1. Обработка событий от Embedded Controller (метод EC0._Qxx)..... 85
3.5.2. Обработка событий от GPIO (метод GPIO._EVT)..... 86
3.6. Драйвер GPIO контроллера для процессоров AMD (amdgpio)..... 87
3.7. Взаимодействие с GPIO в ОС FreeBSD (GPIO Bus)..... 89
3.8. Вызов метода _EVT при смене состояния линий GPIO (модуль gpioaei)..... 91
3.9. Анализ кода обработки прерываний от GPIO..... 93
3.10. Подсистема обработки аппаратных прерываний в ОС FreeBSD..... 96
3.10.1. API intr_event..... 96
3.10.2. API bus_setup_intr..... 98
3.10.3. Аппаратные прерывания как виртуальный ресурс (global IRQ)..... 98
4. Разработка модуля «gpiopwrb» для протезирования прерываний от кнопки «Power»..... 103
4.1. Структура драйвера (модуля) в ОС FreeBSD..... 103

4.1.1. Контекст (softc)....104
4.1.2. Методы устройства (device)....104
4.1.3. Методы экспортируемого интерфейса (bus interface)....105
4.1.4. Регистрация модуля (драйвера) в Nexus....107
4.1.5. Сборочный скрипт (Makefile)....107
4.2. Структура разрабатываемого модуля gpiopwrb....108
4.3. Контекст модуля gpiopwrb (struct gpiopwrb_softc)....110
4.4. Инициализация и деинициализация модуля gpiopwrb....112
4.4.1. Реализация метода device_identify....112
4.4.2. Реализация метода device_probe....114
4.4.3. Реализация метода device_attach....115
4.4.4. Реализация метода device_detach....117
4.5. Реализация потока gpiopwrb_daemon....120
4.6. Сборка и тестирование модуля gpiopwrb....122
4.7. Подготовка man-странички руководства для gpiopwrb....126
4.7.1. Подсистема встроенной документации mdoc....127
4.7.2. Структура документа mdoc....128
4.7.3. Добавление документации в проект....132
4.8. Установка драйвера в систему....133
5. Заключение....134
Список используемых источников....135

1. Погружение в ACPI

Впереди предстояли долгие майски праздники и я попросил коллегу предоставить мне его ноутбук с «нерабочей» кнопкой «Power» для исследования — была небольшая надежда на то, что я смогу достаточно быстро исправить эту проблему стандартными средствами, например, подкруткой переменных в *sysctl.conf* или загрузкой дополнительного модуля ядра про который я просто не знаю. Погода, кстати, выдалась располагающая к «режиму глубокого хака» — почти все выходные и праздничные дни мая лил дождь и дул сильный ветер, слезать с теплого дивана было противопоказано.

У меня уже было некоторое представление о том, что всё что, связано с «электроснабжением» на x86 платформе так или иначе завязано или управляется с помощью подсистемы ACPI. Поэтому на следующий день я открыл свой ноутбук с ОС FreeBSD 13.5 и принялся исследовать вопрос.

1.1. Что такое ACPI ?

Так как мой уровень представления о технологии «Advanced Configuration and Power Interface» был близок к абсолютному нулю, то первым делом я решил прочитать обзорную статью на Wikipedia. Из неё я усвоил следующие основные моменты:

1. ACPI придумали в конце 90-х годов прошлого века как способ замены устаревшей на тот момент подсистемы «Advanced Power Managment» (APM) унаследованной от архитектуры IBM PC. К разработке и внедрению ACPI приложили усилия гиганты индустрии: Intel, Toshiba и, конечно же, Microsoft. На данный момент ACPI является промышленным стандартом, имеется масса версий и разновидностей спецификации. Текущая версия спецификации имеет номер 6.6. Группа разработчиков ACPI входит в состав и контролируется организацией «UEFI Forum».
2. Как следует из названия, ACPI заведует всем тем, что связано с источниками питания (батареи, внешние AC/DC преобразователи), с режимами энергосбережения, дополнительными кнопками («Power», «Sleep»), «kill switch» для RF-модулей, с подсветкой экрана монитора и даже с крышкой корпуса ноутбука («Lid»), а также занимается контролем за тепловыделением и охлаждением («Термальные Зоны», вентиляторы, датчики температуры).
3. ACPI также заведует конфигурацией аппаратуры и её диагностикой — с помощью ACPI операционная система: а) получается список устройств имеющихся в данной машине, б) производит их первичную инициализацию и тестирование, в) управляет электропитанием и режимами энергосбережения каждого устройства в отдельности.
4. Подсистема ACPI разделяется на относительно небольшую аппаратную и объемную, и по большей части проприетарную, программную части. При этом, программная часть исполняется в ядре основной операционной системы и обязательно содержит в себе виртуальную машину, которая выполняет проприетарный байт-код выраженный на «ACPI Machine Language» (**AML**), поставляемый производителем «железа» (OEM).
5. Этот байт-код принято называть «Firmware». На стадии загрузки операционной системы, еще до запуска загрузчика ОС, BIOS размещает копию AML кода в оперативной памяти машины в виде специально организованных таблиц под общим названием **DSDT —** «Differentiated System Description Table». Операционная система обнаруживает эти таблицы, парсит их, и запускает исполнение виртуальной машины.
6. Таблиц много, они могут содержать как байт-код (AML), так и различные структуры данных необходимые программе «firmware» для работы. Также есть таблицы чисто

описательные, например, важная таблица **FADT** содержит различные параметры и особенности реализации подсистемы ACPI на конкретной машине.

7. «Firmware» выполняется с самого начала загрузки ОС и продолжает до момента выключения питания, постоянно общаясь с основной операционной системой через специализированный API. Структура кода AML построена по событийной модели — события (аппаратные прерывания от ACPI аппаратуры, от ядра ОС, драйверов или даже от приложений пользователя) поступают в виртуальную машину, обрабатываются там AML-кодом, который формирует поток других событий внутри себя, и/или поднимает поток событий в основную ОС.
8. «Firmware» имеет доступ абсолютно ко всей аппаратуре машины на которой он исполняется и может читать/изменять любой регистр любого контроллера или устройства, при этом ОС обязана обеспечить примитивы для работы с аппаратурой (методы ввода/вывода и доступа к аппаратным портам).
9. «Firmware» разрабатывают в процессе разработки BIOS на специализированном (domain-specific) объектно-ориентированном языке программирования, который называется «ACPI Source Language» (**ASL**). Исходный текст программ написанных на ASL компилируют в AML байт-код специализированным компилятором **iASL** поставляемым компанией Intel в месте с эталонной реализацией программной части подсистемы ACPI, которая называется «ACPI Component Architecture» (**ACPICA**).
10. iASL и ACPICA поставляются с открытым исходным кодом с разрешительной лицензией. Код ACPICA адаптирован всеми современными операционными системами, в том числе Linux и *BSD.
11. У Microsoft в ОС Windows имеются свои проприетарные реализации iASL и ACPICA, они местами могут отходить от спецификации стандарта (и от того, что распространяет Intel). Это значит, что AML байт-код «firmware» в открытых ОС, использующих ACPICA от Intel, и в закрытой ОС Windows исполняются по-разному. В этом есть весьма существенный нюанс часто приводящий к тому, что какая-то часть аппаратуры имеет лучшую поддержку в Windows, так как OEM производитель часто не беспокоится о совместимости с иными ОС кроме Windows!

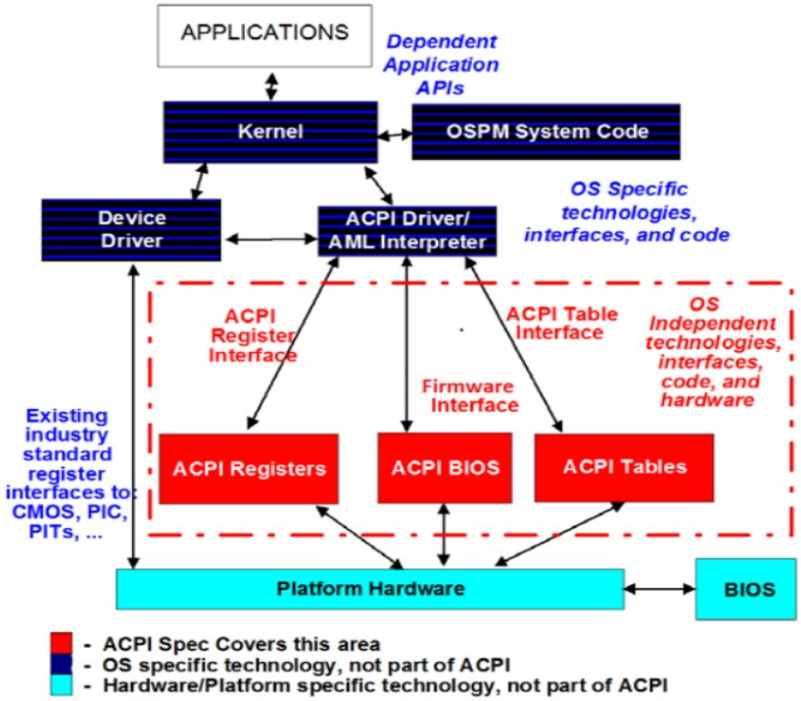

Рис 1. Структура подсистемы ACPI в составе вычислительной системы.

На рис.1 представлена структура программной части ACPI из которой видно, что ACPI создает прослойку между операционной системой и оборудованием, она затягивает на себя всё общение ОС с «железом», местами замещая драйверы проприетарным AML кодом.

Очевидно, что такая сложная подсистема не может быть реализована без изъянов. В ACPI постоянно обнаруживаются проблемы совместимости и даже критические к безопасности уязвимости. Ниже приведена пара цитат с сайта Wikipedia от известных в индустрии людей:

> *In November 2003, Linus Torvalds—author of the Linux kernel—described ACPI as "a complete design disaster in every way".[56][57]*
>
> *Ubuntu founder Mark Shuttleworth says ACPI is a security risk.[55] He says "arguing for ACPI on your next-generation device is arguing for a trojan horse of monumental proportions to be installed in your living room and in your data centre" and "firmware on your device is the NSA's best friend". He goes on to say, "Your biggest mistake is to assume that the NSA is the only institution abusing this position of trust – in fact, it's reasonable to assume that all firmware is a cesspool of insecurity, courtesy of incompetence of the highest degree from manufacturers, and competence of the highest degree from a very wide range of such agencies." He suggests open-source, declarative firmware (ACPI or non-ACPI) as a solution.[56]*

Стоит заметить, что ACPI не единственный метод получения конфигурации и начальной инициализации аппаратуры для операционных систем. В «embedded» среде (в основном на ARM и RISC-V архитектурах) широко используется метод под названием «Flattened Device-Tree» (FDT). Вкратце суть метода FDT такова:

1. Всей аппаратурой заведует операционная система, для чего в ней для каждого аппаратного блока должен быть установлен свой драйвер (модуль).
2. При загрузке ОС, ей передается, в бинарном виде конфигурационное дерево (**.DTB** файл) - блок данных содержащий описание аппаратуры, её ресурсов (регистры, области памяти) и специфические настройки. Также для каждого блока аппаратуры указывается имя обслуживающего драйвера (**compatible**).
3. Операционная система, при инициализации, перебирает узлы в этом конфигурационном дереве и вызывает метод **probe()** у соответствующего драйвера (находя его по полю **compatible**), передавая ему на вход «ветку дерева» относящуюся к этому устройству.
4. Драйвер самостоятельно производит проверку (диагностику) соответствующего блока аппаратуры, производит его инициализацию и настройку.
5. Драйвер обязан сохранять состояние обслуживаемого блока аппаратуры при изменении режимов работы операционной системы и при переходе в режимы энергосбережения. В том числе, драйвер ответственен за «усыпление» (**suspend**) и возобновление работы (**resume**) устройства.
6. Драйвер может отказать в обслуживании данного блока аппаратуры. В этом случае операционная система никак не работает с данной аппаратурой, не инициализирует, не включает/выключается её и не переводит в режим энергосбережения. Нет драйвера — нет аппаратуры.

Таким образом, ACPI является строго противоположным к «Device-Tree» подходом к работе с аппаратурой, где все перечисленные выше функции возложены на проприетарный «firmware» и AML код.

Известный в среде Open Source разработчик ПО Matthew Garrett в октябре 2023 года опубликовал статью озаглавленную «Why ACPI?». В этой статье он попытался показать какие фундаментальные проблемы решает внедрение ACPI и почему ACPI лучше чем старый APM и новый Device-Tree метод. По утверждению автора на написание этой статьи его воодушевила «нитка» на одном из форумов озаглавленная «Why does ACPI exist», в которой насчитывалось **12 239 страниц** комментариев состоящих из горячих дебатов, после чего модератор форума отключил возможность оставлять комментарии в этой нитке. Статью Matt-а рекомендую к прочтению, а вот «горячие дебаты» мне найти не удалось (видимо их уже удалили по требованию Microsoft ;)).

Также я узнал, что ACPI является неизбежным злом не только на всей x86-совместимой платформе (в т.ч. и Mac), но и то, что с 2011 года ACPI активно внедряется на платформах с ARM архитектурой. Предпосылкой к этому является попытка компании Microsoft заполонить своей ОС Windows мобильные платформы на базе ARM. С мая 2025 года в спецификацию ACPI была добавлена поддержка открытой архитектуры RISC-V, что вызывает определенные опасения.

Но вернемся к техническим аспектам.

Чтобы понять как с технической точки зрения устроена и работает ACPI, мне пришлось пробежаться по спецификации ACPI версии 6.6 и выборочно прочитать пару разделов (всю спецификацию прочитать просто не реально). Ниже я попытаюсь кратко пересказать своё видение глав «ACPI Hardware» и «ACPI Software» спецификации. Но прежде всего определимся с терминологией используемой в спецификации ACPI:

- «Operating System-directed Power Management» (OSPM) — часть основной операционной системы содержащая внутри себя «ACPI Component Architecture» (ту самую виртуальную машину) и соответствующие программные интерфейсы для функционирования ACPI.
- «Embedded Controller» (EC) — отдельный микроконтроллер в составе вычислительной системы, реализующий различные вспомогательные функции, в том числе функции ACPI. EC контроллер имеет отдельный интерфейс (ECI) для связи с центральным процессором представляющий собой фиксированный набор регистров.
- «System Management Bus» (SMBus) — шина обмена (обычно I^2C) к которой подключены все устройства задействованные ACPI: датчики температуры, датчики оборотов вентилятора, датчики напряжения батареи, датчики потребляемого тока и т. д. Такие устройства в спецификации называют «Devices».
- «System Management Interrupt» (SMI) — линия прерываний предназначенная для подачи сигнала от устройств на SMBus в операционную систему. В старых версиях APCI предусматривался механизм прямого взаимодействия драйвера внутри операционной системы с источником прерывания (с датчиком). В новых редакциях ACPI данная линия прерываний используется как источник события SCI от EC контроллера. OSPM обязана подвесить на эту линию обработчик прерываний извлекающий из EC контроллера номера событий и распространяющий их внутри OSPM.

- «System Control Interrupt» (SCI) — линия прерывания (вместо SMI для новых редакций) для информирования OSPM о произошедших в ACPI событиях. На x86 платформе такой линией прерывания обычно является **IRQ9**.
- «General-Purpose Event» (GPE) — событие от ACPI подсистемы, формируется в обработчике прерываний SCI путем считывания набора GPE-регистров из аппаратуры ACPI.
- «Fixed Feature» (FF) — некая фича для реализации которой OSPM может взаимодействовать с аппаратурой напрямую, а не через AML код. Часто такими фичами являются кнопки «Power» и «Sleep». Для таких фич ACPI предоставляет фиксированный набор регистров с битовыми масками определенными прямо в спецификации ACPI. По сути, это линии GPIO состояние которых операционная система может считывать напрямую из «Fixed Feature Registers».
- «Control Method» (CM) — способ формирования и обработки событий от аппаратуры основанный на вызове методов в коде AML.

Уже из терминологии понятно, что существует старый (через SMI прерывание) и новый (CM через SCI и GPE) методы получения событий от аппаратуры. Так же есть два вида ACPI устройств — с прямым доступом через регистры («Fixed Feature») и все остальные, доступ к которым осуществляется только через AML код. Но это еще не весь набор сложностей. Как я уже отмечал выше, Microsoft старательно пытается насадить свою ОС Windows на мобильные и прочие «embedded» платформы, где часто нет никаких дополнительных контроллеров, а всё управление делается через линии GPIO, в крайнем случае имеется микросхема PMIC с интерфейсом I^2C. Вот для таких платформ со спецификации ACPI 5.0 вводится понятие «Hardware-Reduced ACPI». OSPM обязана распознавать с какой именно реализацией ACPI она имеет дело в данной вычислительной системе и применять разные способы управления. Для определения реализованного стандарта, а также набора поддерживаемых фич («Reduced» или «не-Reduced», «Fixed» или «не-Fixed»), используется таблица «Fixed ACPI Descriptor Table» (**FADT**), в ней содержится перечень переменных с битовыми полями отвечающими за наличие/отсутствие различных функций.

Замечу, что таблица FADT в поле сигнатуры содержит строку **"FACP"**, для всех остальных таблиц сигнатура совпадает с названием таблицы. Это, разумеется, сделано для того, чтобы окончательно запутать разработчиков. ;)

В рамках спецификации ACPI определяются несколько различных уровней энергосбережения (состояний) для различных компонентов вычислительной системы:

- C0, C1, C2, Cn — состояния процессорных блоков (CPU), где C0 — максимальная производительность, а Cn — пониженный уровень производительность по сравнению с предыдущим. Обычно понижение уровня «Cx» выражается в занижении тактовой частоты процессора, в плоть до полного останова.
- D0, D1, D2, D3 — состояния уровня энергосбережения для аппаратных блоков или устройств (Device). Здесь D0 подразумевает, что устройство полностью включено и работает на максимуме потребляемого тока, а D3 — устройство полностью обесточено.
- S0, S1, S2, S3, S4, S5 — состояния «sleeping states» основной операционной системы. В состоянии S0 система работает на полную мощность, никаких энергосберегающих

функций не включено. В состоянии S5 система полностью остановлена, машина обесточена. Переход системы от меньшего Sx к большему осуществляется программно. Переход от большего Sx к меньшему — только с помощью аппаратной поддержки: нажатие кнопки, получение аппаратного сигнала WAKE, по аппаратному прерыванию от таймера, и т. д.

В спецификации ACPI сделано важное исключение для PCI/PCIe устройств — для управления их электропитанием ACPI не применяется. Управление режимами энергосбережения для таких устройств осуществляется через заложенные в стандарт PCI/PCIe механизмы, как правило, реализуемые драйвером шины. Это еще одно исключение которое нарушает стройный ряд идей ACPI как универсального средства «от всего».

На рис. 2 ниже, взятом из спецификации, представлена схема событийной модели ACPI. На этом рисунке слева обозначены источники событий, а с права — то, каким способом события доставляются в OSPM. Таких способов два: через прерывание SMI или SCI.

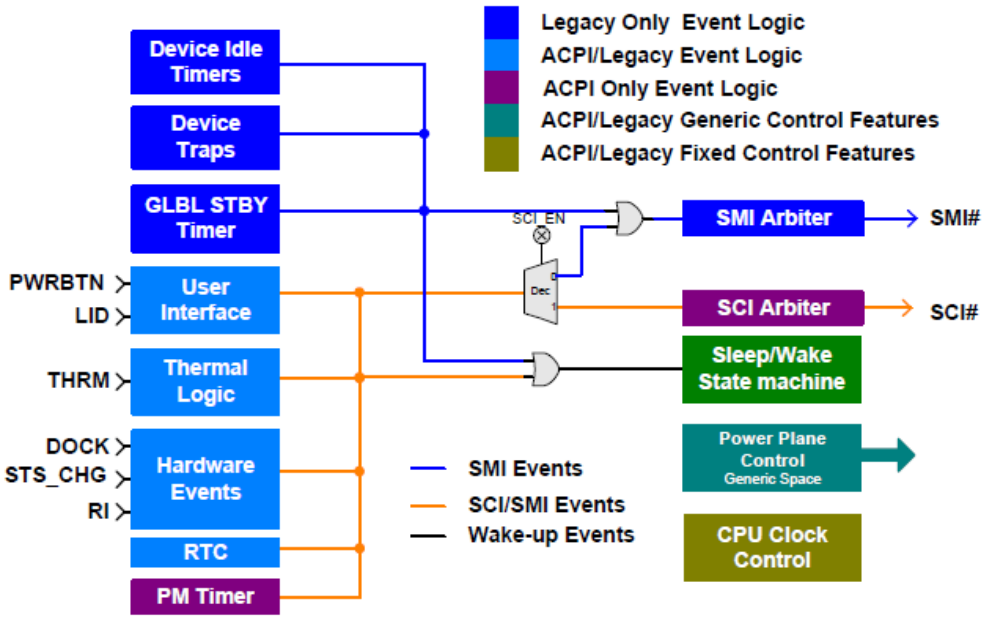

Рис.2. Схема событийной модели ACPI.

Среди источников событий (слева) имеется событие обозначенное как **PWRBTN** — это как раз интересующая нас часть всей это сложной подсистемы, т. е. кнопка включения/выключения питания. Из схемы видно, что события от PWRBTN могут транслироваться либо через линию прерывания SMI, либо через SCI, в зависимости от состояния конфигурационного бита **SCI_EN**. Этот бит мы далее обнаружим, когда будем рассматривать структуру таблицы FADT. Сейчас просто запомним этот нюанс.

Согласно логике приведенной в схеме и описанной в спецификации, при установленном бите SCI_EN, изменение состояние кнопки PWRBTN должно формировать следующую цепочку действий:

1. Аппаратный блок ACPI к которому подключена кнопка PWRBTN (см. рис 3а ниже) записывает в GPE регистры номер события ассоциированного с данным источником и формирует аппаратное прерывание (IRQ) по линии SCI.

2. Прерывание SCI обрабатывается процедурой-обработчиком в составе OSPM. Обработчик считывает из GPE регистров номер события и вызывает AML метод передавая ему номер события в качестве параметра. То, какой метод и у какой сущности будет вызван зависит от настроек и AML кода содержащегося в таблицах.

3. Внутри AML кода, программа реализующая вызванный метод производит какие-то только ей известные внутренние вычисления (она может, например, сделать запрос в другие аппаратные регистры, дождаться другого события) и в конченом счете формирует нотификацию (**Notify**) в OSPM с именем сущности **_SB.PWRB**.

4. OSPM получив от AML нотификацию транслирует её в соответствующий обработчик, в данном случае в обработчик события для сущности **_SB.PWRB** которым является системная процедура обработки нажатия кнопки «Power».

5. Операционная система, внутри процедуры обработки кнопки «Power», решает что делать: изменить режим работы (выполнить переход S0 → Sx), завершить работу и отключить питание машины (S5) или просто игнорировать это событие.

Если же конфигурационный бит SCI_EN не установлен, то изменение состояние кнопки PWRBTN будет обработано иначе:

1. Изменение состояния PWRBTN будет отмечено флагом **PWRBTN_STS** в регистре аппаратном регистре **PM1_STATUS** у EC контроллера, после чего контроллер сформирует аппаратное прерывание SMI.

2. Обработчик прерывания SMI внутри OSPM прочитает регистр PM1_STATUS и проверит все биты статуса, каждый бит ассоциирован с определенным в спецификации фиксированным событием («Fixed Features»). Обработчик SMI проверит, имеется ли зарегистрированный в системе обработчик событий для данного установленного бита и вызовет его. В данном случае будет вызвана системная процедура обработки нажатия кнопки «Power».

3. Далее все как в п.5. для SCI.

Попробуем разобраться в чем принципиальная разница между этими двумя способами. Во-первых, способ через SMI жестко привязан к ограниченному числу битов регистра PM1_STATUS, что не позволяет разработчикам аппаратуры (OEM) добавлять свои источники событий (свои кнопки и прочее) за рамками спецификации. Во-вторых, такие события от «Fixed Features» сущностей минуют виртуальную машину с кодом AML, а это не позволяет разработчикам «firmware» осуществлять более сложную обработку события до того как о нём станет известно операционной системе. Чтобы обойти эти ограничения в спецификацию ACPI был введен более обобщенный и универсальный способ не привязанный к конкретным кнопка и источникам - **«Control Method» (CM)**. В этом способе обработки, то, какой источник с каким событие будет ассоциирован, и то, как это событие будет обработано, полностью определяется проприетарным AML кодом «firmware».

В спецификации отмечается, что на многих вычислительных системах с поддержкой ACPI для ряда фич реализуются оба метода формирования событий, а режим задается конфигурационным битом SCI_EN, который устанавливается для систем совместимых с последней редакцией стандарта ACPI, либо остается в сброшенном состоянии для «ACPI legacy».

1.2. Аппаратные регистры ACPI

Так как для решения поставленной задачи нам далее придется немного «подебажить» ядро ОС FreeBSD (в части реализации ACPI), то считаю необходимым рассмотреть аппаратные ресурсы которыми оснащается подсистема ACPI.

Аппаратные ресурсы подсистемы ACPI доступны программно через набор регистров ввода/вывода. Напомню, что на платформе **x86_amd64** могут быть следующие виды регистров ввода/вывода, и заранее прошу не путать их с архитектурными регистрами процессора:

- Регистры доступные через порты ввода/вывода с помощью машинных инструкций **IN** и **OUT**. Номер порта (адрес) задается 16-ти битовым значением, а само значение (размер регистра) может быть 8, 16 и 32 бита.
- Регистры отображаемые в адресуемую память. Такие регистры адресуются 64-х битовым адресом (в AML коде используются два слова по 32 бита), а размер регистра может быть 8, 16, 32 и 64 бита.
- Регистры доступные через шины PCI/PCIe и отображаемые в основное адресное пространство процессора со смещением определяемым в BAR0-BARn. Адрес регистров внутри конфигурационного блока задается одним байтом (смещением), а размер регистра всегда составляет 32 бита. Адреса регистров отображаемые в память, на которые указывает BARn, могут иметь размер 32 или 64 бита, размер данных - 8, 16, 32 и 64 бита.
- Регистры доступные через шину SMBus (I^2C), где и адрес регистра и его размер обычно составляют один байт.

Ниже приведен перечень основных регистров используемы аппаратурой ACPI, их расположение и назначение.

Регистры «PM1 Event Registers»

Данные регистры располагаются на системной шине I/O и доступны через инструкции IN/OUT процессора (в x86_amd64). Адреса (номера) регистров и их размер задается в таблице FADT. Представляют собой группу из двух регистров статуса **PM1A_STS** и **PM1B_STS**. Каждый бит в регистре «A» и «B» отвечает за определенное фиксированное событие которое может формировать аппаратура. Чтобы получить значение из этих регистров, необходимо сначала считать оба регистра группы, выполнить операцию «ИЛИ» между ними, после чего выполнить операцию «И» над результатом и битовой маской задающей интересующий бит (или группу битов). Для установки или сброса соответствующего флага необходимо записать одинаковое значение как в **PM1A_STS**, так и в **PM1B_STS**, при этом для сбора соответствующее поле должно содержать «1», запись «0» игнорируется. Если ссылка на **PM1B_STS** в таблице FADT указывает на нулевое значение, значит этот регистр отсутствует и операцию «ИЛИ» выполнять не надо.

Когда в аппаратуре ACPI происходит событие, в регистре PM1x_STS один из соответствующих событию битов устанавливается в значение «1», и, если бит с таким же номером в регистра PM1x_EN установлен, то происходит аппаратное прерывание SMI или SCI. Назначение битов в регистре PM1x_STS следующее:

- Бит 0 — TMR_STS, флаг переноса таймера устанавливается всякий раз, когда старший (23-й или 31-й) бит счетчика таймера изменяет своё значение. Если в этот момент установлен бит 0 в регистре разрешения (TMR_EN), то формируется аппаратное прерывание.
- Биты 1-3 — зарезервированы.
- Бит 4 — BM_STS, статус мастера шины. Устанавливается в «1» когда мастер шины запрашивает доступ системной шине и таким образом индицирует активность мастера шины. Данные бит требуется для обеспечения когерентности памяти когда процессор находится в режиме энергосбережения C3. Данный бит сбрасывается программно записью «1» в эту же позицию.
- Бит 5 — GBL_STS, устанавливается когда произошло высвобождение глобального семафора блокировки «Global Lock», создает прерывание для вызова обработчика внутри «firmware». Данный механизм служит аппаратной поддержкой блокировок («мютексов») для AML кода.
- Биты 6-7 — зарезервированы.
- Бит 8 — PWRBTN_STS, опциональный бит, устанавливается когда происходит нажатие кнопки «Power Button». Если в этот момент установлен соответствующий разрешающий бит (PWRBTN_EN), то формируется аппаратное прерывание. Служит для индикации необходимости перевода системы в режим пониженного энергопотребления. Если система находится в режиме «сна» (S3, S4) или обесточена (S5), то выполняется «пробуждение» системы (WAKE). Бит устанавливается аппаратурой, но сбрасывается программно записью «1» в эту же позицию. Поддержка этой фичи задается в таблице FADT.
- Бит 9 — SLPBTN_STS, опциональный бит, устанавливается когда происходит нажатие кнопки «Sleep Button». Если в этот момент установлен соответствующий разрешающий бит (SLPBTN_EN), то формируется прерывание. Поведение полностью аналогично биту PWRBTN_STS.
- Бит 10 — RTC_STS, опциональный бит, устанавливается когда часы реального времени RTC формируют тревогу (срабатывает «будильник»). Если в этот момент установлен разрешающий бит RTC_EN, то формируется прерывание. Либо, если система находится в режиме «сна», то инициируется процесс «пробуждения» (WAKE). Бит устанавливается аппаратурой и сбрасывается программно записью «1» в эту же позицию.
- Бит 11 — должен быть игнорирован в ПО.
- Биты 12-13 — зарезервированы.
- Бит 14 — PCIEXP_WAKE_STS, опциональный бит для систем с PCI Express. Устанавливается аппаратурой PCIe индицирующий факт «пробуждения» PICe контроллера по входному сигналу WAKE#.
- Бит 15 — WAK_STS, устанавливается когда система находится в состоянии «сна» и произошло одно из пробуждающих событий. Устанавливается аппаратурой, сбрасывается программно записью «1» в эту же позицию.

Регистры «PM1 Enable Registers»

Данные регистры располагаются на системной шине I/O и доступны через инструкции IN/OUT процессора. Адреса (номера) регистров и их размер задается в таблице FADT. Представляют собой группу из двух регистров разрешения **PM1A_EN** и **PM1B_EN**. Каждый бит в регистре «A» и «B» отвечает за определенное фиксированное событие формируемое аппаратурой которое можно заблокировать. Для получения значения необходимо сначала считать оба регистра группы, выполнить операцию «ИЛИ» между ними, после чего выполнить операцию «И» над результатом и битовой маской задающей интересующий бит (или группу битов). Для установки или сброса бита необходимо записать одинаковое значение как в **PM1A_EN**, так и в **PM1B_EN**. Если ссылка на PM1B_EN в таблице FADT указывает на нулевое значение, значит этот регистр отсутствует. Когда соответствующий событию бит в этом регистре установлен в «1» и происходит событие, то аппаратура формирует аппаратное прерывание. Назначение битов следующее:

- Бит 0 — TMR_EN, разрешить прерывание по переполнению таймера.
- Биты 1-4 — зарезервированы.
- Бит 5 — GBL_EN — глобальный бит разрешения ACPI подсистемы.
- Биты 6-7 — зарезервированы.
- Бит 8 — PWRBTN_EN, опциональный бит разрешения прерывания от кнопки «Power Button». Данный бит должен игнорироваться (прерывания запрещаются) со стороны OSPM, если в таблице FADT установлен конфигурационный флаг PWR_BUTTON или в пространстве имен ACPI присутствует устройство PWRB.
- Бит 9 — SLPBTN_EN, опциональный бит разрешения прерывания от кнопки «Sleep Button». Данный бит должен игнорироваться (прерывания запрещаются) со стороны OSPM, если в таблице FADT установлен конфигурационный флаг SLP_BUTTON, или в пространстве имен ACPI присутствует устройство SLPB.
- Бит 10 — RTC_EN разрешает установку бита RTC_STS и формирование прерывания от часов RTC (будильник).
- Биты 11-13 — зарезервированы.
- Бит 14 — PCIEXP_WAKE_DIS, опциональный бит для чипсетов с портами PCIe. Установка данного бита запрещает пробуждение системы при установке бита PCIEXP_WAKE_STS. Изменение данного бита не влияет на установку бита PCIEXP_WAKE_STS.
- Бит 15 — зарезервирован.

Регистры «PM1 Control Registers»

Регистры этой группы также состоят из двух регистров **PM1A_CNT** и **PM1B_CNT**, логика работы с ними аналогична предыдущим регистрам. Битовые поля регистров данной группы содержат управляющие биты для фиксированных фич («Fixed Features») поддержка которых также включается или отключается в таблице FADT. Назначение битов данной группы регистров следующее:

- Бит 0 — SCI_EN, задает способ формирования событий: с помощью SMI или SCI прерываний (см. описание алгоритма формирования событий выше). Если данный бит установлен, то аппаратура ACPI будет формировать прерывание SCI с последуюшщим

вызовом управляющего AML метода, иначе — прерывание SMI. Бит устанавливается или сбрасывается аппаратурой, OSPM не может изменять этот бит.

- Бит 1 — BM_RLD, данный бит разрешает переводить процессоры из состояния «сна» (C3) в рабочее состояние (C0) по запросу от мастера шины.
- Бит 2 — GBL_RLS, бит только для записи. Запись «1» в этот бит вызывает прерывание SMI с целью вызвать обработчик и передать управление AML коду. OSPM устанавливает этот бит для индикации того, что произошло высвобождение глобального блока («Global Lock Release»).
- Биты 3-8 — зарезервированы.
- Бит 9 — должен игнорироваться.
- Биты 12:10 — SLP_TYPx, определяет тип (номер) состояния энергосбережения при установленном бите SLP_EN. С этим трех-битовым полем ассоциирован поименованный объект **_Sx** в коде AML.
- Бит 13 — SLP_EN, запись «1» в этот бит инициирует процесс перехода в режим энергосбережения заданный битовым полем SLP_TYPx.
- Биты 14-15 — зарезервированы.

Регистр «Power Management Timer»

PMTIMER содержит 24 бита (или 32 бита для новых систем) счетчика-таймера работающего на частоте 3.579545 МГц. Счетчик ведет счет только в режиме S0, и сбрасывается, когда ACPI переходит из любого состояния в S0. Каждый раз, когда происходит изменения старшего бита счетчика (23-го или 31-го), и установлен бит TMR_EN, происходит формирование аппаратного прерывания SMI или SCI с установкой бита статуса TMR_STS.

Регистр «PM2 Control Registers»

Содержит разрешающие биты для иных фич описанных в FADT. Если фича в FATD запрещена, то соответствующий ей бит в этом регистре игнорируется. Отдельное назначение у нулевого бита в этом регистре, он имеет название **ARB_DIS** и отвечает за работу системного (шинного) арбитра. Если ARB_DIS сброшен (равен «0»), то работа системного арбитра разрешена и арбитр управляет доступом к системной шине. Иначе системный арбитр отключен и шиной управляет центральный процессор.

Регистры «Processor Register Block»

Данный блок регистров содержит несколько одинаковых регистров управления для каждого процессора. В том числе:

- 32-х битовый регистр P_CNT - Биты 0-3 — CLK_VAL, определяет «зарезание» частоты процессора (clock throttling value). Бит 4 — THT_EN, разрешает зарезать частоту процессора.
- 8-ми битовый регистр P_LVL2 — чтение из этого регистра всегда возвращает нули и переводит процессор в режим C2. Запись игнорируется.
- 8-ми битовый регистр P_LVL3 — чтение из этого регистра всегда возвращает нули и переводит процессор в режим C3. Запись игнорируется.

Регистр «RESET_REG»

Представляет собой 8-ми битовый регистр для выполнения полного сброса, в том числе всех процессоров, всех шин и периферии (аналог цикла перезагрузки). Для того, чтобы выполнить сброс, необходимо записать в это регистр значение **RESET_VALUE** заданное в таблице FADT. Предполагается, что сброс происходит сразу после записи в регистр и операционная система более не выполняет своих функций, то есть должна перейти в вечный цикл ожидания сброса.

Регистры «Sleep Control and Status Registers»

Представляют собой группу из двух 8-ми битовых регистров SLEEP_CONTROL_REG и SLEEP_STATUS_REG предназначенных для ввода (и вывода) системы в один из режимов энергосбережения. Например, для того чтобы перейти в одно из состояний **Sx**, OSPM должна добыть значение объекта **_Sx** из таблиц, записать его в поле SLP_TYPx регистра SLEEP_CONTROL_REG и установить бит SLP_EN в этом же регистре. После чего OSPM непрерывно опрашивает бит **WAK_STS** регистра SLEEP_STATUS_REG дожидаясь возврата обратно в рабочий режим (значение «1» - система в рабочем режиме).

Назначение битов регистра SLEEP_CONTROL_REG:

- Бит 0 — зарезервирован.
- Бит 1 — игнорируется OSPM.
- Биты 2-4 — SLP_TYPx, устанавливает тип режима энергосбережения в который переходит система при установке бита SLP_EN.
- Биты 6-7 — зарезервированы.

Назначение битов регистра SLEEP_STATUS_REG:

- Биты 0-3 — игнорируются OSPM.
- Биты 4-5 — зарезервированы.
- Бит 6 — игнорируется OSPM.
- Бит 7 — WAK_STS, устанавливается когда система находится в одном из режимов энергосбережение («сна»), либо «0» - система в рабочем режиме. Данный бит устанавливается аппаратно и сбрасывается программно путем записи «1» в эту же позицию.

1.3. Поддержка произвольных источников событий (GPE)

Часть поддерживаемой периферии может формировать события в подсистеме ACPI через специальный механизм называемый «General-Purpose Events» (GPEs). Для этого в аппаратной поддержке ACPI выделяются отдельные сигнальные линии для подключения источников сигнала от внешней периферии. Состояния этих линий отображаются в 8-ми битовых регистрах **GPEx_STS**, а разрешение той или иной линии задается соответствующим битом в регистрах **GPEx_EN**. Оба регистра GPEx_STS и GPEx_EN образуют регистровый блок GPEx_BLK, которых спецификацией определено ровно два: **GPE0_BLK и GPE1_BLK**, их адреса и размерности также задаются в таблице FADT. Источником таких событий могут быть:

- Подключения/отключение внешних устройств: Dock-станция, AC-адаптер.

- Батарея.
- Подсистема контроля температуры (TZ0).
- Подача или отключение иных источников питания.
- Крышка мобильного устройства/ноутбука (Lid).
- Embedded Controller (EC0).
- События пробуждения (WAKE) от модема или сетевого адаптера.
- Подключение/отключение устройств Plug-and-Play.

Когда периферийное устройство желает сигнализировать о событии (например, произошло закрытие крышки ноутбука), то на соответствующую линию подается логический «0» (линия физически подтягивается к нулевому потенциалу, т. е. к GND), при этом если в регистре разрешения GPEx_EN соответствующий бит установлен в «1», то в регистре статуса GPEx_STS устанавливается соответствующий бит в «1» и ACPI формирует аппаратное прерывание SCI. Обработчик SCI внутри OSPM считывает номер установленного бита из регистра GPEx_STS и вызывает метод в AML коде для обработки этого события (**_GPE.Lxx** или подобный), передавая в него номер бита как номер события. Если в регистре статуса установлено несколько битов, то метод вызывается несколько раз с разным параметром. Сброс битов статуса после обработки осуществляется программно, записью «1» в соответствующие позиции регистра статуса.

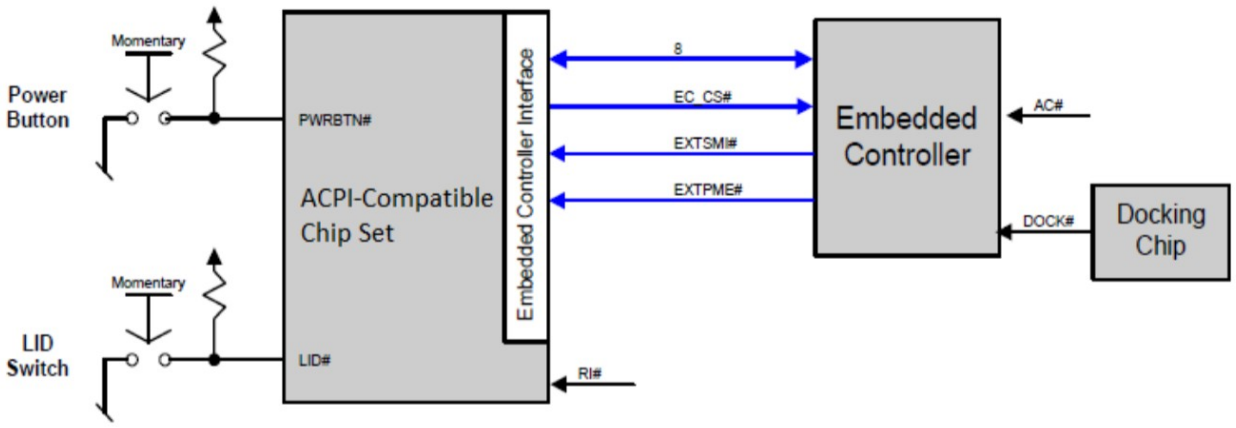

Рис. 3а. Схема подключение внешних источников GPE событий к ACPI.

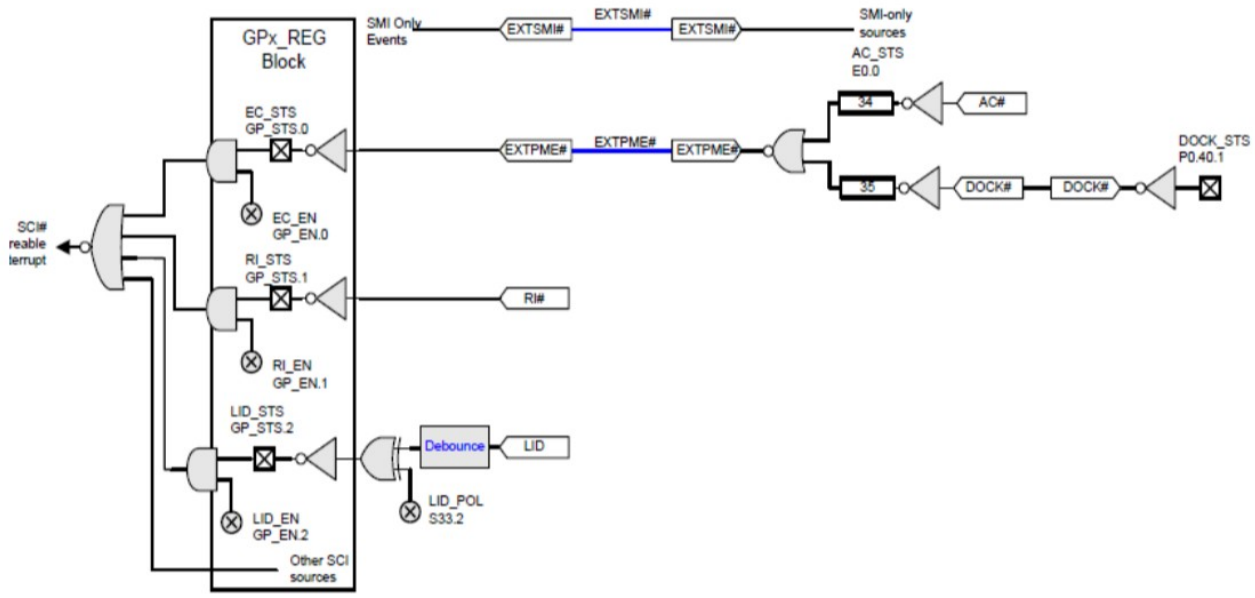

Рис. 3б. Логическая схема обработки GPE событий от внешних источников к ACPI.

Источником GPE события может служить и внутренний «Embedded Controller» (EC0), который может, например, мониторить уровень напряжения на линиях питания (AC) или батареи (DC). В этом случае EC0 передает по внутренней шине ECI в ACPI подсистему 8-ми битовый код события, этот код запоминается во внутреннем регистре аппаратуры ACPI. Далее, если по линии GPE, к которой подключен EC0, имеется разрешение (установлен соответствующий бит в GPEx_EN), то формируется аппаратное прерывание. OSPM вызывает метод в AML коде, где, обнаружив событие от EC0, процедура обработки события считывает 8-ми битовый код события из внутреннего регистра ACPI и производит необходимые манипуляции для обработки данного события (например, шлет нотификацию в ОС о том, что разряжена батарея). Выше на рис. 3а и 3б показан способ подключения источников GPE событий к ACPI.

1.4. Поддержка событий через GPIO

Как выяснилось, существует еще один способ формирования событий от внешних источников (в т.ч. может использоваться и для кнопки «Power») - через стандартные линии GPIO способные формировать аппаратные прерывания. Как я уже упоминал выше, со спецификации ACPI 5.0 вводится понятие «Hardware-Reduced ACPI». Это вычислительные системы с упрощенной системой управления питанием, в них отсутствуют сложные контроллеры обработки событий и нет аппаратуры поддержки ACPI. Кнопки «Power» и «Sleep», а так же датчик закрывания крышки ноутбука «LID» в таких вычислительных системах подключены к системному контроллеру GPIO, как правило, встроенному в систему-на-кристалле (SoC). Такой метод называется «GPIO-signaled ACPI Events».

Линии GPIO настраиваются на формирование аппаратного прерывания при изменении состояния кнопки или датчика. Обработчик прерываний GPIO внутри операционной системы должен понять от какого вывода (GPIO pin-a) поступило событие, для этого он считывает регистр состояния у GPIO контроллера, после чего производит вызов соответствующего метода (обычно это **_SB.GPIO._EVT**) в ACPI. Далее управление получает AML код внутри которого ведется обработка события, также как и в режиме «Control Method». Можно сказать,

что «Hardware-Reduced ACPI» это упрощенный вариант «СМ». Или же всё наоборот ? Так или иначе, вся работа по «маршрутизации» события возлагается на операционную систему, а аппаратная поддержка ACPI в таких вычислительных системах может отсутствовать.

На этом закончим обзор аппаратной части и приступим к программной.

1.5. ACPI Namespace

Для того, чтобы операционная система могла взаимодействовать с ACPI, например, переключать состояния процессоров C0-Cn, состояния устройств D0-D4 или всей системы S0-S5, а также для обнаружения возможностей (Capabilities) различных устройств, в OSPM предусмотрены механизмы отправки соответствующих сообщений основанные на вызове методов у объектов в AML коде. В ответ, объект, которому было отправлено сообщение, может выдать код статуса или целое ответное сообщение (Package). Для адресации объектов в ACPI вводится древовидная система имён (Namespace). Все сущности в этой системе имён разделяются на несколько разных типов. Адресуемой сущностью может быть как устройство типа «Device», так и любая виртуальная поименованная сущность отображаемая в AML коде в виде объекта с соответствующим набором поименованных методов.

Для именования узлов (объектов, устройств, методов, переменных и всего остального) в древовидной системе имен ACPI используются четырех-символьные обозначения узлов связанных символом «.» в длинный путь (Namepath). При именовании узлов в пути следуют следующим правилам:

- Пространство имён является общим для всех таблиц и загружаемого кода.
- Имя узла всегда состоит из четырех ASCII символов (или 32-х бит).
- Первый байт имени узла должен представлять ASCII символ из списка: ‘A’-‘Z’, ‘_’, (0x41-0x5A, 0x5F).
- Последующие три байта имени должны представлять ASCII символы из списка: ‘A’-‘Z’, ‘0’-‘9’, ‘_’, (0x41-0x5A, 0x30-0x39, 0x5F).
- Согласно конвенции, компилятор ASL подставляет (добавляет) лидирующие символы подчеркивания (‘_’) если имя узла состоит менее чем из 4-х символов.
- Имена узлов (объектов) начинающихся с подчеркивания (‘_’) зарезервированы спецификацией ACPI под служебные нужны, существует таблица служебных объектов.
- Если имя начинается с символа «обратный слэш» (‘\’), то это указывает на корень дерева.
- Если имя начинается с символа «крышка» (‘^’), то это указывает на то, что последующая ветка дерева выстроена относительно предыдущего (родительского) уровня иерархии (аналог двух точек ‘..’ в файловый системах). Символов ‘^’ может присутствовать несколько штук, каждый из них ссылается на один уровень иерархии выше. Таким образом по дереву имён можно ссылаться относительно, также как в файловой системе.
- Если имя объекта дано без символов ‘\’, ‘^’ или без ‘.’, то такое имя задано относительно текущей области видимости. Если в текущей области видимости объекта с таким именем не обнаруживается, то осуществляется поиск в предыдущем уровне (parent level), и так далее пока поиск не упрется в корень дерева. В это случае

считается что объект с заданным именем считается отсутствующем в пространстве имен.

- Область видимости можно изменять оператором **Scope()**.
- При объявлении новых объектов с составным путем, вся предыдущая ветка пути должна заранее существовать.

Примеры адресации объектов в иерархическом пространстве имен ACPI приведены в листинге 1а ниже.

Листинг 1а. Примеры адресации объектов в пространстве имен ACPI.

```
ABCD // Относительно текущей области видимости с поиском по всему дереву вверх.
^ABCD // Относительно области видимости предка, без поиска по дереву.
^^ABCD // Относительно области видимости предка у предка, без поиска по дереву.
XYZ.ABCD // Относительно текущей области видимости, без поиска по дереву.
\XYZ.ABCD // Относительно корня, без поиска по дереву.
Device (XYZ.ABCD) {…} // Создать объект типа Device с именем ABCD в области (ветке) XYZ адресуемой относительно текущей области видимости.
OperationRegion (\XYZ.ABCD, ...) // Создать объект типа OperationRegion с именем ABCD с указанием абслютного пути.
Scope (\XYZ) { Name (BAR, 5) } // Создает глобальную переменную \XYZ.BAR со значением 5.
```

Более подробно об операторах **Scope()**, **Device()**, **Name()** и **OperationRegion()** будет сказано далее при рассмотрении языка AML машины.

1.5.1. Корневые пространства имен

Спецификацией ACPI определяются следующие пути (Namepaths) первого уровня или «Root Namespaces» - так они называются в спецификации:

- **_GPE** — Для «General Purpose Events», их регистров и обработчиков событий.
- **_PR** — Для объектов описывающих присутствующие в вычислительной системе процессоры*.
- **_SB** — Все устройства (Devices), шины (Buses) и их процедуры располагаются в этом пространстве.
- **_SI** — т. н. «System Indicators», см. спецификацию ACPI.
- **_TZ** — Термальные зоны: датчики температуры, вентиляторы и их процедуры обработки*.
- **_PIDx** — Ресурсы управления питанием для подсистемы IDE дисков (давно не используется).

* Со спецификации ACPI версии 2.0 допускается нахождение объектов пространств **_PR** и **_TZ** внутри пространства **_SB**. На рис. 3в приведен пример дерева пространства имен ACPI.

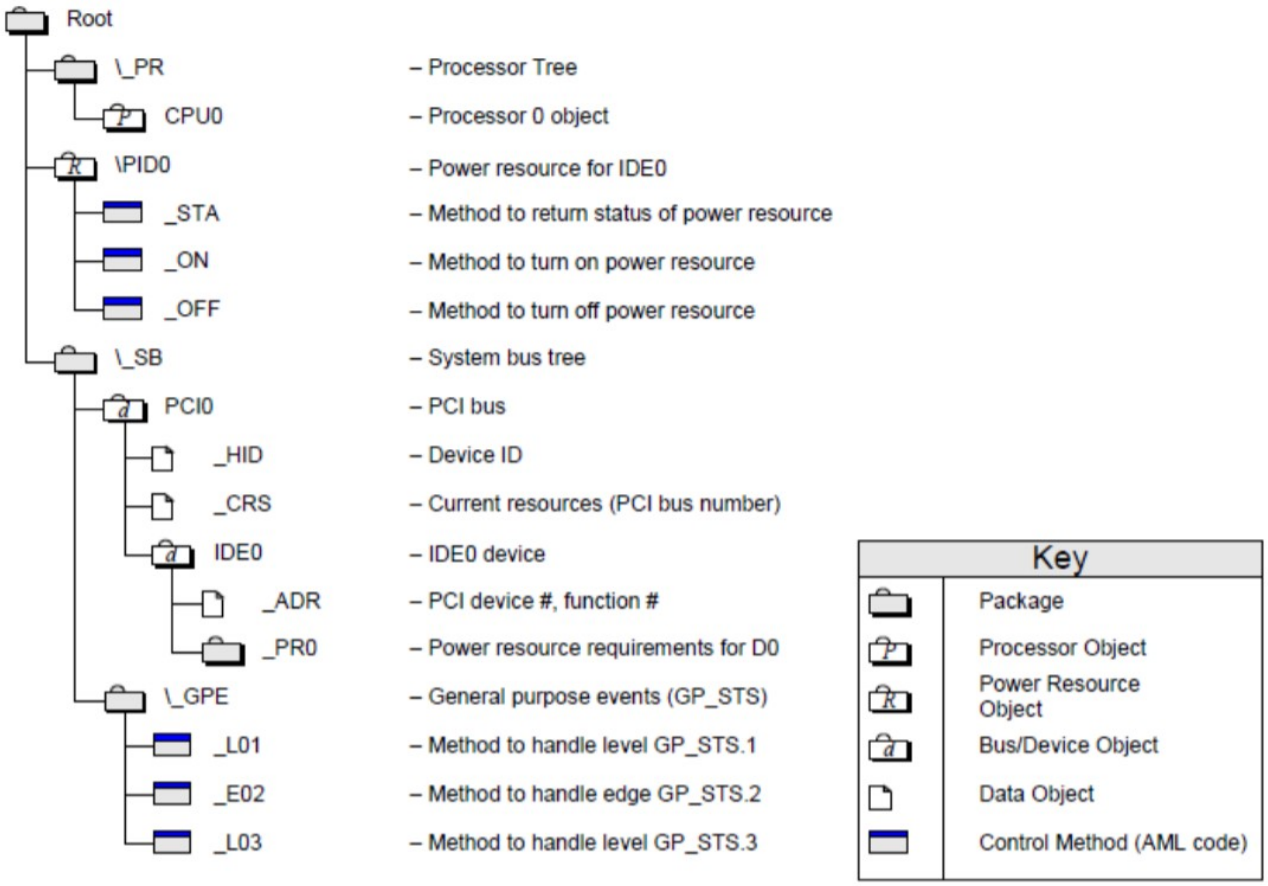

Рис. 3в. Пример дерева имен ACPI.

Важное замечание! Отличить объект типа «устройство» от типа «метода» по его именованию и нахождению в пространстве имен не представляется возможным. При отправки сообщения всегда используется именованная сущность типа «метод», поэтому необходимо знать какие методы и у каких устройств имеются в распоряжении. Ниже мы рассмотрим ряд стандартных (зарезервированных) устройств и методов для работы с ними.

1.5.2. Служебные объекты и их методы

Спецификацией ACPI определяется большое количество зарезервированных объектов с короткими именами (символом ‘_’ в начале имени) используемых под различные служебные нужны. Не все из них используются при разработке «firmware», но некоторые присутствуют почти в любом OEM изделии. Существует стандартный набор методов применяемых почти к любому объект типа «Device», позволяющих выполнять базовые операции, такие как инициализация, получение списка ресурсов, получение и изменение статуса. Ниже в таблице 1а приведен перечень таких универсальных методов.

В таблицах 1б, 1в, 1г и 1д приведены другие часто используемые объекты. Среди них интерес для нас могут представлять объекты **_SB.PWRB** и **_SB.GPIO**. Первый информирует о наличии/отсутствии и статусе кнопки «Power Button», второй — позволяет управлять линиями GPIO к которым обычно подключается кнопка «Power Button» на упрощенных (Hardware-Reduced) ACPI системах.

Таблица 1а. Перечень общих методов для всех объектов типа «Device».

Путь в пространстве имен	Имя метода	Функция метода
_SB.XXXX._HID	Hardware ID	Возвращает уникальную строку идентификатора устройства Plug-and-Play.
_SB.XXXX._ADR	Address	Определяет физический адрес устройства на родительской шине (например, слот PCI).
_SB.XXXX._STA	Status	Определяет, включено ли устройство, присутствует ли оно, правильно ли оно функционирует.
_SB.XXXX._CRS	Current Resource Settings	Возвращает буфер с описанием ресурсов устройства (IRQs, I/O, MMIO, GPIO).
_SB.XXXX._PRS	Possible Resource Settings	Список дополнительных аппаратных ресурсов, которые настраиваемое устройство способно выделить.
_SB.XXXX._SRS	Set Resource Settings	Настраивает или изменяет распределение аппаратных ресурсов устройства.
_SB.XXXX._DIS	Disable	Полностью отключает целевой объект устройства.
_SB.XXXX._DSM	Device Specific Method	Индивидуальный метод для «кастомных» конфигураций оборудования от производителя.

Таблица 1б. Основные системные и процессорные устройства.

Путь в пространстве имен	Название устройства	Аппаратный ID (_HID)	Ключевые методы	Функция метода
_SB	System Bus	ACPI0015	_INI	Инициализация системной шины
_SB	System Bus	ACPI0015	_STA	Получение статуса системной шины
_SB.PR00	Processor Container	ACPI0010	_PTC	Управление типом обработки
_SB.PR00	Processor Container	ACPI0010	_PSS	Состояния производительности процессора
_SB.PR00	Processor Container	ACPI0010	_CPC	Непрерывное управление производительностью
_SB.PR00.CP00	Processor Device	ACPI0007	_PPC	Управление производительностью процессора

_SB.PR00.CP00	Processor Device	ACPI0007	_CST	C-состояния процессора (энергосбережение)
_SB.PCI0	PCI Express Bus	PNP0A08	_STA	Получение статуса шины PCI
_SB.PCI0	PCI Express Bus	PNP0A08	_CRS	Получение буфера с текущими ресурсами шины PCI
_SB.PCI0	PCI Express Bus	PNP0A08	_PRT	Таблица маршрутизации PCI
_SB.PCI0.SBR.EC0	Embedded Controller Device	PNP0C09	_GPE	Обработка GPE событий

Таблица 1в. Устройства питания и батареи.

Путь в пространстве имен	Название устройства	Аппаратный ID (_HID)	Ключевые методы	Функция метода
_SB.ACDC	ACDC Device / Power Source	ACPI0003	_PSR	Статус источника питания (сеть)
_SB.BAT0	Control Method Battery	PNP0C0A	_BIF	Информация о батарее
_SB.BAT0	Control Method Battery	PNP0C0A	_BIX	Расширенная информация о батарее
_SB.BAT0	Control Method Battery	PNP0C0A	_BST	Статус батареи (скорость разряда, напряжение)
_SB.BAT0	Control Method Battery	PNP0C0A	_BTP	Контрольная точка батареи (пороги срабатывания)
_SB.CHG0	Smart Battery Charger	ACPI0002	_BXC	Матрица возможностей зарядного устройства
_SB.GPIO	GPIO Signaled Power Device	AMDI0030	_EVT	Передача прерываний для событий питания
_SB.GPIO	GPIO Signaled Power Device	AMDI0030	_STA	Оценка статуса устройства питания на базе GPIO

Таблица 1г. Термозоны и устройства охлаждения.

Путь в пространстве имен	Название устройства	Аппаратный ID (_HID)	Ключевые методы	Функция метода
_TZ.TZ01	Thermal Zone		_CRT	Критическая точка температуры
_TZ.TZ01	Thermal Zone		_HOT	Точка высокой температуры (перегрев)
_TZ.TZ01	Thermal Zone		_TMP	Текущее чтение температуры
_TZ.TZ01	Thermal Zone		_PSV	Точка пассивного охлаждения
_TZ.TZ01	Thermal Zone		_ACx	Точка активного охлаждения
_SB.FAN0	Fan / Cooling Device	PNP0C0B	_FST	Статус вентилятора (скорость, питание)
_SB.FAN0	Fan / Cooling Device	PNP0C0B	_FIF	Информационные поля вентилятора

Таблица 1д. Устройства ввода и пользовательского интерфейса.

Путь в пространстве имен	Название устройства	Аппаратный ID (_HID)	Ключевые методы	Функция метода
_SB.ALS0	Ambient Light Sensor	ACPI0008	_ALI	Освещенность датчика внешнего света (Люкс)
_SB.ALS0	Ambient Light Sensor	ACPI0008	_ALR	График реакции на изменение света
_SB._LID	Lid Device	PNP0C0D	_LID	Определяет, открыта или закрыта крышка ноутбука
_SB.PWRB	Power Button	PNP0C0C	_STA	Определяет наличие и статус кнопки питания
_SB.SLPB	Sleep Button	PNP0C0E	_STA	Определяет наличие кнопки спящего режима

1.5.3. Hardware ID (Plug-and-Play ID)

Как-то так получилось, что древовидного пространства имен оказалось недостаточно для того, чтобы операционная система могла однозначно ассоциировать обработчик событий с устройством (или целым классом устройств) от которого эти события поступают. Дело в том, что спецификация ACPI не накладывает жесткого ограничения на структуру дерева имен

(за исключением ряда моментов с вязанных с «корневым» пространством имен), а это означает, что каждый OEM производитель вправе придумывать свои, удобные для него структуры и давать устройствам свои имена.

Для того, чтобы операционная система могла отличать события поступающие от различных устройств, им (устройствам) присваиваются специальные строковые идентификаторы называемые «Hardware ID» или «Plug-and-Play ID» (так как используется операционной системой для автоматического подбора драйвера). Этот уникальный идентификатор получил название HID, а метод для получения идентификатора от объекта по его пути - **_HID**. HID идентификаторы для широко используемых устройств определены спецификацией ACPI, в таблицах 1б-1д выше можно наблюдать HID для приведенных устройств в графе «Аппаратный ID (_HID)». Консорциум UEFI, в которые входит рабочая группа ACPI, ведет учет HID идентификаторов, любой производитель OEM может зарегистрировать HID для своего устройства если оно уникально и достаточно распространено.

Спецификацией ACPI предусматривается **четыре** вида HID идентификаторов:

- **Plug-and-Play (PNP) ID** — имеет вид "**AAA####**", где символом **A** представлен любой символ из таблицы ASCII в диапазоне [A-Z] (заглавные буквы), а символом **#** - шестнадцатеричные цифры. Пример: "**PNP0C0C**".
- **ACPI ID** — имеет вида "**NNNN####**", где вместо NNNN используется ASCII строка "ACPI". Пример: "**ACPI0003**".
- **Vendor-defined ID** — аналогично предыдущему, но вместо первых четырех символов занимает четырех-буквенный идентификатор OEM производителя. Пример: "**MSFT0003**".
- Сжатый EISA тип — 32-х битное число.

Кнопке «Power» присвоен HID идентификатор со строкой символов "**PNP0C0C**", запомним его на всякий случай!

1.5.4. Уникальный идентификатор устройства (UUID)

Как бы парадоксально это не звучало, но четырех видов HID идентификаторов и древовидного пространства имён тоже оказалось недостаточно. OEM производители захотели иметь свободу в обозначении своих устройств без регистрации, но так, чтобы эти обозначения имели уникальные и непересекающиеся идентификаторы. С версии ACPI 5.0 вводится такое понятие как «Universally Unique ID» (сокращенно UUID или GUID) — это глобально уникальный номер состоящий из 16 байт и представляемый обычно в шестнадцатеричном виде. OEM производитель может присваивать своим устройствам произвольные UUID.

В рамках ACPI устройство может иметь одновременно и UUID и HID идентификаторы. Более того, UUID идентификаторы могу присваиваться методам вызываемым через универсальный метод **_DSM** (Device-Specific Method). Для того, чтобы операционная система могла работать с такими устройствами или вызывать их методы, в ней должен присутствовать список известных идентификаторов UUID. Ниже в листинге 1б приведены некоторые из идентификаторов UUID, выбранные из системного заголовочного файла **acpica/include/acuuid.h** входящего в исходный код ACPICA. Каким образом UUID от различных OEM производителей попадают в этот список — не совсем понятно, видимо тоже через участие в консорциуме, но тогда чем это отличается от HID ? :-)

У объектов типа «Device» предусматривается наличие метод **_UID** позволяющего получить UUID идентификатор этого устройства по аналогии с методом _HID.

Листинг 1б. Примеры UUID для некоторых устройств (файл acuuid.h).

```
/* Controllers */

#define UUID_GPIO_CONTROLLER            "4f248f40-d5e2-499f-834c-27758ea1cd3f"
#define UUID_USB_CONTROLLER             "ce2ee385-00e6-48cb-9f05-2edb927c4899"
#define UUID_SATA_CONTROLLER            "e4db149b-fcfe-425b-a6d8-92357d78fc7f"

/* Devices */
#define UUID_PCI_HOST_BRIDGE            "33db4d5b-1ff7-401c-9657-7441c03dd766"
#define UUID_I2C_DEVICE                 "3cdff6f7-4267-4555-ad05-b30a3d8938de"
#define UUID_POWER_BUTTON               "dfbcf3c5-e7a5-44e6-9c1f-29c76f6e059c"
#define UUID_MEMORY_DEVICE              "03b19910-f473-11dd-87af-0800200c9a66"
#define UUID_GENERIC_BUTTONS_DEVICE     "fa6bd625-9ce8-470d-a2c7-b3ca36c4282e"
#define UUID_NVDIMM_ROOT_DEVICE         "2f10e7a4-9e91-11e4-89d3-123b93f75cba

/* TPM */
#define UUID_HARDWARE_INFORMATION       "cf8e16a5-c1e8-4e25-b712-4f54a96702c8"
#define UUID_START_METHOD               "6bbf6cab-5463-4714-b7cd-f0203c0368d4"
#define UUID_MEMORY_CLEAR               "376054ed-cc13-4675-901c-4756d7f2d45d"

/* Processor Properties (ACPI 6.2) */
#define UUID_CACHE_PROPERTIES           "6DC63E77-257E-4E78-A973-A21F2796898D"
#define UUID_PHYSICAL_PROPERTY          "DDE4D59A-AA42-4349-B407-EA40F57D9FB7"
```

Как видно, у кнопки «Power» также имеется свой UUID: "**dfbcf3c5-e7a5-44e6-9c1f-29c76f6e059c**".

1.6. Таблицы с «Firmware»

Как я уже отмечал выше, весь проприетарный код «Firmeware», вместе с данным и настройками, передается при загрузке операционной системы от UEFI BIOS к OSPM в виде специализированных «таблиц», которые принято называть DSDT (от «Differentiated System Description Table»). На самом деле это название не совсем верное, так как DSDT это только одна из нескольких десятков таблиц, но это название широко закрепилось в среде разработчиков операционных систем, да и таблицами эти структуры тоже назвать сложно, они больше походят на древовидный список объектов. Доступен этот список объектов уже на стадии исполнения загрузчика операционной системы. Более того, у загрузчиков современных ОС имеются механизмы для подмены данных в DSDT таблицах, частично или полностью, и это очень важная фича. Для чего может потребоваться подмена DSDT я расскажу чуть позже, а сейчас посмотрим на этот список таблиц, разберем какие именно данные в нём представлены и как операционная система обнаруживает его в памяти.

Когда операционной системе (или её загрузчику) требуется считать содержимое DSDT таблиц, она делает сканирование некоторых областей ОЗУ на наличие специальной сигнатуры «Root System Description Pointer» (**RSDP**), которая состоит из 8-ми байт и представляет следующую ASCII строку: **"RSD PTR "** (в конце — символ пробела). На старых IBM PC совместимых ЭВМ поиск RSDP осуществляется в 1-м килобайте т. н. «Extended BIOS Data Area» (EBDA). Ссылка на EBDA в таких машинах записывается в

ячейку памяти со смещением 40:0Eh относительно BIOS Data Area» которая обычно располагается в самом начале адресного пространства. Если там ничего не обнаружено, то поиск продолжается в диапазоне адресов 0E0000h — 0FFFFFh (напомню, что всё это происходит в «реальном» режиме процессора). В современных вычислительных системах, с UEFI BIOS, указатель на структуру RSDP передается UEFI-совместимому загрузчику в момент его старта через «EFI System Table». Структура RSDP представлена в Таблице 2a.

Таблица 2a. Структура Root System Description Pointer» (RSDP).

Field	Byte Length	Byte Offset	Description
Signature	8	0	"RSD PTR " (Notice that this signature must contain a trailing blank character.)
Checksum	1	8	This is the checksum of the fields defined in the ACPI 1.0 specification. This includes only the first 20 bytes of this table, bytes 0 to 19, including the checksum field. These bytes must sum to zero.
OEMID	6	9	An OEM-supplied string that identifies the OEM.
Revision	1	15	The revision of this structure. Larger revision numbers are backward compatible to lower revision numbers. The ACPI version 1.0 revision number of this table is zero. The ACPI version 1.0 RSDP Structure only includes the first 20 bytes of this table, bytes 0 to 19. It does not include the Length field and beyond. The current value for this field is 2.
RsdtAddress	4	16	32 bit physical address of the RSDT.
Length*	4	20	The length of the table, in bytes, including the header, starting from offset 0. This field is used to record the size of the entire table. This field is not available in the ACPI version 1.0 RSDP Structure.
XsdtAddress*	8	24	64 bit physical address of the XSDT.
Extended Checksum*	1	32	This is a checksum of the entire table, including both checksum fields.
*Reserved**	3	33	Reserved field

Загрузчик ОС добывает из RSDP адрес другой таблицы: для 32-х битной системы это «Root System Description Table» (**RSDT**), а для 64-х битных систем - «eXtended System Description Table» (**XSDT**).

Структура RSDT/XSDT содержит общую информацию о производителе (OEM), о версии «firmware» и т. д., она являются корнем списка и ссылается на другие поименованные структуры. На рис. 4 показана связь между разными частями этого списка таблиц в обобщенном виде.

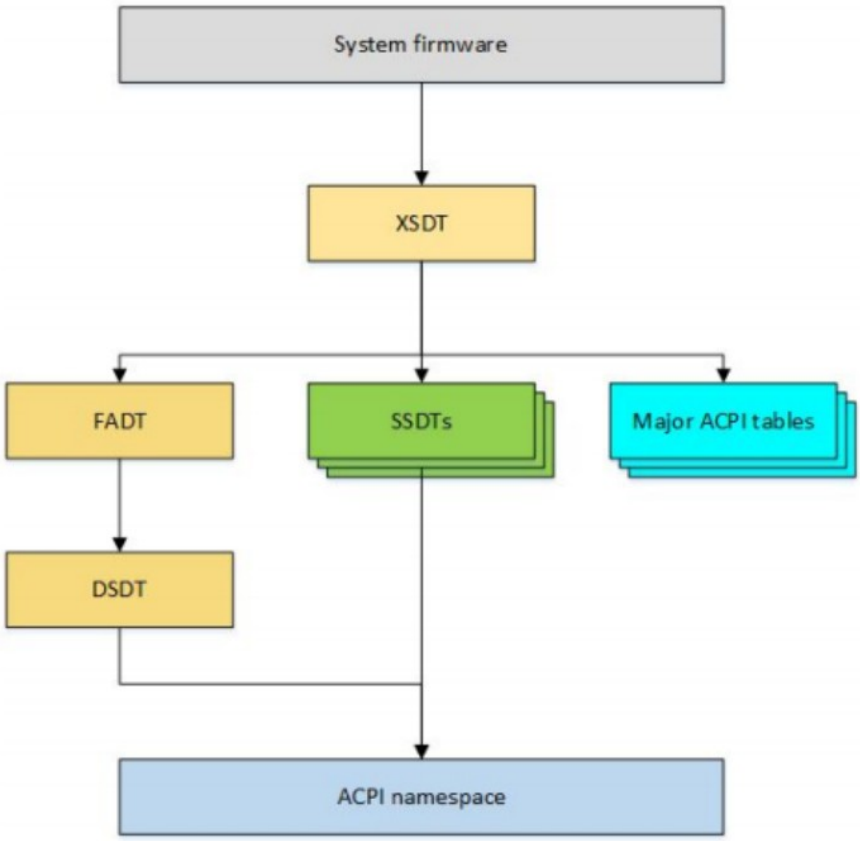

Рис. 4. «Firmware» представляется несколькими таблицами (структурами) ссылающимися друг на друга.

Все таблицы в списке, включая RSDT/XSDT, имеют одинаковый заголовок называемый «System Description Table Header». Первые четыре байта этого заголовка всегда содержат сигнатуру (Signature) таблицы представленную в виде ASCII символов. Заголовок также содержит общую длину таблицы в байтах (Length) и номер версии (Revision) структуры. Также в заголовке представлена информация о компиляторе AML или утилите (Creator ID), с помощью которого был получен бинарный код содержащийся в таблице, и его версия (Creator Revision). Формат общего заголовка представлен в Таблице 2б.

Таблица RSDT/XSDT сразу после заголовка содержит массив из 32-х или 64-х битных указателей задающих физические адреса размещения других таблиц из списка. Этих указателей может быть произвольное количество, их число определяется размером данной таблицы.

Таблица 2б. Формат общего заголовка всех таблиц в списке DSDT.

Field	Byte Length	Byte Offset	Description
Signature	4	0	The ASCII string representation of the table identifier. Note that if OSPM finds a signature in a table that is not listed in Table 5.5, then OSPM ignores the entire table (it is not loaded into ACPI namespace); OSPM ignores the table even though the values in the Length and Checksum fields are correct.
Length	4	4	The length of the table, in bytes, including the header, starting from offset 0. This field is used to record the size of the entire table.
Revision	1	8	The revision of the structure corresponding to the signature field for this table. Larger revision numbers are backward compatible to lower revision numbers with the same signature.
Checksum	1	9	The entire table, including the checksum field, must add to zero to be considered valid.
OEMID	6	10	An OEM-supplied string that identifies the OEM.
OEM Table ID	8	16	An OEM-supplied string that the OEM uses to identify the particular data table. This field is particularly useful when defining a definition block to distinguish definition block functions. The OEM assigns each dissimilar table a new OEM Table ID.
OEM Revision	4	24	An OEM-supplied revision number. Larger numbers are assumed to be newer revisions.
Creator ID	4	28	Vendor ID of utility that created the table. For tables containing Definition Blocks, this is the ID for the ASL Compiler.
Creator Revision	4	32	Revision of utility that created the table. For tables containing Definition Blocks, this is the revision for the ASL Compiler.

Спецификацией ACPI определяется (резервируется) более 50-ти сигнатур для различных структур и кода программного обеспечения «firmware», но не все они используются. Далее мы рассмотрим только четыре структуры, которые обязательно присутствуют в любом «firmware», а именно:

- FADT (она же FACP) — «Fixed ACPI Description Table»;
- FACS — «Firmware ACPI Control Structure»;
- DSDT — «Differentiated System Description Table»;
- SSDT — «Secondary System Description Table».

1.6.1. Таблица «Fixed ACPI Description Table» (FADT)

Таблица FADT начинается со строки "FACP" в поле сигнатуры. Так называлась эта таблица до ACPI v1.0 и с целью сохранения совместимости сигнатуру не стали менять. FADT — это описательная таблица, она содержит информацию о поддерживаемых подсистемой ACPI фича и методах их реализации (битовые поля или флаги), а так же

содержит адреса основных блоков регистров: PM1a_EVT_BLK, PM1b_EVT_BLK, PM1a_CNT_BLK, PM1b_CNT_BLK, PM2_CNT_BLK, PM_TMR_BLK, GPE0_BLK и GPE1_BLK. Также эта таблица содержит указатель на другую таблицу - **DSDT**.

Изначально спецификация ACPI разрабатывалась для 32-х битной платформы x86, позже она была расширена для 64-х битных машин. Поэтому поля с указателями и адресами в этой таблице имеются два значения — одно 32-х битное, другое — 64-х битное. Для 64-х битных полей к имени добавляется префикс **X_**. Например, в FADT есть поле **DSDT** и **X_DSDT**. Если в данной платформе используется 32-х битная адресация, то задействуется поле DSDT, иначе используется поле X_DSDT. В спецификации делается следующее важное замечание относительно адресации: *если поле с префиксом X_ имеет ненулевое значение, то следует использовать поле с префиксом X_ даже на 32-х битной платформе игнорируя аналогичное 32-х битное поле.*

Для «Hardware-Reduced» платформ поля содержащие адреса регистров могут игнорироваться. В спецификации указываются следующие диапазоны смещений которые не используются для таких платформ: байты **с 46 по 108**, и байты **с 148 по 232**. Также не используются следующие биты в поле флагов: **1, 2, 3,7,8,13, 14, 16 и 17**.

Так как далее нам придется обращаться к данным из таблицы FADT, то ниже в Таблице 2в приведено полное описание FADT в том виде как оно представлено в спецификации ACPI 6.6. Длина таблицы вместе со стандартным заголовком составляет **276** байт.

Таблица 2в. Структура таблицы «Fixed ACPI Description Table» (FADT).

Field	Byte Length	Byte Offset	Description
Header			
• Signature	4	0	'FACP'. Signature for the Fixed ACPI Description Table. (This signature predates ACPI 1.0, explaining the mismatch with this table's name.)
• Length	4	4	Length, in bytes, of the entire FADT.
FADT Major Version	1	8	6 Major Version of this FADT structure, in "Major.Minor" form, where 'Minor' is the value in the Minor Version Field (Byte offset 131 in this table) It is the intention that everything contained in the ACPI table would comply with what is contained in the ACPI specification itself. The FADT Major and Minor version follow in lock-step with the version of the ACPI Specification. Conforming to a given ACPI specification means that each and every ACPI-related table conforms to the version number for that table that is listed in that version of the specification.

Checksum	1	9	Entire table must sum to zero.
OEMID	6	10	OEM ID
OEM Table ID	8	16	For the FADT, the table ID is the manufacture model ID. This field must match the OEM Table ID in the RSDT.
OEM Revision	4	24	OEM revision of FADT for supplied OEM Table ID.
Creator ID	4	28	Vendor ID of utility that created the table. For tables containing Definition Blocks, this is the ID for the ASL Compiler.
Creator Revision	4	32	Revision of utility that created the table. For tables containing Definition Blocks, this is the revision for the ASL Compiler.
FIRMWARE_CTRL	4	36	Physical memory address of the FACS, where OSPM and Firmware exchange control information. See Section 5.2.10 for more information about the FACS. If the X_FIRMWARE_CTRL field contains a non zero value which can be used by the OSPM, then this field must be ignored by the OSPM. If the HARDWARE_REDUCED_ACPI flag is set, and both this field and the X_FIRMWARE_CTRL field are zero, there is no FACS available.
DSDT	4	40	Physical memory address of the DSDT. If the X_DSDT field contains a non-zero value which can be used by the OSPM, then this field must be ignored by the OSPM.
Reserved	1	44	ACPI 1.0 defined this offset as a field named INT_MODEL, which was eliminated in ACPI 2.0. Platforms should set this field to zero but field values of one are also allowed to maintain compatibility with ACPI 1.0.
Preferred_PM_Profile	1	45	This field is set by the OEM to convey the preferred power management profile to OSPM. OSPM can use this field to set default power management policy parameters during OS installation. Field Values: 0 Unspecified 1 Desktop 2 Mobile 3 Workstation 4 Enterprise Server 5 SOHO Server 6 Appliance PC 7 Performance Server 8 Tablet >8 Reserved
SCI_INT	2	46	System vector the SCI interrupt is wired to in 8259 mode. On systems that do not contain the 8259, this field contains the Global System Interrupt number of the SCI interrupt. OSPM is required to treat the ACPI SCI interrupt as a shareable, level, active low interrupt.

SMI_CMD	4	48	System port address of the SMI Command Port. During ACPI OS initialization, OSPM can determine that the ACPI hardware registers are owned by SMI (by way of the SCI_EN bit), in which case the ACPI OS issues the ACPI_ENABLE command to the SMI_CMD port. The SCI_EN bit effectively tracks the ownership of the ACPI hardware registers. OSPM issues commands to the SMI_CMD port synchronously from the boot processor. This field is reserved and must be zero on system that does not support System Management mode.
ACPI_ENABLE	1	52	The value to write to SMI_CMD to disable SMI ownership of the ACPI hardware registers. The last action SMI does to relinquish ownership is to set the SCI_EN bit. During the OS initialization process, OSPM will synchronously wait for the ntransfer of SMI ownership to complete, so the ACPI system releases SMI ownership as quickly as possible. This field is reserved and must be zero on systems that do not support Legacy Mode.
ACPI_DISABLE	1	53	The value to write to SMI_CMD to re-enable SMI ownership of the ACPI hardware registers. This can only be done when ownership was originally acquired from SMI by OSPM using ACPI_ENABLE. An OS can hand ownership back to SMI by relinquishing use to the ACPI hardware registers, masking off all SCI interrupts, clearing the SCI_EN bit and then writing ACPI_DISABLE to the SMI_CMD port from the boot processor. This field is reserved and must be zero on systems that do not support Legacy Mode.
S4BIOS_REQ	1	54	The value to write to SMI_CMD to enter the S4BIOS state. The S4BIOS state provides an alternate way to enter the S4 state where the firmware saves and restores the memory context. A value of zero in S4BIOS_F indicates S4BIOS_REQ is not supported. (See Section 5.2.10)
PSTATE_CNT	1	55	If non-zero, this field contains the value OSPM writes to the SMI_CMD register to assume processor performance state control responsibility.
PM1a_EVT_BLK	4	56	System port address of the PM1a Event Register Block. See Section 4.8.3.1 for a hardware description layout of this register block. This is a required field. If the X_PM1a_CNT_BLK field contains a non zero value which can be used by the OSPM, then this field must be ignored by the OSPM.
PM1b_EVT_BLK	4	60	System port address of the PM1b Event Register Block. See Section 4.8.3.1 for a hardware description layout of this register block. This field is optional; if this register block is not supported, this field contains zero. If the X_PM1b_EVT_BLK field contains a non zero value which can be used by the OSPM, then this field must be ignored by the OSPM.

PM1a_CNT_BLK	4	64	System port address of the PM1a Control Register Block. See Section 4.8.3.1 for a hardware description layout of this register block. This is a required field. If the X_PM1a_CNT_BLK field contains a non zero value which can be used by the OSPM, then this field must be ignored by the OSPM.
PM1b_CNT_BLK	4	68	System port address of the PM1b Control Register Block. See Section 4.8.3.1 for a hardware description layout of this register block. This field is optional; if this register block is not supported, this field contains zero. If the X_PM1b_CNT_BLK field contains a non zero value which can be used by the OSPM, then this field must be ignored by the OSPM.
PM2_CNT_BLK	4	72	System port address of the PM2 Control Register Block. See Table 4.4 for a hardware description layout of this register block. This field is optional; if this register block is not supported, this field contains zero. If the X_PM2_CNT_BLK field contains a non zero value which can be used by the OSPM, then this field must be ignored by the OSPM.
PM_TMR_BLK	4	76	System port address of the Power Management Timer Control Register Block. See the Section 4.8.3.3 for a hardware description layout of this register block. This is an optional field; if this register block is not supported, this field contains zero. If the X_PM_TMR_BLK field contains a non-zero value which can be used by the OSPM, then this field must be ignored by the OSPM.
GPE0_BLK	4	80	System port address of General-Purpose Event 0 Register Block. See Section 4.8.4.1 for more information. If this register block is not supported, this field contains zero. If the X_GPE0_BLK field contains a nonzero value which can be used by the OSPM, then this field must be ignored by the OSPM.
GPE1_BLK	4	84	System port address of General-Purpose Event 1 Register Block. See Section 4.8.4.1 for more information. This is an optional field; if this register block is not supported, this field contains zero. If the X_GPE1_BLK field contains a nonzero value which can be used by the OSPM, then this field must be ignored by the OSPM.
PM1_EVT_LEN	1	88	Number of bytes decoded by PM1a_EVT_BLK and, if supported, PM1b_ EVT_BLK. This value is >= 4.
PM1_CNT_LEN	1	89	Number of bytes decoded by PM1a_CNT_BLK and, if supported, PM1b_CNT_BLK. This value is >= 2.
PM2_CNT_LEN	1	90	Number of bytes decoded by PM2_CNT_BLK. Support for the PM2 register block is optional. If supported, this value is >= 1. If not supported, this field contains zero.
PM_TMR_LEN	1	91	Number of bytes decoded by PM_TMR_BLK. If the PM Timer is supported, this field's value must be 4. If not supported, this field contains zero.

GPE0_BLK_LEN	1	92	The length of the register whose address is given by X_GPE0_BLK (if nonzero) or by GPE0_BLK (otherwise) in bytes. The value is a non-negative multiple of 2.
GPE1_BLK_LEN	1	93	The length of the register whose address is given by X_GPE1_BLK (if nonzero) or by GPE1_BLK (otherwise) in bytes. The value is a non-negative multiple of 2.
GPE1_BASE	1	94	Offset within the ACPI general-purpose event model where GPE1 based events start.
CST_CNT	1	95	If non-zero, this field contains the value OSPM writes to the SMI_CMD register to indicate OS support for the _CST object and C States Changed notification.
P_LVL2_LAT	2	96	The worst-case hardware latency, in microseconds, to enter and exit a C2 state. A value > 100 indicates the system does not support a C2 state.
P_LVL3_LAT	2	98	The worst-case hardware latency, in microseconds, to enter and exit a C3 state. A value > 1000 indicates the
FLUSH_SIZE	2	100	If WBINVD=0, the value of this field is the number of flush strides that need to be read (using cacheable addresses) to completely flush dirty lines from any processor's memory caches. Notice that the value in FLUSH_STRIDE is typically the smallest cache line width on any of the processor's caches (for more information, see the FLUSH_STRIDE field definition). If the system does not support a method for flushing the processor's caches, then FLUSH_SIZE and WBINVD are set to zero. Notice that this method of flushing the processor caches has limitations, and WBINVD=1 is the preferred way to flush the processors caches. This value is typically at least 2 times the cache size. The maximum allowed value for FLUSH_SIZE multiplied by FLUSH_STRIDE is 2 MB for a typical maximum supported cache size of 1 MB. Larger cache sizes are supported using WBINVD=1. This value is ignored if WBINVD=1. This field is maintained for ACPI 1.0 processor compatibility on existing systems. Processors in new ACPI-compatible systems are required to support the WBINVD function and indicate this to OSPM by setting the WBINVD field = 1.
FLUSH_STRIDE	2	102	If WBINVD=0, the value of this field is the cache line width, in bytes, of the processor's memory caches. This value is typically the smallest cache line width on any of the processor's caches. For more information, see the description of the FLUSH_SIZE field. This value is ignored if WBINVD=1. This field is maintained for ACPI 1.0 processor compatibility on existing systems. Processors in new ACPI-compatible systems are required to support the WBINVD function and indicate this to OSPM by setting the WBINVD field = 1.

DUTY_OFFSET	1	104	The zero-based index of where the processor's duty cycle setting is within the processor's P_CNT register.
DUTY_WIDTH	1	105	The bit width of the processor's duty cycle setting value in the P_CNT register. Each processor's duty cycle setting allows the software to select a nominal processor frequency below its absolute frequency as defined by: THTL_EN = 1 BF * DC/(2DUTY_WIDTH) Where: BF-Base frequency DC-Duty cycle setting When THTL_EN is 0, the processor runs at its absolute BF. A DUTY_WIDTH value of 0 indicates that processor duty cycle is not supported and the processor continuously runs at its base frequency.
DAY_ALRM	1	106	The RTC CMOS RAM index to the day-of-month alarm value. If this field contains a zero, then the RTC day of the month alarm feature is not supported. If this field has a non-zero value, then this field contains an index into RTC RAM space that OSPM can use to program the day of the month alarm. See Section 4.8.2.4 for a description of how this hardware works.
MON_ALRM	1	107	The RTC CMOS RAM index to the month of year alarm value. If this field contains a zero, then the RTC month of the year alarm feature is not supported. If this field has a non-zero value, then this field contains an index into RTC RAM space that OSPM can use to program the month of the year alarm. If this feature is supported, then the DAY_ALRM feature must be supported also.
CENTURY	1	108	The RTC CMOS RAM index to the century of data value (hundred and thousand year decimals). If this field contains a zero, then the RTC centenary feature is not supported. If this field has a non-zero value, then this field contains an index into RTC RAM space that OSPM can use to program the centenary field.
IAPC_BOOT_ARCH	2	109	IA-PC Boot Architecture Flags. See Section 5.2.9.3 for a description of this field.
Reserved	1	111	Must be 0.
Flags	4	112	Fixed feature flags. See Table 5.10 for a description of this field.
RESET_REG	12	116	The address of the reset register represented in Generic Address Structure format (See Section 4.8.3.6 for a description of the reset mechanism.) Note: Only System I/O space, System Memory space and PCI Configuration space (bus #0) are valid for values for Address_Space_ID. Also, Register_Bit_Width must be 8 and Register_Bit_Offset must be 0.
RESET_VALUE	1	128	Indicates the value to write to the RESET_REG port to reset the system. (See Section 4.8.3.6 for a description of the reset mechanism.)
ARM_BOOT_ARCH	2	129	ARM Boot Architecture Flags. See Table 5.12 for a description of this field.

FADT Minor Version	1	131	5.0 (errata bits 4-7 = 0) Minor Version of this FADT structure, in “Major.Minor” form, where ‘Major’ is the value in the Major Version Field (Byte offset 8 in this table). Bits 0-3 - The low order bits correspond to the minor version of the specification version. For instance, ACPI 6.3 has a major version of 6, and a minor version of 3. Bits 4-7 - The high order bits correspond to the version of the ACPI Specification errata this table complies with. A value of 0 means that it complies with the base version of the current specification. A value of 1 means this is compatible with Errata A, 2 would be compatible with Errata B, and so on.
X_FIRMWARE_CTRL	8	132	Extended physical address of the FACS. If this field contains a nonzero value which can be used by the OSPM, then the FIRMWARE_CTRL field must be ignored by the OSPM. If the HARDWARE_REDUCED_ACPI flag is set, and both this field and the FIRMWARE_CTRL field are zero, there is no FACS available.
X_DSDT	8	140	Extended physical address of the DSDT. If this field contains a nonzero value which can be used by the OSPM, then the DSDT field must be ignored by the OSPM.
X_PM1a_EVT_BLK	12	148	Extended address of the PM1a Event Register Block, represented in Generic Address Structure format. See Section 4.8.3.1 for a hardware description layout of this register block. This is a required field. If this field contains a nonzero value which can be used by the OSPM, then the PM1a_EVT_BLK field must be ignored by the OSPM.
X_PM1b_EVT_BLK	12	160	Extended address of the PM1b Event Register Block, represented in Generic Address Structure format. See Section 4.8.3.1 for a hardware description layout of this register block. This field is optional; if this register block is not supported, this field contains zero. If this field contains a nonzero value which can be used by the OSPM, then the PM1b_EVT_BLK field must be ignored by the OSPM.
X_PM1a_CNT_BLK	12	172	Extended address of the PM1a Control Register Block, represented in Generic Address Structure format. See Section 4.8.3.2 for a hardware description layout of this register block. This is a required field. If this field contains a nonzero value which can be used by the OSPM, then the PM1a_CNT_BLK field must be ignored by the OSPM.

X_PM1b_CNT_BLK	12	184	Extended address of the PM1b Control Register Block, represented in Generic Address Structure format. See Section 4.8.3.2 for a hardware description layout of this register block. This field is optional; if this register block is not supported, this field contains zero. If this field contains a nonzero value which can be used by the OSPM, then the PM1b_CNT_BLK field must be ignored by the OSPM.
X_PM2_CNT_BLK	12	196	Extended address of the PM2 Control Register Block, represented in Generic Address Structure format. See PM2 Control (PM2_CNT) for a hardware description layout of this register block. This field is optional; if this register block is not supported, this field contains zero. If this field contains a nonzero value which can be used by the OSPM, then the PM2_CNT_BLK field must be ignored by the OSPM.
X_PM_TMR_BLK	12	208	Extended address of the Power Management Timer Control Register Block, represented in Generic Address Structure format. See Section 4.8.3.3 for a hardware description layout of this register block. This field is optional; if this register block is not supported, this field contains zero. If this field contains a nonzero value which can be used by the OSPM, then the
X_GPE0_BLK	12	220	Extended address of the General-Purpose Event 0 Register Block, represented in Generic Address Structure format. See Section 4.8.4.1 for more information. This is an optional field; if this register block is not supported, this field contains zero. If this field contains a nonzero value which can be used by the OSPM, then the GPE0_BLK field must be ignored by the OSPM. Note: Only System I/O space and System Memory space are valid for Address_Space_ID values, and the OSPM ignores Register_Bit_Width, Register_Bit_Offset and Access_Size.
X_GPE1_BLK	12	232	Extended address of the General-Purpose Event 1 Register Block, represented in Generic Address Structure format. See Section 4.8.4.1 for more information. This is an optional field; if this register block is not supported, this field contains zero. If this field contains a nonzero value which can be used by the OSPM, then the GPE1_BLK field must be ignored by the OSPM. Note: Only System I/O space and System Memory space are valid for Address_Space_ID values, and the OSPM ignores Register_Bit_Width, Register_Bit_Offset and Access_Size.
SLEEP_CONTROL_REG	12	244	The address of the Sleep register, represented in Generic Address Structure format (see Section 4.8.3.7 for a description of the sleep mechanism). Note: Only System I/O space, System Memory space and PCI Configuration space (bus #0) are valid for values for Address_Space_ID. Also, Register_Bit_Width must be 8 and Register_Bit_Offset must be 0.

SLEEP_STATUS_REG	12	256	The address of the Sleep status register, represented in Generic Address Structure format (see Section 4.8.3.7 for a description of the sleep mechanism). Note: Only System I/O space, System Memory space and PCI Configuration space (bus #0) are valid for values for Address_Space_ID. Also, Register_Bit_Width must be 8 and Register_Bit_Offset must be 0.
Hypervisor Vendor Identity	8	268	64-bit identifier of hypervisor vendor. All bytes in this field are considered part of the vendor identity. These identifiers are defined independently by the vendors themselves, usually following the name of the hypervisor product. Version information should NOT be included in this field - this shall simply denote the vendor's name or identifier. Version information can be communicated through a supplemental vendor-specific hypervisor API. Firmware implementers would place zero bytes into this field, denoting that no hypervisor is present in the actual firmware.

Как уже отмечалось выше, в таблице FADT имеется поле флагов **Flags** размером 32 бита расположенное со смещением 112 байт от её начала. В рамках спецификации это поле называется «**Fixed Feature Flags**», каждый бит в этом поле отвечает на наличие/отсутствует поддержки определенной фичи (функции или возможности). Всего на данный момент задействовано 22 бита под 21 фичу, последняя фича PERSISTENT_CPU_CACHES занимает два бита. В Таблице 2г приведено описание каждого бита (каждой фича) из спецификации ACPI версии 6.6.

Таблица 2г. Назначение битов в поле Flags («Fixed Feature Flags»).

FACP - Flag	Bit Length	Bit Off-set	Description
WBINVD	1	0	Processor properly implements a functional equivalent to the WBINVD IA-32 instruction. If set, signifies that the WBINVD instruction correctly flushes the processor caches, maintains memory coherency, and upon completion of the instruction, all caches for the current processor contain no cached data other than what OSPM references and allows to be cached. If this flag is not set, the ACPI OS is responsible for disabling all ACPI features that need this function. This field is maintained for ACPI 1.0 processor compatibility on existing systems. Processors in new ACPI-compatible systems are required to support this function and indicate this to OSPM by setting this field.
WBINVD_FLUSH	1	1	If set, indicates that the hardware flushes all caches on the WBINVD instruction and maintains memory coherency, but does not guarantee the caches are invalidated. This provides the complete semantics of the WBINVD instruction, and provides enough to support the system sleeping states. If neither of the WBINVD flags is set, the system will require FLUSH_SIZE and FLUSH_STRIDE to support sleeping states. If the FLUSH parameters are also not supported, the machine cannot support sleeping states S1, S2, or S3.
PROC_C1	1	2	A one indicates that the C1 power state is sup-
P_LVL2_UP	1	3	A zero indicates that the C2 power state is configured to only work on a uniprocessor (UP) system. A one indicates that the C2 power state is configured to work on a UP or multiprocessor (MP) system.
PWR_BUTTON	1	4	A zero indicates the power button is handled as a fixed feature programming model; a one indicates the power button is handled as a control method device. If the system does not have a power button, this value would be “1” and no power button device would be present. Independent of the value of this field, the presence of a power button device in the namespace indicates to OSPM that the power button is handled as a control method device.

SLP_BUTTON	1	5	A zero indicates the sleep button is handled as a fixed feature programming model; a one indicates the sleep button is handled as a control method device. If the system does not have a sleep button, this value would be “1” and no sleep button device would be present. Independent of the value of this field, the presence of a sleep button device in the namespace indicates to OSPM that the sleep button is handled as a control method device.
FIX_RTC	1	6	A zero indicates the RTC wake status is supported in fixed register space; a one indicates the RTC wake status is not supported in fixed register space.
RTC_S4	1	7	Indicates whether the RTC alarm function can wake the system from the S4 state. The RTC must be able to wake the system from an S1, S2, or S3 sleep state. The RTC alarm can optionally support waking the system from the S4 state, as indicated by this value.
TMR_VAL_EXT	1	8	A zero indicates TMR_VAL is implemented as a 24-bit value. A one indicates TMR_VAL is implemented as a 32-bit value. The TMR_STS bit is set when the most significant bit of the TMR_VAL toggles.
DCK_CAP	1	9	A zero indicates that the system cannot support docking. A one indicates that the system can support docking. Notice that this flag does not indicate whether or not a docking station is currently present; it only indicates that the system is capable of docking.
RESET_REG_SUP	1	10	If set, indicates the system supports system reset via the FADT RESET_REG as described in Section 4.8.3.6.
SEALED_CASE	1	11	System Type Attribute. If set indicates that the system has no internal expansion capabilities and the case is sealed.
HEADLESS	1	12	System Type Attribute. If set indicates the system cannot detect the monitor or keyboard / mouse devices.
CPU_SW_SLP	1	13	If set, indicates to OSPM that a processor native instruction must be executed after writing the SLP_TYPx register.
PCI_EXP_WAK	1	14	If set, indicates the platform supports the PCIEXP_WAKE_STS bit in the PM1 Status register and the PCIEXP_WAKE_EN bit in the PM1 Enable register. This bit must be set on platforms containing chipsets that implement PCI Express and supports PM1 PCIEXP_WAK bits.

USE_PLATFORM_CLOCK	1	15	A value of one indicates that OSPM should use a platform provided timer to drive any monotonically non-decreasing counters, such as OSPM performance counter services. Which particular platform timer will be used is OSPM specific, however, it is recommended that the timer used is based on the following algorithm: If the HPET is exposed to OSPM, OSPM should use the HPET. Otherwise, OSPM will use the ACPI power management timer. A value of one indicates that the platform is known to have a correctly implemented ACPI power management timer. A platform may choose to set this flag if a internal processor clock (or clocks in a multi-processor configuration) cannot provide consistent monotonically non-decreasing counters. Note: If a value of zero is present, OSPM may arbitrarily choose to use an internal processor clock or a platform timer clock for these operations. That is, a zero does not imply that OSPM will necessarily use the internal processor clock to generate a monotonically non-decreasing counter to the system.
S4_RTC_STS_VALID	1	16	A one indicates that the contents of the RTC_STS flag is valid when waking the system from S4. See Table 4.11 for more information. Some existing systems do not reliably set this input today, and this bit allows OSPM to differentiate correctly functioning platforms from platforms with this errata.
REMOTE_POWER_ON_CAPABLE	1	17	A one indicates that the platform is compatible with remote power- on. That is, the platform supports OSPM leaving GPE wake events armed prior to an S5 transition. Some existing platforms do not reliably transition to S5 with wake events enabled (for example, the platform may immediately generate a spurious wake event after completing the S5 transition). This flag allows OSPM to differentiate correctly functioning platforms from platforms with this type of errata.
FORCE_APIC_CLUSTER_MODEL	1	18	A one indicates that all local APICs must be configured for the cluster destination model when delivering interrupts in logical mode. If this bit is set, then logical mode interrupt delivery operation may be undefined until OSPM has moved all local APICs to the cluster model. This bit is intended for xAPIC based machines that require the cluster destination model even when 8 or fewer local APICs are present in the machine.

FORCE_APIC_PHYSICAL_-DESTINATION_MODE	1	19	A one indicates that all local xAPICs must be configured for physical destination mode. If this bit is set, interrupt delivery operation in logical destination mode is undefined. On machines that contain fewer than 8 local xAPICs or that do not use the xAPIC architecture, this bit is ignored.
HW_REDUCED_ACPI *	1	20	A one indicates that the Hardware-Reduced ACPI (section 4.1) is implemented, therefore software-only alternatives are used for supported fixed-features defined in chapter 4.
LOW_POWER_S0_IDLE_CAPABLE	1	21	A one informs OSPM that the platform is able to achieve power savings in S0 similar to or better than those typically achieved in S3. In effect, when this bit is set it indicates that the system will achieve no power benefit by making a sleep transition to S3.
PERSISTENT_CPU_CACHES	2	22	The following values describe whether cpu caches and any other caches that are coherent with them, are considered by the platform to be persistent. The platform evaluates the configuration present at system startup to determine this value. System configuration changes after system startup may invalidate this. 00b - Not reported by the platform. Software should reference the NFIT Platform Capabilities 01b - Cpu caches and any other caches that are coherent with them, are not persistent. Software is responsible for flushing data from cpu caches to make stores persistent. Supersedes NFIT Platform Capabilities. 10b - Cpu caches and any other caches that are coherent with them, are persistent. Supersedes NFIT Platform Capabilities. When reporting this state, the platform shall provide enough stored energy for ALL of the following: - Time to flush cpu caches and any other caches that are coherent with them - Time of all targets of those flushes to complete flushing stored data - If supporting hot plug, the worst case CXL device topology that can be hot plugged 11b - Reserved
Reserved	8	24	

В приведенной выше Таблице 2г присутствуют ряд битовых полей которые могут быть нам интересны при исследованием проблемы неработающей кнопки «Power». В частности интерес представляют следующие биты:

- **Бит 4 - PWR_BUTTON**. Если данный бит установлен в «0», то для обработки кнопки используется модель «Fixed Feature», то есть изменение её состояния порождает аппаратное прерывание SMI, значение статуса передается в регистровой группе PM1A_STS/PM1B_STS в бите PWRBTN_STS (бит номер 8), при этом учитывается значение разрешающего бита PWRBTN_EN находящегося в регистровой группе PM1A_EN/PM1B_EN. Обработка события (прерывания) производится внутри OSPM. Если же данный бит установлен в «1», то изменение состояния кнопки «Power» обслуживается как «Control Method» событие, то есть вызывается аппаратное прерывание SCI, в регистр GPE заносится код события, а обработка осуществляется в AML коде.
- **Бит 20 — HW_REDUCED_ACPI**. Если данный бит установлен в «1», то вычислительная система не имеет аппаратной поддержки ACPI, то есть является «Hardware-Reduced ACPI», все события формируются и обрабатываются программно, с минимальной поддержкой от аппаратуры.

Чтобы не быть голословным, приведу Листинг 2 содержащий полностью декомпиллированную таблицу FADT снятую с ноутбука моего коллеги «**HP Laptop Model 15s-eq1337ur (ProductID 461U9EA#ACB)»**. В данном листинге слева приведены три колонки цифр, их назначение следующее: первая колонка содержит смещение в байтах в шестнадцатеричном формате, вторая колонка — смещение в десятичном формате, третья — размер поля в байтах в шестнадцатеричном формате. О том, как получить все таблицы ACPI «Firmware» и декомпилировать их я расскажу в последующих главах.

Содержимое данной таблицы может представлять интерес не только разработчиков операционных систем, но и для обычного пользователя. Так, из этой таблицы я узнал, что мой рабочий ноутбук «Lenovo IdeaPad» не поддерживает режим энергосбережения «Low Power S0» - режим быстрого «засыпания/пробуждение» (без полной остановки процессора). К сожалению, в современных операционных системах, как правило, нет стандартных утилит для просмотра содержимого FADT таблицы, а зря.

Листинг 2. Декомпилированная таблица FADT с ноутбука «HP Laptop Model 15s-eq1337ur».

```
/*
 * Intel ACPI Component Architecture
 * AML/ASL+ Disassembler version 20260408 (64-bit version)
 * Copyright (c) 2000 - 2026 Intel Corporation
 *
 * Disassembly of facp.dat
 *
 * ACPI Data Table [FACP]
 *
 * Format: [HexOffset DecimalOffset ByteLength]  FieldName : FieldValue (in hex)
 */

[000h 0000 004h]                        Signature : "FACP"     [Fixed ACPI Description Table
(FADT)]
[004h 0004 004h]                Table Length : 00000114
[008h 0008 001h]                    Revision : 06
[009h 0009 001h]                    Checksum : FF
[00Ah 0010 006h]                      Oem ID : "HPQOEM"
[010h 0016 008h]                Oem Table ID : "SLIC-MPC"
[018h 0024 004h]                Oem Revision : 0107200a
```

```
[01Ch 0028 004h]             Asl Compiler ID : "HP  "
[020h 0032 004h]       Asl Compiler Revision : 00010013

[024h 0036 004h]                FACS Address : C74FE000
[028h 0040 004h]                DSDT Address : C7241000
[02Ch 0044 001h]                       Model : 00
[02Dh 0045 001h]                  PM Profile : 02 [Mobile]
[02Eh 0046 002h]                SCI Interrupt : 0009
[030h 0048 004h]            SMI Command Port : 000000B2
[034h 0052 001h]           ACPI Enable Value : A0
[035h 0053 001h]          ACPI Disable Value : A1
[036h 0054 001h]              S4BIOS Command : 00
[037h 0055 001h]              P-State Control : 00
[038h 0056 004h]    PM1A Event Block Address : 00000800
[03Ch 0060 004h]    PM1B Event Block Address : 00000000
[040h 0064 004h]  PM1A Control Block Address : 00000804
[044h 0068 004h]  PM1B Control Block Address : 00000000
[048h 0072 004h]   PM2 Control Block Address : 000000B4
[04Ch 0076 004h]      PM Timer Block Address : 00000808
[050h 0080 004h]          GPE0 Block Address : 00000820
[054h 0084 004h]          GPE1 Block Address : 00000000
[058h 0088 001h]      PM1 Event Block Length : 04
[059h 0089 001h]    PM1 Control Block Length : 02
[05Ah 0090 001h]    PM2 Control Block Length : 01
[05Bh 0091 001h]       PM Timer Block Length : 04
[05Ch 0092 001h]           GPE0 Block Length : 08
[05Dh 0093 001h]           GPE1 Block Length : 00
[05Eh 0094 001h]            GPE1 Base Offset : 00
[05Fh 0095 001h]                _CST Support : 00
[060h 0096 002h]                  C2 Latency : 0065
[062h 0098 002h]                  C3 Latency : 03E9
[064h 0100 002h]              CPU Cache Size : 0400
[066h 0102 002h]          Cache Flush Stride : 0010
[068h 0104 001h]           Duty Cycle Offset : 01
[069h 0105 001h]            Duty Cycle Width : 03
[06Ah 0106 001h]         RTC Day Alarm Index : 0D
[06Bh 0107 001h]       RTC Month Alarm Index : 00
[06Ch 0108 001h]           RTC Century Index : 32
[06Dh 0109 002h]  Boot Flags (decoded below) : 0010
               Legacy Devices Supported (V2) : 0
             8042 Present on ports 60/64 (V2) : 0
                         VGA Not Present (V4) : 0
                       MSI Not Supported (V4) : 0
                 PCIe ASPM Not Supported (V4) : 1
                    CMOS RTC Not Present (V5) : 0
[06Fh 0111 001h]                     Reserved : 00
[070h 0112 004h]       Flags (decoded below) : 0020C5BD
      WBINVD instruction is operational (V1) : 1
              WBINVD flushes all caches (V1) : 0
                    All CPUs support C1 (V1) : 1
                  C2 works on MP system (V1) : 1
             Control Method Power Button (V1) : 0
             Control Method Sleep Button (V1) : 1
        RTC wake not in fixed reg space (V1) : 0
             RTC can wake system from S4 (V1) : 1
                         32-bit PM Timer (V1) : 1
                    Docking Supported (V1) : 0
               Reset Register Supported (V2) : 1
                           Sealed Case (V3) : 0
                  Headless - No Video (V3) : 0
        Use native instr after SLP_TYPx (V3) : 0
              PCIEXP_WAK Bits Supported (V4) : 1
                     Use Platform Timer (V4) : 1
               RTC_STS valid on S4 wake (V4) : 0
                Remote Power-on capable (V4) : 0
                 Use APIC Cluster Model (V4) : 0
     Use APIC Physical Destination Mode (V4) : 0
                      Hardware Reduced (V5) : 0
                    Low Power S0 Idle (V5) : 1

[074h 0116 00Ch]              Reset Register : [Generic Address Structure]
[074h 0116 001h]                    Space ID : 01 [SystemIO]
[075h 0117 001h]                   Bit Width : 08
[076h 0118 001h]                  Bit Offset : 00
[077h 0119 001h]        Encoded Access Width : 00 [Undefined/Legacy]
[078h 0120 008h]                     Address : 00000000000000B2
```

```
[080h 0128 001h]        Value to cause reset : EB
[081h 0129 002h]   ARM Flags (decoded below) : 0000
                             PSCI Compliant : 0
                         Must use HVC for PSCI : 0

[083h 0131 001h]         FADT Minor Revision : 02
[084h 0132 008h]                FACS Address : 0000000000000000
[08Ch 0140 008h]                DSDT Address : 00000000C7241000
[094h 0148 00Ch]           PM1A Event Block : [Generic Address Structure]
[094h 0148 001h]                   Space ID : 01 [SystemIO]
[095h 0149 001h]                  Bit Width : 20
[096h 0150 001h]                 Bit Offset : 00
[097h 0151 001h]        Encoded Access Width : 02 [Word Access:16]
[098h 0152 008h]                    Address : 0000000000000800

[0A0h 0160 00Ch]           PM1B Event Block : [Generic Address Structure]
[0A0h 0160 001h]                   Space ID : 01 [SystemIO]
[0A1h 0161 001h]                  Bit Width : 00
[0A2h 0162 001h]                 Bit Offset : 00
[0A3h 0163 001h]        Encoded Access Width : 02 [Word Access:16]
[0A4h 0164 008h]                    Address : 0000000000000000

[0ACh 0172 00Ch]          PM1A Control Block : [Generic Address Structure]
[0ACh 0172 001h]                   Space ID : 01 [SystemIO]
[0ADh 0173 001h]                  Bit Width : 10
[0AEh 0174 001h]                 Bit Offset : 00
[0AFh 0175 001h]        Encoded Access Width : 02 [Word Access:16]
[0B0h 0176 008h]                    Address : 0000000000000804

[0B8h 0184 00Ch]          PM1B Control Block : [Generic Address Structure]
[0B8h 0184 001h]                   Space ID : 01 [SystemIO]
[0B9h 0185 001h]                  Bit Width : 00
[0BAh 0186 001h]                 Bit Offset : 00
[0BBh 0187 001h]        Encoded Access Width : 02 [Word Access:16]
[0BCh 0188 008h]                    Address : 0000000000000000

[0C4h 0196 00Ch]           PM2 Control Block : [Generic Address Structure]
[0C4h 0196 001h]                   Space ID : 01 [SystemIO]
[0C5h 0197 001h]                  Bit Width : 08
[0C6h 0198 001h]                 Bit Offset : 00
[0C7h 0199 001h]        Encoded Access Width : 01 [Byte Access:8]
[0C8h 0200 008h]                    Address : 00000000000000B4

[0D0h 0208 00Ch]              PM Timer Block : [Generic Address Structure]
[0D0h 0208 001h]                   Space ID : 01 [SystemIO]
[0D1h 0209 001h]                  Bit Width : 20
[0D2h 0210 001h]                 Bit Offset : 00
[0D3h 0211 001h]        Encoded Access Width : 03 [DWord Access:32]
[0D4h 0212 008h]                    Address : 0000000000000808

[0DCh 0220 00Ch]                 GPE0 Block : [Generic Address Structure]
[0DCh 0220 001h]                   Space ID : 01 [SystemIO]
[0DDh 0221 001h]                  Bit Width : 40
[0DEh 0222 001h]                 Bit Offset : 00
[0DFh 0223 001h]        Encoded Access Width : 01 [Byte Access:8]
[0E0h 0224 008h]                    Address : 0000000000000820

[0E8h 0232 00Ch]                 GPE1 Block : [Generic Address Structure]
[0E8h 0232 001h]                   Space ID : 01 [SystemIO]
[0E9h 0233 001h]                  Bit Width : 00
[0EAh 0234 001h]                 Bit Offset : 00
[0EBh 0235 001h]        Encoded Access Width : 01 [Byte Access:8]
[0ECh 0236 008h]                    Address : 0000000000000000


[0F4h 0244 00Ch]      Sleep Control Register : [Generic Address Structure]
[0F4h 0244 001h]                   Space ID : 00 [SystemMemory]
[0F5h 0245 001h]                  Bit Width : 00
[0F6h 0246 001h]                 Bit Offset : 00
[0F7h 0247 001h]        Encoded Access Width : 00 [Undefined/Legacy]
[0F8h 0248 008h]                    Address : 0000000000000000

[100h 0256 00Ch]       Sleep Status Register : [Generic Address Structure]
[100h 0256 001h]                   Space ID : 00 [SystemMemory]
[101h 0257 001h]                  Bit Width : 00
[102h 0258 001h]                 Bit Offset : 00
[103h 0259 001h]        Encoded Access Width : 00 [Undefined/Legacy]
```

```
[104h 0260 008h]                    Address : 0000000000000000

[10Ch 0268 008h]               Hypervisor ID : 0000000000000000

Raw Table Data: Length 276 (0x114)
```

1.6.2. Таблица «Firmware ACPI Control Structure» (FACS)

Таблица **FACS** начинается со строки "FACS" в поле сигнатуры. Эта таблица размещается в read-write области памяти доступной (разделяемой) и для подсистемы ACPI и для основной операционной системы. FACS таблица содержит два важный поля:

Global Lock (смещение 14, размер 4 байта) — содержит область памяти для глобальной блокировки (mutex). Этот «мютекс» используется для разграничения доступа к регистрам и чувствительной аппаратуре ACPI между основной операционной системой и подсистемой ACPI. Иными словами, если AML код или ОС желают произвести доступ (запись) в какой-то из регистров, то они предварительно «захватывают» Global Lock, а закончив работу — «высвобождают» его. Из 32-х байт в поле Global Lock используются только два младших бита:

- Бит 0 — **Pending**, если установлен в «1», то показывает, что уже действует активный запрос на «владение» блокировкой.
- Бит 1 — **Owned**, если установлен в «1», то показывает, что кто-то уже занял (заблокировал) доступ.

Захват блокировки производится по следующей схеме:

1. Считать значение словам Global Lock и сбросить Pending бит.
2. Проверить, установлен ли бит Owned. Если установлен, то установить бит Pending.
3. Установить бит Owned.
4. Попытаться записать новое значение слова в Global Lock и проверить какое значение записалось, используя атомарные инструкции (например, **lock cmpxchg**).
5. Если НЕ записалось, то повторит попытку с п.1
6. Иначе проверить, какое значение записалось: если **0b11**, то захват блокировки выполнен успешно, иначе — находится в состоянии ожидания (Pending).
7. В случае ожидания (установился бит Pending), требуется дождаться прерывания или GPE события от Global Lock и попробовать выполнить захват блокировки снова.

Схема высвобождения блокировки Global Lock выглядит следующим образом:

1. Считать текущее значение Global Lock.
2. Сбросить биты Owned и Pending.
3. Попытаться записать новое значение в Global Lock используя атомарные инструкции.
4. Если записать не удалось, то выполнить попытку с п.1
5. Если записанное значение содержит **0b00**, то высвобождение прошло успешно.

6. Если записанное значение содержит установленный бит Pending, то вызывающая процедура обязана самостоятельно установить бит GBL_RLS в группе регистров PM1A_CNT/PM1B_CNT.

X Firmware Waking Vector (смещение 24, размерность 8 байт) — содержит указатель (вектор) на функцию пробуждения основной операционной системы. При «засыпании» OSPM заносит в это поле адрес функции которая будет вызвана сразу после пробуждения.

1.6.3. Таблица «Differentiated System Description Table» (DSDT)

Таблица «Differentiated System Description Table» (DSDT) в поле сигнатуры содержит строку символов "DSDT". Данная таблица, а также множество таблиц «Secondary System Description Table» (SSDT), после стандартного заголовка содержат байт-код «Firmware» на языке AML, предназначенный для исполнения на виртуальной машине. Напомню, что указатель на таблицу DSDT содержится в структуре FADT, а массив указателей на таблицы SSDT находится в корневой структуре RSDT/XSDT.

Обе структуры DSDT и SSDT имеют одинаковый формат (см. таблицу 2д) — после стандартного заголовка, ровно с 36-го байта следует N байтов AML байт-кода, где N — размер таблицы (поле Length) за вычетом размера стандартного заголовка (минус 36 байт). Операционная система загружает данные таблицы в следующей последовательности: сначала загружается и инициализируется код из таблице DSDT, после чего последовательно загружается и инициализируется код из всех таблиц SSDT в той последовательности, в которой они приведены в массиве в конце структуры RSDT/XSDT.

Таблица 2д. Формат структуры таблиц DSDT и SSDT.

Field	Byte Length	Byte Offset	Description
Length	4	4	Length, in bytes, of the entire DSDT (including the header).
Revision	1	8	2. This field also sets the global integer width for the AML interpreter. Values less than two will cause the interpreter to use 32-bit integers and math. Values of two and greater will cause the interpreter to use full 64-bit integers and math.
Checksum	1	9	Entire table must sum to zero.
OEMID	6	10	OEM ID
OEM Table ID	8	16	The manufacture model ID.
OEM Revision	4	24	OEM revision of DSDT for supplied OEM Table ID.
Creator ID	4	28	Vendor ID for the ASL Compiler.
Creator Revision	4	32	Revision number of the ASL Compiler.
Definition Block	n	36	n bytes of AML code (see Section 5.4).

Под загрузкой понимается размещение AML кода в памяти, создание описанных в таблице пространств имен (ACPI Namespace) и вызов AML оператора **Load()** или **LoadTable()**. Согласно спецификации ACPI, данные из последующих таблиц могут лишь добавлять новые данные структуры данных и пространства имён, но не могут изменят или удалять уже загруженные. Предполагается что OEM производитель предоставляет базовую поддержку платформы в основной таблице DSDT, а потом с помощью одной или нескольких дополнительных таблиц SSDT расширяет базовый функционал в зависимости от

модификации (модели) изделия. У такой схемы есть одно важное следствие — код в таблицах SSDT может полагаться только на данные (и другой код) находящийся в таблице DSDT, он не может полагаться на данные и код в других таблицах SSDT.

Содержимое данных таблиц представляют особый интерес, так как всё самое важное происходит именно в программной (байт-код) части ACPI. Поэтому, далее мы немного рассмотрим что же из себя представляет язык AML и его Си-подобное представление — ASL.

1.7. Язык «ACPI Machine Language» и его Си-подобное представление (ASL)

Исполняемый виртуальной машиной байт-код представляет собой последовательность из одно-байтовых OP-кодов с переменным числом параметров (операндов). Параметрами могут выступать различные типы и структуры данных, либо результат выполнения других OP-кодов. Среди OP-кодов есть как чисто арифметические операции, так и операции ветвления, определения (объявления) переменных или методов, а также операции взаимодействия с внешним миром. Все данные и код хранятся в глобальной области памяти представляемой в виде иерархической древовидной структуры называемой «Пространство имён» (Namespace). У методов, поименованных кусков исполняемого кода, могут быть объявлены входные параметры (аргументы), а также локальные и динамические переменные. Метод может возвращать либо один параметр, либо структуру данных называемую «пакетом» (Package).

1.7.1. Типы данных в AML/ASL

Виртуальная машина AML является нестрого типизированной, она оперирует небольшим набором типов данных, которые местами можно преобразовывать «на ходу», хотя для многих типов имеются соответствующие операции преобразования. Пройдемся по основным типам:

- Integer — целое число разрядносться 32 или 64 бита.
- String — строка символов ASCII (NULL-терминированная).
- Buffer — произвольный массив байтов.
- Package — упорядоченная коллекция, массив, элементов. Каждый элемент массива тоже может быть типом Packages, что делает структуру древовидной.
- Reference — ссылка (указатель) на поименованный объект.
- Field/Region — структуры в области памяти, обычно используются для доступа к аппаратным регистрам.
- Method — метод, исполняемый кусок кода.
- Mutex — семафор для разграничения доступа.

Замечание по размеру типа Integer. Если номер ревизии (поле Revision в стандартном заголовке) равен или более 2, то целочисленный тип Integer имеет размер 64 бит, иначе — 32 бита. Как правило, поле Revision заполняется компилятором автоматически. Однако, разные таблицы могут использовать разные размер типа Integer, поэтому разработчик «firmware» должен уделять внимание тому какого размера параметры он передает при вызове методов между различными таблицами.

1.7.2. Конструкции языка AML/ASL (операторы)

При программировании используется Си-подобное представления кода, этот кода компилируется в байт-код с помощью утилиты компилятора/декомпилятора **iasl**. Текстовое представления программы для AML машины называется ASL. Ниже приведены некоторые конструкции выраженные на языке ASL.

Переменные и структуры данных:

- Восемь локальных переменных **Local0 - Local7** обнуляемых при входе в исполнение метода.
- Восемь аргументов **Arg0 - Arg7** передаваемых в качестве параметров при вызове метода.
- Динамические четырех-буквенные переменные в пространстве имен объявляемые оператором **Name (NameString, TermArg)**.
- **OperationRegion (NameString, Space, Offset, Length)** — объявляет структуру типа «операционная область». Делает доступной аппаратную область памяти или I/O для методов. Как правило в этих областях находятся регистры управления той или иной аппаратурой.
- **Field (OpRegionName, AccessType, …)** — определяет поля в операционной области, фактически - поименованные аппаратные регистры.
- **Buffer (Length) { data }** — объявляет линейный массив данных.
- **Package (Elements)** — объявляет пакет с данными (древовидная структура данных).

Все динамически переменные, если они созданы внутри метода, разрушаются после его исполнение. Динамические переменные сознанные внутри **Scope()** остаются до разрушение этой области видимости. Синтаксис объявления структуры «операционная область» и её полей приведены в листинга 3а и 3б.

Листинг 3а. Синтаксис объявления структуры типа «операционная область».

```
OperationRegion (
    RegionName, // 4-х символьное обозначение операционной области.
    RegionSpace, // 4-х символьное обозначение типа адресного пространства (Memory, I/O, и
т. д.).
    Offset, // Фактически задает физический адрес в памяти или номер порта I/O.
    Length  // Размер области в байтах или портах I/O.
)

// Пример:
OperationRegion (POWR, IPMI, 0x3000, 0x100) // Power network function
```

В примере из листинга 3а строка **"IPMI"** является зарезервированным словом указывающим на то, что объявляемая область будет размещена в пространстве адресов ввода/вывода центрального процессора, доступное на архитектуре x86 через инструкции процессора IN/OUT. 0x3000 — это номер базового (первого) порта I/O, а 0x100 — отведенное под область число портов.

Листинг 3б. Описания полей структуры типа «операционная область».

```
Field(
    RegionName, // Имя операционной области
    AccessType, // Тип доступа (всегда BufferAcc)
    LockRule, // Способ блокировки доступа – указывает требуется ли использовать Global Lock.
    UpdateRule // Правила записи (не используется)
) {FieldUnitList} // в фигурных скобках идет перечисление полей.

// Пример:
Field(POWR, BufferAcc, NoLock, Preserve)
{
    Offset(0xC1), // Задать смещение от начала области - 0xC1
    SPWL, 8, // Set power limit [command value 0xC1] – первое поле
    GPWL, 8, // Get power limit [command value 0xC2] – второе поле
    Offset(0xC8), // Задать смещение от начала области - 0xC8
    GPMM, 8 // Get power meter measurement [command value 0xC8] – третье поле
}
```

В примере данном на листинге 3б объявляются три поименованных поля: POWR.SPWL, POWR.GPWL и POWR.GPMM доступ к которым организуется через порты ввода/вывода с номерами 0x30C1, 0x30C2 и 0x30C8. Можно рассматривать это как поименованные аппаратные регистры.

Операции управления потоком:

- **If (Predicate) { … } Else { … } —** условное исполнение кода;
- **While (Predicate) { … } —** цикл типа «while»;
- **Return (Value)** — возвращает управление из вызванного метода с возвратом значения Value;
- **Break** — выход из ближайшего цикла;

Методы и пространство имён:

- **Method (NameString, NumArgs, SerializeFlag) { … } —** объявляет новый метод с четырех-символьным именем NameString, принимающем на вдох NumArgs число параметров. SerializeFlag — указывает на то, можно ли исполнять метод параллельно с другими или только последовательно (доступные варианты: NotSeriallized, Serialized).
- **External (NameString, ObjectType)** — объявить внешний объект (предположительно находящийся и загружаемый из другой таблицы) на который будет ссылаться код из данной таблицы.
- **Scope (NameString) { … } —** устанавливает текущую область видимости в иерархии пространства имен.
- **Device (NameString) { … } —** объявляет объект типа **Device** с четырех-буквенным именем NameString.

Работа с данными и арифметикой:

- **Store (Y, X)** — сохранить значение, эквивалентно X = Y.
- **CopyObject (Source)** — сделать копию объекта.

- **Add(X, Y, X)**, **Subtract(X, Y, X)**, **Multiply(X, Y, X)**, **Divide(X, Y, X)** — соответствующие арифметические операции: X op Y → X.
- **Mod (X, Y, X)** — вычисление остатка от деления X mod Y → X.
- **Shl**, **Shr**, **And**, **Or**, **Xor**, **Not** — битовые операции.
- **Concat (X, Y, X)** — конкатенация буферов или строк.
- **ToBuffer()**, **ToDecimalString()**, **ToHexString()**, **ToInteger()** — преобразование типов.
- **SizeOf (Object)** — получение размера объекта в байта (или элементах, для массива).

Предикатная логика:

- **CondRefOf (Name)** — принимает значение **true** если объект **Name** существует/объявлен.
- **Match (Value, PackageTemplate)** — сравнение по шаблону.
- **LNot**, **LAnd**, **LOr** — логические операции используемые в предикатах (в условиях).

Доступ к объектам:

- **Load (NameString)** — загрузить объект из внешнего источника (зависит от реализации в ОС).
- **RefOf (NameString)** — возвращает ссылку на поименованный объект (получение указателя).
- **DerefOf (Reference)** — превращает ссылку/указатель в объект (разыменование).
- **Index (X, Y, Z)** — взятие Y-го элемента в массиве X, эквивалентно Z ← X[Y].

Взаимодействие с внешним миром:

- **Notify (NameString|Object, Value)** — послать событие поименованному объекту в ACPI системе (в OSPM).
- **Mutex (NameString, SyncLevel)** — объявление мьютекса.
- **Acquire (SyncObject, Timeout)**, **Release(SyncObject)** — работа с мютексами: заём и высвобождение.
- **LoadTable() —** загружает ACPI таблицу.
- **Debug (FormatString, Args)** — выводит отладочное сообщение.
- **Fatal (Type, Code, Arg)** — аварийный останов машины с кодом.
- **Sleep (TimeMs)**, **Stall (TimeUsec)** — задержки исполнения.

Как было отмечено выше, AML допускает динамическое преобразование типов. В примере на Листинге 3в в локальную переменную **Local1** заносится строка символов, которая далее автоматически преобразовывается в тип Integer при исполнении операции **Add**. Результат исполнения операции сложения является тип Integer, он автоматически преобразовывается в тип Buffer и сохраняется в динамической переменной **BUF1** как элемент массива.

Листинг 3в. Преобразование типов данных в процессе исполнения.

```
Store ("5678", Local1)
Add (0x1234, Local1, BUF1)
```

1.7.3. Оператор Notify() для отправки нотификаций в OSPM

Отдельно следует рассмотреть оператор **Notify()**. В процессе выполнения байт-код программы у неё может возникнут необходимость что-то сообщить системе, то есть передать оповещение/нотификацию в операционную систему. Обычно источниками таких нотификаций могут быть различные устройства: их подключение, отключение, изменение состояния; фиксация ошибки вызывает поток событий о которых следует сообщить системе. Чтобы отправить нотификацию, в языке байт-код машины предусмотрен оператор **Notify()**, он принимает два параметра: **Arg0** — строка-путь в древовидной иерархии к устройству-источнику события, и **Arg1** — одно-байтовый информационный код события. Если требуется передать большее количество информации, то у соответствующего объекта типа «Device» предусматривается какой нибудь метод (например, **_DSM**) позволяющий вынуть сложную структуру с данными. Этот метод вызовет операционная система в процессе обработки нотификации.

Значения параметра **Arg1**:

- 0x00 - 0x7F — Одинаковые для всех объектов типа «Device»:
  - 0 — Сообщает OSPM о необходимости выполнить пересканирование шины и перенумерацию устройств в дереве начиная с текущей позиции.
  - 1 — Информирует OSPM о том, что было подключено (или отключено) новое устройства и требуется выполнить процедуру перенумерации устройств с текущей позиции.
  - 2 — Сигнализирует в OSPM о том, что устройство сигнализирует о необходимости пробуждения (WAVE event) и значить OSPM должна вызвать соответствующий метод драйвера этого устройства.
  - 3 — Необходимо выполнить процедуру «Eject» для данного устройства, например для CD-ROM или Floppy. OSPM должна вызвать соответствующий метод драйвера.
  - 4 — Облегченный вариант подключения/отключения устройства. При отключении устройства перенумерация не выполняется, устройство просто помечается как «нерабочее».
  - 5 — Информирует OSPM о том, что подключенное устройство не способно работать на текущей частоте шины (например для устаревших 33 МГц PCI устройств).
  - 6 — Информирует OSPM о том, что подключенное устройство не может функционировать в текущем слоте, порту или бухте по причине несовместимости параметров.
  - 7 — Информирует OSPM о том, что устройство не может быть успешно выведено из состояния энергосбережения D3 по причине сбоя системы питания.
  - 8 — Сообщает OSPM о том, что у данного устройства изменились свойства (набор «capabilities») и требуется выполнить запрос метода **_OSC**.
  - 9 — Информирует о том, что у данного устройства изменилась точка подключения и необходимо вызвать метод **_PLD**.
  - 0x0A — зарезервировано.

    - 0x0B — Произошло обновления информации в блоке «System Locality Information», требуется вызвать метод **_SLI**. Динамическая реконфигурация системы.
    - 0x0C — зарезервировано.
    - 0x0D — Произошло обновление информации в блоке «System Resource Affinity», требуется вызвать метод _PXM. Динамическая миграция устройств.
    - 0x0E — Произошло обновление информации в блоке «Heterogeneous Memory Attributes», требуется вызвать метод _HMA. Динамическое изменение характеристик устройств (скорость, задержка, атрибуты кэша).
    - 0x0F — Сообщает OSPM о том, что произошло асинхронное отключение устройства по причине выявленной ошибки (например, сбой четности памяти или иных контрольных сумм у устройства).
    - 0x10 — 0x7F — зарезервировано.
- 0x80 - 0xBF — Зависит от конкретного устройства.
- 0xC0 - 0xFF — Отведено для нужд OEM производителя.

Ниже в Листинге 3в приведен пример метода **XL19** вызов которого отправляет сразу несколько нотификаций с кодом 0x02 - «Device Wake» в OSPM для различных устройств.

Листинг 3в. Пример отправки нотификаций в OSPM.

```
Method (XL19, 0, NotSerialized)
        {
            Notify (\_SB.PCI0.GP17.XHC0, 0x02) // Device Wake
            Notify (\_SB.PCI0.GP17.XHC1, 0x02) // Device Wake
            Notify (\_SB.PWRB, 0x02) // Device Wake
        }
```

В спецификации ACPI для устройств **_SB.PWRB** (кнопка «Power»), **_SB.SLPB** (кнопка «Sleep»), **_SB._LID** (крышка ноутбука) и **_SB.ACDC** (источник питания постоянного тока) определен только один код нотификации — **0x80** - «Device Status Changed», он информирует OSPM о смене состояние этого устройства: нажата кнопка, закрыта/открыта крышка ноутбука, подключен/отключен источник питания. Запомним эту информацию, возможно она пригодится нам при исследовании байт-кода.

1.7.4. Компиляция и декомпиляция ASL/AML

Полноценная программа на языке ASL должна содержать небольшой заголовок с описанием идентификаторов OEM и указанием имени таблицы в которой она должна быть размещена. Ниже в Листинге 4а приведен пример простой демонстрационной программы для сложения двух переменных в представлении ASL. В заголовке данной программы приводится строка "Sample" в качестве её идентификатора и идентификатора OEM, а так же указывается, что AML код программы должен быть размещен в таблице DSDT.

Листинг 4а. Пример программы для сложения двух переменных на языке ASL.

```
DefinitionBlock ("", "DSDT", 1, "Sample", "Sample", 0) {
        Method (ADD2, 2, NotSerialized) { // Arg0, Arg1
                Local0 = 0 // initialize destination
                Add (Arg0, Arg1, Local0) // Local0 := Arg0 + Arg1
                Return (Local0)
        }
}
```

Если поместить код из листинга 4а в файл **add2.asl** и вызвать **iasl** компилятор с указанием этого файла, то получим на выходе файл **add2.aml** содержащий AML байт-код уже упакованный в таблицу с сигнатурой "DSDT" и готовый к загрузке в OSPM. В листинге 4б приведен пример получения байт-кода (компиляции) и дамп результирующего файла.

Листинг 4б. Компиляция программы «Sample» в AML байт-код.

```
rz@butterfly:~/ACPI % cat add2.asl

DefinitionBlock ("", "DSDT", 1, "Sample", "Sample", 0) {
       Method (ADD2, 2, NotSerialized) { // Arg0, Arg1
              Local0 = 0 // initialize destination
              Add (Arg0, Arg1, Local0) // Local0 := Arg0 + Arg1
              Return (Local0)
       }
}

rz@butterfly:~/ACPI % iasl add2.asl

Intel ACPI Component Architecture
ASL+ Optimizing Compiler/Disassembler version 20201113
Copyright (c) 2000 - 2020 Intel Corporation

ASL Input:     add2.asl -     221 bytes      4 keywords     10 source lines
AML Output:    add2.aml -      52 bytes      3 opcodes       1 named objects

Compilation successful. 0 Errors, 0 Warnings, 0 Remarks, 1 Optimizations

rz@butterfly:~/ACPI % hexdump -C add2.aml
00000000  44 53 44 54 34 00 00 00  01 a6 53 61 6d 70 6c 65  |DSDT4.....Sample|
00000010  53 61 6d 70 6c 65 00 00  00 00 00 00 49 4e 54 4c  |Sample......INTL|
00000020  13 11 20 20 14 0f 41 44  44 32 02 70 00 60 72 68  |..  ..ADD2.p.`rh|
00000030  69 60 a4 60                                       |i`.`|
```

Как видно, результирующий файл **add2.aml** содержит 52 байта, 36 байт из которых представляют собой стандартный заголовок таблицы, за ним следует AML байт-код в шестнадцатеричном представлении: **14 0f 41 44 44 32 02 70 00 60 72 68 69 60 a4 60**. Чтобы лучше понять как работает AML машина, проведем ручное декомпилирование этого байт-кода, то есть разберем его по отдельным OP-кодам:

Байты **14 0f** — оператор **CreateName()**, указывают на то, что далее следует объявление метода, где **0f** — длина тела кода метода включая один байт длины.

Байты **41 44 44 32** задают четырех-символьное имя метода в текущем пространстве имен: "ADD2".

Байт **02** — указывает, что у метода имеется два входных параметра **Arg0** и **Arg1**, вызов метода осуществляется без «сериализации» (бит 3 - флага Serialize не установлен).

Байт **07** — оператор **Store()**.

Байт **00** — малая константа **Zero**, первый аргумент-источник.

Байт **60** — второй аргумент-приемник, локальная переменная **Local0**.

Байт **72** — оператор **Add() -** сложение двух аргументов и запись результата в третий.

Байт **68** — первый аргумент-источник, полученный на входе в метод параметр **Arg0**.

Байт **69** — второй аргумент-источник, полученный на входе в метод параметр **Arg1**.

Байт **60** — третий аргумент-приемник, локальная переменная **Local0**.

Байт **a4** — оператор **Return() -** возврат значения из метода.

Байт **60** — аргумент для возврата — локальная переменная Local0.

Таким образом, после ручной декомпиляции получим следующий код на языке ASL приведенный в Листинге 4в.

Листинг 4в. Декомпилированый AML байт-код в ASL представлении.

```
Method (ADD2, 2, NotSerialized)
{
    Store(Zero, Local0)
    Add(Arg0, Arg1, Local0)
    Return (Local0)
}
```

Чтобы убедиться в том, что мы верно провели ручное декомпилирование байт-кода, попросим выполнить эту же процедуру **iasl** компиятор. Для выполнения декомпиляции у него имеется ключик **-d**, а на вход передается готовая таблица. Скопируем файл **add2.aml** полученный в результате компиляции в новый файл **add2-1.aml**, и передадим его на вход в **iasl** с опцией **-d**. Результат автоматической декомпиляции приведен в Листинге 4г.

Листинг 4г. Декомпилирование AML байт-кода в ASL представление с помощью iasl.

```
rz@butterfly:~/ACPI % cp add2.aml add2-1.aml

rz@butterfly:~/ACPI % iasl -d add2-1.aml

Intel ACPI Component Architecture
ASL+ Optimizing Compiler/Disassembler version 20201113
Copyright (c) 2000 - 2020 Intel Corporation

File appears to be binary: found 18 non-ASCII characters, disassembling
Binary file appears to be a valid ACPI table, disassembling
```

```
Input file add2-1.aml, Length 0x34 (52) bytes
ACPI: DSDT 0x0000000000000000 000034 (v01 Sample Sample   00000000 INTL 20201113)
Pass 1 parse of [DSDT]
Pass 2 parse of [DSDT]
Parsing Deferred Opcodes (Methods/Buffers/Packages/Regions)

Parsing completed
Disassembly completed
ASL Output:    add2-1.dsl - 838 bytes

rz@butterfly:~/ACPI % cat add2-1.dsl
/*
 * Intel ACPI Component Architecture
 * AML/ASL+ Disassembler version 20201113 (64-bit version)
 * Copyright (c) 2000 - 2020 Intel Corporation
 *
 * Disassembling to symbolic ASL+ operators
 *
 * Disassembly of add2-1.aml, Sat Jun 13 20:20:02 2026
 *
 * Original Table Header:
 *     Signature        "DSDT"
 *     Length           0x00000034 (52)
 *     Revision         0x01 **** 32-bit table (V1), no 64-bit math support
 *     Checksum         0xA6
 *     OEM ID           "Sample"
 *     OEM Table ID     "Sample"
 *     OEM Revision     0x00000000 (0)
 *     Compiler ID      "INTL"
 *     Compiler Version 0x20201113 (538972435)
 */
DefinitionBlock ("", "DSDT", 1, "Sample", "Sample", 0x00000000)
{
    Method (ADD2, 2, NotSerialized)
    {
        Local0 = Zero
        Local0 = (Arg0 + Arg1)
        Return (Local0)
    }
}
```

При выполнении декомпиляции **iasl** старается заменять некоторые операторы выражениями более понятными человеку. Так, из примера выше видно, что оператор **Store()** заменен на оператор присваивания **«=»**, а оператор **Add() -** на оператор сложения **«+»**.

На этом закончим наш краткий курс по ASL/AML программированию, уверен что приведенной информации будет вполне достаточно чтобы понимать код добытый из «Firmware», чем мы далее и займемся. Замечу лишь, что AML является полной по Тьюрингу машиной, а её синтаксис и структура байт-кода была навеяна JVM и языком Java.

2. Анализируем таблицы ACPI «Firmware» от ноутбука «HP Laptop 15s»

В базовый комплект операционной системы FreeBSD уже входят утилиты для работы с таблицами ACPI, в том числе компилятор/декомпилятор **iasl**, отладчик AML кода **acpidb**, а так же утилита **acpudump**, позволяющая считывать содержимое всех таблиц ACPI и записывать в бинарные файлы. Но с этими утилитами имеется следующий нюанс — версия компилятор **iasl** в базовой поставке операционной системы реализует немного устаревшую спецификацию ACPI, что не всегда позволяет декомпилировать AML код с современных машин. К счастью, в репозитории FreeBSD имеется пакет **acpica-tools** содержащий самую свежую реализацию спецификации ACPI, плюс несколько дополнительных полезных утилит. Установить этот пакет можно традиционным способом:

```
rz@hp15s:~ % sudo pkg search acpica-tools
acpica-tools-20260408          Tools from the ACPI Component Architecture (ACPICA) project

rz@hp15s:~ % sudo pkg install acpica-tools
```

После установки пакета **acpica-tools** в системе будут находиться сразу две версии утилиты **iasl** и **acpidump**, и тут главное — не запутаться в том, какую именно версию утилиты Вы запускаете, так как синтаксис командной строки у них немного отличается, как отличается и результат работы. Комплект утилит входящий в базовый набор операцилнной системы FreeBSD (будем называет его BSD-версией) всегда располагается в системном каталоге **/usr/sbin**, в то время как вновь установленные утилиты из пакета располагается в каталоге пользовательских программ **/usr/local/bin**. Ниже показано как вызывать утилиты разных версии и не запутаться при этом.

Вызов утилиты **iasl** из системной поставки:

```
rz@hp15s:~ % /usr/sbin/iasl
Intel ACPI Component Architecture ASL+ Optimizing Compiler/Disassembler version
20221020 Copyright (c) 2000 - 2022 Intel Corporation
Supports ACPI Specification Revision 6.3
```

Вызов утилиты **acpidump** из системной поставки:

```
rz@hp15s:~ % /usr/sbin/acpidump -h
usage: /usr/sbin/acpidump [-d] [-t] [-h] [-v] [-f dsdt_input] [-o dsdt_output]
To send ASL:
/usr/sbin/acpidump -dt | gzip -c9 > foo.asl.gz
```

Вызов утилиты **iasl** входящей в пакет acpica-tools:

```
rz@hp15s:~ % /usr/local/bin/iasl
Intel ACPI Component Architecture
ASL+ Optimizing Compiler/Disassembler version 20260408
Copyright (c) 2000 - 2026 Intel Corporation

Supports ACPI Specification Revision 6.5
```

Вызов утилиты **acpidump** входящей в пакет acpica-tools:

```
rz@hp15s:~ % /usr/local/bin/acpidump -h
Usage: acpidump [options]
Options:
  -b                  Dump tables to binary files
  -h -?               This help message
  -o <File>           Redirect output to file
  -r <Address>        Dump tables from specified RSDP
  -s                  Print table summaries only
  -v                  Display version information
  -vd                 Display build date and time
  -z                  Verbose mode

Table Options:
  -a <Address>        Get table via a physical address
  -c <on|off>         Turning on/off customized table dumping
  -f <BinaryFile>     Get table via a binary file
  -n <Signature>      Get table via a name/signature
  -x                  Use RSDT instead of XSDT

Invocation without parameters dumps all available tables
Multiple mixed instances of -a, -f, and -n are supported
```

Как видно, утилита компилятора **iasl** из системы поддерживает спецификацию ACPI версии 6.3, а эта же утилита из пакета может работать со спецификацией 6.5. Также обратите внимание на то, что утилита **acpidump** из пакета имеет более насыщенный синтаксис командной строки и принимает гораздо больше опций.

В с пакетом **acpica-tools** поставляется еще одна полезная утилита — **acpihelp**, позволяющая получить справочную информацию по OP-кодам, операторам языка ASL и синтаксису языке.

Ниже приведен пример запроса справки на оператор **Notify**:

```
rz@hp15s:~ % acpihelp -a Notify

Intel ACPI Component Architecture
ACPI Help Utility version 20260408
Copyright (c) 2000 - 2026 Intel Corporation

          Notify: Notify Object of event
                  (Object, NotificationValue)

          NotifyOp: Opcode=0x86      Type (TermObject) FixedArgs (SuperName TermArg)
                                     DefNotify := NotifyOp NotifyObject NotifyValue
```

Далее, чтобы не запутаться, будем указывать полный путь к этим утилитам при их вызове.

2.1. Дамп таблиц и их декомпиляция

Теперь когда у нас имеется весь инструментарий, настало время снять дамп всех ACPI таблиц с исследуемого ноутбука. Сначала создадим подкаталог в который будут записаны все

таблицы (напомню, что их может быть несколько десятков), а затем вызовем утилиту **acpidump** с опцией **-b** для сохранения содержимого таблиц в бинарном виде:

```
rz@hp15s:~ % mkdir ACPI

rz@hp15s:~ % cd ACPI

rz@hp15s:~/ACPI % sudo /usr/local/bin/acpidump -b

rz@hp15s:~/ACPI % ls -al
total 424
drwxr-xr-x  2 rz   rz   1024 Jun 25 04:22 .
drwxr-xr-x  3 rz   rz   1536 Jun 25 04:22 ..
-rw-r--r--  1 rz  rz    222 May  9 02:21 apic.dat
-rw-r--r--  1 rz  rz     56 May  9 02:21 bgrt.dat
-rw-r--r--  1 rz  rz     41 May  9 02:21 cdit.dat
-rw-r--r--  1 rz  rz   2024 May  9 02:21 crat.dat
-rw-r--r--  1 rz  rz  84998 May  9 02:21 dsdt.dat
-rw-r--r--  1 rz  rz    276 May  9 02:21 facp.dat
-rw-r--r--  1 rz  rz     64 May  9 02:21 facs.dat
-rw-r--r--  1 rz  rz    156 May  9 02:21 fidt.dat
-rw-r--r--  1 rz  rz     68 May  9 02:21 fpdt.dat
-rw-r--r--  1 rz  rz     56 May  9 02:21 hpet.dat
-rw-r--r--  1 rz  rz    420 May  9 02:21 ivrs.dat
-rw-r--r--  1 rz  rz     60 May  9 02:21 mcfg.dat
-rw-r--r--  1 rz  rz     85 May  9 02:21 msdm.dat
-rw-r--r--  1 rz  rz  29244 May  9 02:21 ssdt.dat
-rw-r--r--  1 rz  rz    552 May  9 02:21 ssdt1.dat
-rw-r--r--  1 rz  rz   1303 May  9 02:21 ssdt10.dat
-rw-r--r--  1 rz  rz     66 May  9 02:21 ssdt11.dat
-rw-r--r--  1 rz  rz    522 May  9 02:21 ssdt12.dat
-rw-r--r--  1 rz  rz   1798 May  9 02:21 ssdt13.dat
-rw-r--r--  1 rz  rz    703 May  9 02:21 ssdt14.dat
-rw-r--r--  1 rz  rz     80 May  9 02:21 ssdt2.dat
-rw-r--r--  1 rz  rz   3596 May  9 02:21 ssdt3.dat
-rw-r--r--  1 rz  rz    313 May  9 02:21 ssdt4.dat
-rw-r--r--  1 rz  rz    551 May  9 02:21 ssdt5.dat
-rw-r--r--  1 rz  rz   3383 May  9 02:21 ssdt6.dat
-rw-r--r--  1 rz  rz   4261 May  9 02:21 ssdt7.dat
-rw-r--r--  1 rz  rz  12488 May  9 02:21 ssdt8.dat
-rw-r--r--  1 rz  rz    125 May  9 02:21 ssdt9.dat
-rw-r--r--  1 rz  rz     76 May  9 02:21 tpm2.dat
-rw-r--r--  1 rz  rz  54916 May  9 02:21 vfct.dat
-rw-r--r--  1 rz  rz     40 May  9 02:21 wsmt.dat
-rw-r--r--  1 rz  rz    268 May  9 02:21 xsdt.dat
```

Утилиту **acpidump** требуется запустить от имени суперпользователя (через **sudo** или иным способом), так как она работает с системной памятью через устройство **/dev/mem**. В результате, для нашей подопытной машины hp15s получим 32 файла с таблицами.

Следующим шагом вызовем утилиту **iasl** чтобы декомпилировать и декодировать содержащиеся в таблицах код и данные, указав в качества параметра опцию **-d** и список бинарных файлов:

```
rz@hp15s:~ACPI % /usr/local/bin/iasl -d *.dat

Intel ACPI Component Architecture
ASL+ Optimizing Compiler/Disassembler version 20260408
Copyright (c) 2000 - 2026 Intel Corporation

File appears to be binary: found 187 non-ASCII characters, disassembling
Binary file appears to be a valid ACPI table, disassembling
Input file apic.dat, Length 0xDE (222) bytes
ACPI: APIC 0x0000000000000000 0000DE (v04 HPQOEM 8707     01072009 HP   00010013)
Acpi Data Table [APIC] decoded
Formatted output:  apic.dsl - 10065 bytes

…
```

```
и так 32 раза
```

В приведенном выше примере утилита **iasl** читает бинарные данные из **.dat** файлов и помещает результат декомпиляции в аналогичный **.dsl** файл. В итоге получим вот такой список файлов с текстом языке ASL:

```
rz@hp15s:~/ACPI % ls -al *.dsl
-rw-r--r--  1 rz  rz   10065 May  9 02:21 apic.dsl
-rw-r--r--  1 rz  rz    1589 May  9 02:21 bgrt.dsl
-rw-r--r--  1 rz  rz    1098 May  9 02:21 cdit.dsl
-rw-r--r--  1 rz  rz   10896 May  9 02:21 crat.dsl
-rw-r--r--  1 rz  rz  589178 May  9 02:21 dsdt.dsl
-rw-r--r--  1 rz  rz   10144 May  9 02:21 facp.dsl
-rw-r--r--  1 rz  rz    1351 May  9 02:21 facs.dsl
-rw-r--r--  1 rz  rz    1655 May  9 02:21 fidt.dsl
-rw-r--r--  1 rz  rz    1787 May  9 02:21 fpdt.dsl
-rw-r--r--  1 rz  rz    1848 May  9 02:21 hpet.dsl
-rw-r--r--  1 rz  rz   23887 May  9 02:21 ivrs.dsl
-rw-r--r--  1 rz  rz    1509 May  9 02:21 mcfg.dsl
-rw-r--r--  1 rz  rz    1783 May  9 02:21 msdm.dsl
-rw-r--r--  1 rz  rz  208989 May  9 02:21 ssdt.dsl
-rw-r--r--  1 rz  rz    4705 May  9 02:21 ssdt1.dsl
-rw-r--r--  1 rz  rz   10431 May  9 02:21 ssdt10.dsl
-rw-r--r--  1 rz  rz     815 May  9 02:21 ssdt11.dsl
-rw-r--r--  1 rz  rz    5723 May  9 02:21 ssdt12.dsl
-rw-r--r--  1 rz  rz   15753 May  9 02:21 ssdt13.dsl
-rw-r--r--  1 rz  rz    7254 May  9 02:21 ssdt14.dsl
-rw-r--r--  1 rz  rz     933 May  9 02:21 ssdt2.dsl
-rw-r--r--  1 rz  rz   40313 May  9 02:21 ssdt3.dsl
-rw-r--r--  1 rz  rz    2298 May  9 02:21 ssdt4.dsl
-rw-r--r--  1 rz  rz    5532 May  9 02:21 ssdt5.dsl
-rw-r--r--  1 rz  rz   19876 May  9 02:21 ssdt6.dsl
-rw-r--r--  1 rz  rz   32887 May  9 02:21 ssdt7.dsl
-rw-r--r--  1 rz  rz   79276 May  9 02:21 ssdt8.dsl
-rw-r--r--  1 rz  rz    1787 May  9 02:21 ssdt9.dsl
-rw-r--r--  1 rz  rz    1708 May  9 02:21 tpm2.dsl
-rw-r--r--  1 rz  rz  272068 May  9 02:21 vfct.dsl
-rw-r--r--  1 rz  rz    1297 May  9 02:21 wsmt.dsl
-rw-r--r--  1 rz  rz    4086 May  9 02:21 xsdt.dsl
```

Посмотрим что там внутри. В файле **facp.dsl** содержится уже известная нам таблица FADT с параметрами подсистемы ACPI для данной машины:

```
rz@hp15s:~/ACPI % head -30 facp.dsl
/*
 * Intel ACPI Component Architecture
 * AML/ASL+ Disassembler version 20260408 (64-bit version)
 * Copyright (c) 2000 - 2026 Intel Corporation
 *
 * Disassembly of facp.dat
 *
 * ACPI Data Table [FACP]
 *
 * Format: [HexOffset DecimalOffset ByteLength]  FieldName : FieldValue (in hex)
 */

[000h 0000 004h]                   Signature : "FACP"    [Fixed ACPI Description Table
(FADT)]
[004h 0004 004h]                Table Length : 00000114
[008h 0008 001h]                    Revision : 06
[009h 0009 001h]                    Checksum : FF
[00Ah 0010 006h]                       Oem ID : "HPQOEM"
[010h 0016 008h]                Oem Table ID : "SLIC-MPC"
[018h 0024 004h]                Oem Revision : 0107200a
[01Ch 0028 004h]             Asl Compiler ID : "HP  "
[020h 0032 004h]       Asl Compiler Revision : 00010013

[024h 0036 004h]                FACS Address : C74FE000
[028h 0040 004h]                DSDT Address : C7241000
[02Ch 0044 001h]                       Model : 00
```

```
[02Dh 0045 001h]                 PM Profile : 02 [Mobile]
[02Eh 0046 002h]              SCI Interrupt : 0009
[030h 0048 004h]            SMI Command Port : 000000B2
[034h 0052 001h]           ACPI Enable Value : A0
[035h 0053 001h]          ACPI Disable Value : A1
```

Далее нам требуется декомпилировать таблицу DSDT отдельно (еще раз), так в ней используются внешние ссылки (external references) на объекты располагающиеся в нескольких вспомогательных таблицах SSDT. Для этого вызовем утилиту **iasl** перечислив **sstd*.dat** файлы в качестве внешних подключений с помощью опции **-e**:

```
rz@hp15s:~/ACPI/ACPI %  /usr/local/bin/iasl -e ssdt*.dat -d dsdt.dat

Intel ACPI Component Architecture
ASL+ Optimizing Compiler/Disassembler version 20260408
Copyright (c) 2000 - 2026 Intel Corporation

File appears to be binary: found 26884 non-ASCII characters, disassembling
Binary file appears to be a valid ACPI table, disassembling
Input file dsdt.dat, Length 0x14C06 (84998) bytes
ACPI: DSDT 0x0000000000000000 014C06 (v02 HPQOEM 8707     01072009 ACPI 20120913)
External object resolution file        ssdt9.dat
Input file ssdt9.dat, Length 0x7D (125) bytes
ACPI: SSDT 0x0000000000000000 00007D (v01 HPQOEM 8707     00000001 ACPI 20120913)
Pass 1 parse of [SSDT]
Pass 2 parse of [SSDT]

…

External object resolution file         ssdt.dat
Input file ssdt.dat, Length 0x723C (29244) bytes
ACPI: SSDT 0x0000000000000000 00723C (v02 HPQOEM 8707     00000002 ACPI 04000000)
Pass 1 parse of [SSDT]
Pass 2 parse of [SSDT]
Pass 1 parse of [DSDT]
Pass 2 parse of [DSDT]
Parsing Deferred Opcodes (Methods/Buffers/Packages/Regions)

Parsing completed

Found 8 external control methods, reparsing with new information
Pass 1 parse of [DSDT]
Pass 2 parse of [DSDT]
Parsing Deferred Opcodes (Methods/Buffers/Packages/Regions)

Parsing completed
Disassembly completed
ASL Output:    dsdt.dsl - 608125 bytes
```

Видно что размер файла **dsdt.dsl** существенно возрос, и в нем теперь можно наблюдать ASL код примерно следующего содержания:

```
rz@hp15s:~/ACPI/ACPI % head -50  dsdt.dsl
/*
 * Intel ACPI Component Architecture
 * AML/ASL+ Disassembler version 20260408 (64-bit version)
 * Copyright (c) 2000 - 2026 Intel Corporation
 *
 * Disassembling to symbolic ASL+ operators
 *
 * Disassembly of dsdt.dat
 *
 * Original Table Header:
 *     Signature        "DSDT"
 *     Length           0x00014C06 (84998)
 *     Revision         0x02
 *     Checksum         0xC9
 *     OEM ID           "HPQOEM"
 *     OEM Table ID     "8707    "
```

```
 *     OEM Revision     0x01072009 (17244169)
 *     Compiler ID      "ACPI"
 *     Compiler Version 0x20120913 (538052883)
 */
DefinitionBlock ("", "DSDT", 2, "HPQOEM", "8707    ", 0x01072009)
{
    External (_PR_.P000, UnknownObj)
    External (_PR_.P000.PPCV, UnknownObj)
    External (_PR_.P001, UnknownObj)
    External (_PR_.P001.PPCV, UnknownObj)
    External (_PR_.P002, UnknownObj)
    External (_PR_.P002.PPCV, UnknownObj)
    External (_PR_.P003, UnknownObj)
    External (_PR_.P003.PPCV, UnknownObj)
    External (_PR_.P004, UnknownObj)
    External (_PR_.P005, UnknownObj)
    External (_PR_.P006, UnknownObj)
    External (_PR_.P007, UnknownObj)
    External (_PR_.P008, UnknownObj)
    External (_PR_.P009, UnknownObj)
    External (_PR_.P00A, UnknownObj)
    External (_PR_.P00B, UnknownObj)
    External (_PR_.P00C, UnknownObj)
    External (_PR_.P00D, UnknownObj)
    External (_PR_.P00E, UnknownObj)
    External (_PR_.P00F, UnknownObj)
    External (_SB_.ALIB, MethodObj)    // 2 Arguments
    External (_SB_.APTS, MethodObj)    // 1 Arguments
    External (_SB_.AWAK, MethodObj)    // 1 Arguments
    External (_SB_.PCI0.GP17.VGA_.AFN7, MethodObj)    // 1 Arguments
    External (_SB_.PCI0.GPP0.PEGP, UnknownObj)
    External (AFN4, MethodObj)    // 1 Arguments
    External (AFN7, MethodObj)    // 1 Arguments
    External (MPTS, MethodObj)    // 1 Arguments
    ...
```

Для ряда методов теперь указано число параметров полученное из подключенных **ssdt*.dat** файлов.

Где-то там, в недрах файла **dsdt.dsl**, можно обнаружить код объявления объекта **_SB.PWRB** типа «Device» для кнопки «Power» без каких либо методов, но с тремя динамическими переменными:

```
    Scope (_SB)
    {
        Device (PWRB)
        {
            Name (_HID, EisaId ("PNP0C0C") /* Power Button Device */)  // _HID: Hardware ID
            Name (_UID, 0xAA)  // _UID: Unique ID
            Name (_STA, 0x0B)  // _STA: Status
        }
    }
```

Такое описание устройства **_SB.PWRB** говорит о том, что запросы к нему на выяснение состояния кнопки не обслуживаются AML кодом, т. е. OSPM получает эту информацию иным путем: либо как «Fixed Feature» (из регистров группы **PM1_STATUS**), либо через нотификации, но обо этом чуть позже.

2.2. Дамп таблиц и их декомпиляция (BSD-версия)

У родной (BSD-версии) утилиты **acpidump** имеется возможность сдампить содержимое всех таблиц SSDT и DSDT, после чего декомпилировать этот бинарный «блоб», соединить результирующий ASL код в один длинный текст и поместить его в файл с заголовком сигнатуры DSDT. Такой объединенный вариант «исходного кода» может потребоваться в случае, если необходимо исправить ошибку в «Fimrware»: отредактировать код, скомпилировать его, и «подсунуть» операционной системе вместо оригинала «Firmware» получаемого от BIOS. Далее мы посмотрим как это делается, а здесь следует отметить что механизм внутри ядра ОС FreeBSD позволяется подсунуть системе только одну таблицу с именем DSDT, таблицы SSDT системой умышленно игнорируются! Зачем разработчики ядра ОС FreeBSD ввели такое ограничение мне не понятно, и с моей точки зрения это не очень удобно. Но что есть, то есть. Скорее всего, по этой же неизвестной причине существует и отдельная BSD версия утилиты **acpidump**.

Итак, чтобы получить объединенный исходный код таблиц DSDT+SSDT, следует выполнить команду:

```
rz@hp15s:~/ACPI % sudo /usr/sbin/acpidump -dt > dsdt_ssdt_unified.asl
```

Исходный код DSDT и SSDT таблиц будет представлен в файле **dsdt_ssdt_unified.asl** весьма внушительного объема:

```
rz@hp15s:~/ACPI % ls -al dsdt_ssdt_unified.asl

-rw-r--r--  1 rz rz 1258391 Jul 24 03:49 dsdt_ssdt_unified.asl
```

На всякий случай сообщаю, что в случае если Вы выявили ошибку в ядре ОС связанную с подсистемой ACPI, полученный таким образом файл (только пожатый утилитой **gzip**) следует отправлять в BUG-репорт на сайте bugs.freebsd.org.

2.3. Компиляция ASL кода и подстановка DSDT таблицы

Помимо того, чтобы снимать дамп и дизассемблировать таблицы ACPI, у нас имеются всех необходимые инструменты для того, чтобы внести изменения в код «Firmware», скомпилировать байт-код и подменить таблицу DSDT при загрузке операционной системы. Как отмечалось выше, причины которые могу побудить пользователя вносить изменения в «Firmware» достаточно прозаичны — в «Firmware» могут присутствовать ошибки или некоторые несовместимости, исправив которые можно решить часть проблем связанных с управлением электропитанием на ноутбуках или выявить и устранить проблемы инициализации сложного и редкого оборудования на серверах. Более подробно об этом написано в главе «11.16 Using and Debugging FreeBSD ACPI» в известном руководстве «FreeBSD Handbook». Вот небольшая цитата из него:

> *In the long run, our goal is for almost everyone to have ACPI work without any user intervention. At this point, however, we are still developing workarounds for common mistakes made by the BIOS vendors.* ***The Microsoft® interpreter (acpi.sys and acpiec.sys) does not strictly check for adherence to the standard, and thus many BIOS vendors who only test ACPI under Windows® never fix their ASL.*** *We hope to continue to identify and*

document exactly what non-standard behavior is allowed by Microsoft's interpreter and replicate it so FreeBSD can work without forcing users to fix the ASL. ***As a workaround and to help us identify behavior, you can fix the ASL manually.*** *If this works for you, please send a diff(1) of the old and new ASL so we can possibly work around the buggy behavior in ACPI-CA and thus make your fix unnecessary.*

Для того, чтобы скомпилировать байт-код по исходному ASL описанию, достаточно вызвать утилиту **iasl** и передать ей имя файла с исходным кодом в качестве параметра. Обычно, в качестве такого исходника служит одна из декомпилированных таблиц, например DSDT. Однако, иногда, полученный в результате декомпиляции исходный код может содержать «синтаксические» ошибки по причине того, что оригинальный AML был получен с помощью совершенно другого компилятора. Эти ошибки и предупреждения вызывают останов утилиты **iasl** при выполнении компиляции. Чтобы обойти их, достаточно указать опцию **-f**, после чего компилятор будет работать без остановки.

Попробуем внести какие-нибудь изменения в ASL код хранящийся в таблице DSDT, пересобрать и подменить её. Первым делом сделаем копию кода из оригинального файла **dsdt.dsl** в файл **dsdt_mod1.dsl** с которым будем далее работать. Откроем в текстовом редакторе файл **dsdt_mod1.dsl** и в самый его конец, *перед последней закрывающейся фигурной скобкой*, добавим код из листинга 5а приведенного ниже.

Листинг 5а. Тестовый ASL код объявления фейкового устройства _SB.TEST.

```
Scope (_SB)
{
    Device (TEST)
    {
        Name (_HID, "NPN0666"); // Fake HID value
        Method (_DSM, 4, Serialized)  // _DSM: Device-Specific Method always takes 4 args
        {
            If (Arg0 == 1)
            {
                return (0xDEADBEEF)
            }
            Else
            {
                return (0xBAAAAAAD)
            }
        }
    }
}
```

Данный ASL код объявляет новый объект **TEST** типа «Device» в дереве пространства имён ACPI, а у объекта объявляет метод **_DST** (в принципе, здесь может быть любое имя согласно требованиям спецификации на обозначение узлов). Этот метод объявляется с 4-мя параметрами, потому что, по спецификации ACPI, метод **_DSM** является служебным и он обязан принимать четыре параметра. В нашей реализации метод **_DSM** обработает только они, первый, параметр согласно простому алгоритму: если параметр равен целочисленной единице, то метод вернет шестнадцатеричное значение **0xDEADBEEF**, иначе — вернет значение **0xBAAAAAAD**.

Скомпилируем новую таблицу DSDT из модифицированного файла с исходным кодом:

```
rz@hp15s:~/ACPI % /usr/local/bin/iasl -f dsdt_mod1.dsl
```

```
Intel ACPI Component Architecture
ASL+ Optimizing Compiler/Disassembler version 20260408
Copyright (c) 2000 - 2026 Intel Corporation

Ignoring all errors, forcing AML file generation

…

ASL Input:     dsdt_mod1.dsl -  312452 bytes   3888 keywords      0 source lines
AML Output:    dsdt_mod1.aml -   37794 bytes   2366 opcodes    1522 named objects

Compilation successful. 0 Errors, 38 Warnings, 167 Remarks, 27 Optimizations
```

В моем случае компилятор выдал 38 предупреждений и еще 167 каких-то ремарок. Чтож, примем к сведению и посмотрим на результирующий файл:

```
rz@hp15s:~/ACPI % ls -al dsdt_mod1*
-rw-r--r--  1 rz rz  37794 Jul  8 03:46 dsdt_mod1.aml
-rw-r--r--  1 rz rz 312452 Jul  8 03:44 dsdt_mod1.dsl
```

Теперь нам требуется «подсунуть» эту новую DSDT таблицу, находящуюся в файле **dsdt_mod1.aml**, операционной системе так, чтобы при следующей загрузке она использовала данные из этого файла, а не из «родной» таблицы предоставляемой BIOS-ом. Для этого скопируем файл **dsdt_mod1.aml** в системный каталог **/boot/**, добавим пару строк в конфигурационный файл **/boot/loader.conf** загрузчика операционной системы FreeBSD и отправим систему на перезагрузку:

```
rz@hp15s:~/ACPI % sudo scp dsdt_mod1.aml /boot/

rz@hp15s:~/ACPI % sudo sysrc -f /boot/loader.conf acpi_dsdt_load="YES"
acpi_dsdt_load:  -> YES

rz@hp15s:~/ACPI % sudo sysrc -f /boot/loader.conf acpi_dsdt_name="/boot/dsdt_mod1.aml"
acpi_dsdt_name:  -> /boot/dsdt_mod1.aml

rz@hp15s:~/ACPI % sudo reboot
```

Сразу подскажу как вернуть обратно «дефолтную» таблицу. Для этого достаточно установить конфигурационную переменную **acpi_dsdt_load** в значение "**NO**", или просто удалить файл **/boot/dsdt_mod1.aml**. Перед этим необходимо загрузить систему в «Single User Mode» (кнопка «S» в меню загрузчика), и выполнить проверку файловой системы командой **fsck** и перемонтировать её в режим «read-write» командой **mount**:

```
root@:~ # fsck -y
** /dev/nda0p2
…
**** FILE SYSTEM MARKED CLEAN ****

root@:~ # mount -a

root@:~ # sysrc -f /boot/loader.conf acpi_dsdt_load="NO"
acpi_dsdt_load: YES -> NO
```

Но вернемся к подмене таблицы DSDT. В процессе загрузки ОС с установленной опцией **acpi_dsdt_load="YES"**, загрузчик подгрузит в ОЗУ содержимое файла **/boot/dsdt_mod1.aml** и передаст коду инициализации ACPICA внутри ядра операционной системы. Далее, подсистема ACPICA обнаружит там сигнатуру таблицы "DSDT" и заменит новым содержимым старую таблицу полученную от BIOS.

После перезагрузки системы встает вопрос — как выяснить, загрузилось ли наше обновление DSDT таблицы в подсистему ACPICA или нет ? Скажу честно, я не обнаружил метода позволяющего выполнить дамп текучей загруженной в систему таблицы. Если же теперь еще раз выполнить дамп таблиц утилитой **acpidump**, то она выдаст нам содержимое из области ОЗУ в которую BIOS разместил свою версию «Firmware», то есть мы получим всё те же «дефолтные» данные без изменения. У утилиты **acpidump** есть параметр "**-c on**" который как бы предназначен для того, чтобы получить загруженные пользователем данные (customized tables), но по какой-то причине у меня это не сработало. Возможно, что для ОС FreeBSD этот параметр не работает или сломан.

Но способ отследить какие ACPI таблицы обрабатываются в процессе загрузки ОС FreeBSD всё же есть!

2.4. Загрузка ядра ОС FreeBSD в режиме «многословности» (Verbose Mode)

Ядро ОС FreeBSD представляет собой взаимосвязанный набор объектов - так называемых «Kernel Objects» (или «Kobjs»). Каждый объект «Kobj» представлен набором процедур (методов) реализующих какую-то функцию операционной системы: планировщик задач, драйвер физического или виртуального устройства, и т.д. Почти каждый метод содержит код для генерации диагностической и отладочной информации. Небольшую часть этой информации пользователь видит на экране консоли в процессе загрузки ОС или при возникновении фатальных ошибок. Существует простой способ включить более расширенный вывод диагностической информации, или так называемы «многословный» (Verbose Mode) режим. Для его включения достаточно добавить в файл конфигурации загрузчика **/boot/loader.conf** опцию **debug.bootverbose="1"** и перезагрузить систему. Этот же «многословный» режим включается командой **boot -v** из командной строки загрузчика, попасть в которую можно из основного меню загрузчика нажатием кнопки «**3**» (выбор меню «*3. Escape to loader prompt*»).

В «многословном» режиме ядро ОС FreeBSD будет выдавать на консоль огромный объем отладочной информации, вся она будет сохранена в лог файл **/var/log/messages**, а также доступна по команде **dmesg**. Включение и выключение этого режима можно производить прямо на рабочей (уже загруженной) системе с помощью команды **sysctl**:

```
rz@hp15s:~ % sudo sysctl debug.bootverbose
debug.bootverbose: 0
rz@hp15s:~ % sudo sysctl debug.bootverbose=1
debug.bootverbose: 1 -> 1
```

Это свойство системы можно использовать для диагностирования различных проблем связанных с инициализацией аппаратуры в процессе загрузки ОС или при загрузке/выгрузке модулей ядра по ходу работы ОС. Часто бывает так, что новое и еще не поддерживаемое

оборудование может приводить к зависанию системы прямо в процессе загрузки, режим «многословности» позволяет определить какой именно «кусок железа» приводит к таким неприятным последствия и временно отключить его.

И тут есть один важный нюанс: отладочной информации выдаваемой ядром ОС на столько много, что вся она не помещается в кольцевой буфер системных сообщений, а значит при просмотре его утилитой **dmesg** мы будем наблюдать только самый конец загрузки! Это легко исправить кратно увеличив размер кольцевого буфера путем установки значения системной переменной **kern.msgbufsize** в конфигурационном файле загрузчика. К сожалению, утилита **sysrc** не позволит нам установить значение переменной потому, что в названии переменной имеется недопустимый, с точки зрения **sysrc**, символ «точка». Поэтому воспользуемся более сложным способом:

```
# Получить текущее значение размера кольцевого буфера

rz@hp15s:~ % sysctl kern.msgbufsize
kern.msgbufsize: 98304

# Записать в конфиг строку присваивающую переменной новое десятикратное значение

rz@hp15s:~ % sudo /bin/sh -c 'echo kern.msgbufsize=983040 >> /boot/loader.conf'
```

В нашем случае режим «многословности» позволяет подсмотреть как идет обработка таблиц DSDT. Для этого, после установки увеличенного размера кольцевого буфера и загрузки системы с включенной опцией «многословности», сделаем выборку отладочной информации из **dmesg** по строке **"ACPI:"**. На машине БЕЗ подставленной таблицы получим примерно следующий результат:

```
rz@hp15s:~ % dmesg | grep ACPI:
ACPI: RSDP 0x00000000C7262014 000024 (v02 HPQOEM)
ACPI: XSDT 0x00000000C7261728 00010C (v01 HPQOEM SLIC-MPC 01072009 AMI  01000013)
ACPI: FACP 0x00000000C7256000 000114 (v06 HPQOEM SLIC-MPC 01072009 HP   00010013)
ACPI: DSDT 0x00000000C7241000 014C06 (v02 HPQOEM 8707     01072009 ACPI 20120913)
ACPI: FACS 0x00000000C74FE000 000040
ACPI: MSDM 0x00000000C7260000 000055 (v03 HPQOEM SLIC-MPC 00000001 HP   00010013)
ACPI: SSDT 0x00000000C7258000 00723C (v02 HPQOEM 8707     00000002 ACPI 04000000)
ACPI: IVRS 0x00000000C7257000 0001A4 (v02 HPQOEM 8707     00000001 HP   00000000)
ACPI: FIDT 0x00000000C7240000 00009C (v01 HPQOEM 8707     01072009 HP   00010013)
ACPI: MCFG 0x00000000C723F000 00003C (v01 HPQOEM 8707     01072009 HP   00010013)
ACPI: HPET 0x00000000C723E000 000038 (v01 HPQOEM 8707     01072009 HP   00000005)
ACPI: SSDT 0x00000000C723D000 000228 (v01 HPQOEM 8707     00000001 ACPI 20120913)
ACPI: FPDT 0x00000000C723C000 000044 (v01 HPQOEM 8707     01072009 HP   01000013)
ACPI: VFCT 0x00000000C722E000 00D684 (v01 HPQOEM 8707     00000001 HP   31504F47)
ACPI: SSDT 0x00000000C722D000 000050 (v01 HPQOEM 8707     00000001 ACPI 20120913)
ACPI: TPM2 0x00000000C722C000 00004C (v04 HPQOEM 8707     00000001 HP   00000000)
ACPI: SSDT 0x00000000C722B000 000E0C (v01 HPQOEM 8707     00000001 ACPI 00000001)
ACPI: CRAT 0x00000000C722A000 0007E8 (v01 HPQOEM 8707     00000001 HP   00000001)
ACPI: CDIT 0x00000000C7229000 000029 (v01 HPQOEM 8707     00000001 HP   00000001)
ACPI: SSDT 0x00000000C7228000 000139 (v01 HPQOEM 8707     00000001 ACPI 20120913)
ACPI: SSDT 0x00000000C7227000 000227 (v01 HPQOEM 8707     00000001 ACPI 20120913)
ACPI: SSDT 0x00000000C7226000 000D37 (v01 HPQOEM 8707     00000001 ACPI 20120913)
ACPI: SSDT 0x00000000C7224000 0010A5 (v01 HPQOEM 8707     00000001 ACPI 20120913)
ACPI: SSDT 0x00000000C7220000 0030C8 (v01 HPQOEM 8707     00000001 ACPI 20120913)
ACPI: WSMT 0x00000000C721F000 000028 (v01 HPQOEM 8707     01072009 HP   00010013)
ACPI: APIC 0x00000000C721E000 0000DE (v04 HPQOEM 8707     01072009 HP   00010013)
ACPI: SSDT 0x00000000C721D000 00007D (v01 HPQOEM 8707     00000001 ACPI 20120913)
ACPI: SSDT 0x00000000C721C000 000517 (v01 HPQOEM 8707     00000001 ACPI 20120913)
ACPI: SSDT 0x00000000C721B000 000042 (v01 HPQOEM 8707     00000001 ACPI 20120913)
ACPI: SSDT 0x00000000C721A000 00020A (v01 HPQOEM 8707     00000001 ACPI 20120913)
```

```
ACPI: SSDT 0x00000000C7219000 000706 (v01 HPQOEM 8707     00000001 ACPI 20120913)
ACPI: SSDT 0x00000000C7218000 0002BF (v01 HPQOEM 8707     00000001 ACPI 20120913)
ACPI: BGRT 0x00000000C7217000 000038 (v01 HPQOEM 8707     01072009 HP   00010013)
ACPI: 16 ACPI AML tables successfully acquired and loaded
```

Видно, что система загрузила 16 таблиц полученных от BIOS. Также сразу бросается в глаза тот факт, что весь AML код для этого, вполне современного, ноутбука был сформирован OEM производителем с помощью некоего компилятора «ACPI» весьма старой версии (20120913), что намекает на наличие проблем с семантикой в коде. Скорее всего китайский производитель ноутбука приобрел лицензию на код «Firmware» у компании «HP» много лет назад и с тех пор ни разу его не обновлял и не изменял.

Теперь подставим наш вариант таблицы DSDT с AML кодом в файле **/boot/dsdt_mod1.aml**, включим опцию **acpi_dsdt_load="YES"**, еще раз загрузим систему в «многословном» режиме и посмотрим какие на этот раз таблицы попадают в ACPICA:

```
rz@hp15s:~ % dmesg | grep ACPI:
ACPI: RSDP 0x00000000C7262014 000024 (v02 HPQOEM)
ACPI: XSDT 0x00000000C7261728 00010C (v01 HPQOEM SLIC-MPC 01072009 AMI  01000013)
ACPI: FACP 0x00000000C7256000 000114 (v06 HPQOEM SLIC-MPC 01072009 HP   00010013)
ACPI: DSDT 0x00000000C7241000 Logical table override, new table: 0xFFFFFFFF820F8000
ACPI: DSDT 0xFFFFFFFF820F8000 014D40 (v02 HPQOEM 8707     01072009 INTL 20260408)
ACPI: FACS 0x00000000C74FE000 000040
ACPI: MSDM 0x00000000C7260000 000055 (v03 HPQOEM SLIC-MPC 00000001 HP   00010013)
ACPI: Ignoring installation of SSDT at 00000000C7258000
ACPI: IVRS 0x00000000C7257000 0001A4 (v02 HPQOEM 8707     00000001 HP   00000000)
ACPI: FIDT 0x00000000C7240000 00009C (v01 HPQOEM 8707     01072009 HP   00010013)
ACPI: MCFG 0x00000000C723F000 00003C (v01 HPQOEM 8707     01072009 HP   00010013)
ACPI: HPET 0x00000000C723E000 000038 (v01 HPQOEM 8707     01072009 HP   00000005)
ACPI: Ignoring installation of SSDT at 00000000C723D000
ACPI: FPDT 0x00000000C723C000 000044 (v01 HPQOEM 8707     01072009 HP   01000013)
ACPI: VFCT 0x00000000C722E000 00D684 (v01 HPQOEM 8707     00000001 HP   31504F47)
ACPI: Ignoring installation of SSDT at 00000000C722D000
ACPI: TPM2 0x00000000C722C000 00004C (v04 HPQOEM 8707     00000001 HP   00000000)
ACPI: Ignoring installation of SSDT at 00000000C722B000
ACPI: CRAT 0x00000000C722A000 0007E8 (v01 HPQOEM 8707     00000001 HP   00000001)
ACPI: CDIT 0x00000000C7229000 000029 (v01 HPQOEM 8707     00000001 HP   00000001)
ACPI: Ignoring installation of SSDT at 00000000C7228000
ACPI: Ignoring installation of SSDT at 00000000C7227000
ACPI: Ignoring installation of SSDT at 00000000C7226000
ACPI: Ignoring installation of SSDT at 00000000C7224000
ACPI: Ignoring installation of SSDT at 00000000C7220000
ACPI: WSMT 0x00000000C721F000 000028 (v01 HPQOEM 8707     01072009 HP   00010013)
ACPI: APIC 0x00000000C721E000 0000DE (v04 HPQOEM 8707     01072009 HP   00010013)
ACPI: Ignoring installation of SSDT at 00000000C721D000
ACPI: Ignoring installation of SSDT at 00000000C721C000
ACPI: Ignoring installation of SSDT at 00000000C721B000
ACPI: Ignoring installation of SSDT at 00000000C721A000
ACPI: Ignoring installation of SSDT at 00000000C7219000
ACPI: Ignoring installation of SSDT at 00000000C7218000
ACPI: BGRT 0x00000000C7217000 000038 (v01 HPQOEM 8707     01072009 HP   00010013)
ACPI: 1 ACPI AML tables successfully acquired and loaded
```

Тут мы видим, что система распознала «подстановку» таблицы DSDT, загрузила её модифицированную (локальную) версию и при этом исключила из загрузки все таблицы SSDT полученные от BIOS. Как упоминалось ранее, исключение SSDT при загрузке «кастомной» версии таблицы DSDT сделано умышлено (предположительно с целью отладки). Если требуется загрузить код в том числе и из SSDT, то необходимо воспользоваться BSD версией утилиты **acpidump** чтобы получить объединенный дамп и работать уже с ним.

Теперь мы можем проверить вставленный нами AML код путем вызова ACPI метода **_DSM** у объекта **_SB.TEST**. О том, как это сделать я расскажу в следующей главе, а сейчас пару слов о том, как восстановить систему если что-то пошло не так.

2.5. Если что-то пошло не так...

Если ОС не сможет загрузить рабочий AML код, то подсистема ACPICA будет остановлена, а следовательно б*о*льшая часть аппаратуры останется не инициализированной — обычно это приводит к отсутствию дисковых накопителей и невозможности смонтировать корневую файловую систему. Для неподготовленного пользователя такая ситуация может показаться фатальной. Но не спешите расстраиваться, восстановить прежнее состояние системы очень просто — необходимо загрузить ядро системы без каких либо опций, отменив все «подстановки» и т. д.

Для этого в меню загрузчика выбираем пункт «*3. Escape to loader prompt*», после чего в командой строке вводим команду **load /boot/kernel/kernel**, а за ней команду **boot -s**:

```
Exiting menu!

Type '?' for a list of commands, 'help' for more detailed help.

OK load /boot/kernel/kernel
/boot/kernel/kernel text=0x188c70 text=0xde25ec text=0x44cf2e data=0x180+0xe80
data=0x198f60+0x8670a0 0x8+0x196938+0x8+0x1bb288/

OK boot -s
```

В этом режиме ядро системы загрузится в первозданном виде (Single User Mode). Сразу после успешной загрузки и появления приглашения вида #, необходимо:

1. Проверить целостность всех файловых систем командой **fsck**:

```
root@:~ # fsck -y
```

2. Смонтировать все файловые систему в режиме «read-write»:

```
root@:~ # mount -a
```

3. Исправить проблему — удалить нерабочий AML, выключить опцию подстановки DSDT таблицы или убрать все сомнительные опции из **/boot/loader.config**.

4. После чего еще раз перезагрузить операционную систему командой **reboot**.

«EasyPeasy!» как говорят французы. ;)

2.6. Вызов метода ACPI из командной строки

К сожалению, в ОС FreeBSD нет стандартной утилиты, и даже не предусматривается системой способа работы с ACPI объектами из «user-space», так как в ядре нет соответствующего системного вызова (устройства). Однако, в репозитории обнаружился пакет под названием **acpi_call** содержащий модуль ядра (драйвер) и одноименную утилиту,

позволяющие осуществлять вызов методов у ACPI объектов. Чтобы воспользоваться этим средством, установим пакет **acpi_call** в систему и загрузим в память модуль ядра **acpi_call.ko**.

```
rz@hp15s:~/ACPI/TEST % sudo pkg install acpi_call

rz@hp15s:~/ACPI/TEST % sudo kldload acpi_call
```

Убедимся в том, что модуль действительно загрузился:

```
rz@hp15s:~/ACPI/TEST % sudo kldstat | grep acpi_call
 29    1 0xffffffff8386e000     20c0 acpi_call.ko
```

Теперь можно осуществить вызов нашего инжектированного метода **_SB.TEST._DSM** с помощью утилиты **acpi_call**. При вызове будем передавать один единственный параметр типа Integer (опция **-i**):

```
rz@hp15s:~/ACPI/TEST % sudo acpi_call -p \\_SB.TEST._DSM -i 0
3131746989
rz@hp15s:~/ACPI/TEST % sudo acpi_call -p \\_SB.TEST._DSM -i 1
3735928559
rz@hp15s:~/ACPI/TEST % sudo acpi_call -p \\_SB.TEST._DSM -i 2
3131746989
```

Возвращаемые методом числа изменяются в зависимости от входного параметра. Если перевести эти десятичные числа (**3131746989** и **3735928559**) в шестнадцатеричный формат, то получим **0xBAAAAAAD** и **0xDEADBEEF**. Таким образом мы видим, что наш AML код внутри операционной системы работает и реагирует на вызовы снаружи.

Интереса ради попробуем вызвать несуществующий метод **_XXX**:

```
rz@hp15s:~/ACPI/TEST % sudo acpi_call -p \\_SB.TEST._XXX -i 1

Unknown object type '0'
```

2.7. Информация о поддерживаемых ОС и её свойствах

В иерархии стандартных методов ACPI имеются два интересных метода расположенных в самом корне дерева — методы **_OSI и _OS**. С помощью этих методов «Firmware» может сделать запрос в операционную систему и узнать тип операционной системы, выяснить поддерживает ли она определенный набор свойств - так называемых «интерфейсов» (Operating System Interfaces).

2.7.1. Метод _OSI для запроса «Operating System Interfaces»

Работает этот механизм весьма необычным способом. Внутри операционной системы должна присутствовать **имплементация метода _OSI**, которая принимает на вход всего один параметр — строку символов (некий «идентификатор свойства»), а на выходе возвращает **True** или **False** в зависимости о того, поддерживается ли операционной системой данное свойство или нет. AML код «firmware» в процессе инициализации, обычно это

происходит в методе **_SB._INI**, производит многократный вызов метода **_OSI** с перебором параметров, чтобы выяснить какие свойства поддерживаются операционной системой, и меняет свое поведение соответственно. То есть вызов метода **_OSI** происходит в противоположном направлении — от AML кода к операционной системе. От того как на эти запросы ответит операционная система, может зависеть поведение всей подсистемы ACPI.

Метод **_OSI** появился в спецификации ACPI версии 3.0. Спецификацией предписывается, что строка идентифицирующая свойство должна состоять и двух других подстрок: «Operating System Vendor String Prefixes» и «Standard ACPI-Defined Feature Group Strings». По факту, первая подстрока представляет собой идентификатор (название) операционной системы ("**Windows**", "**Linux**", "**HP-UX**", "**FreeBSD**", "**OpenVMS**" — это весь список легальных на данный момент идентификаторов). Вторая подстрока задает версию ОС, причем определено это только для ОС Windows. Таким образом AML код может спросить операционную систему поддерживает ли она, например, интерфейс "**Windows 2001**", "**Windows 2001.1 SP1**" или "**Windows 2022**", или даже "**FreeBSD**" или "**OpenVMS**"!

После декомпиляции таблицы DSDT, снятой с ноутбука «HP Laptop 15s» моего коллеги, я обнаружил в коде «Firmware» следующие виды интерфейсов ОС, запросы для которых производит AML код. Этот код запоминает ответ и каким-то образом изменяет логику поведения в зависимости от того, что он запомнил:

```
_OSI ("Windows 2001")
_OSI ("Windows 2001.1")
_OSI ("Windows 2001 SP1")
_OSI ("Windows 2001 SP3")
_OSI ("Windows 2006")
_OSI ("Windows 2006 SP1")
_OSI ("Windows 2009")
_OSI ("Windows 2012")
_OSI ("Windows 2013")
_OSI ("Windows 2015")
_OSI ("Linux")
_OSI ("FreeBSD")
_OSI ("HP-UX")
```

И даже вот такое:

```
_OSI ("OpenVMS")
```

Не без удивления я обнаружил, что на HP-шном ноутбуке предусмотрена адаптация под ОС FreeBSD. Но еще больше меня удивило то, что по умолчанию (если ответы на все запросы были **False**) «Firmware» на этом ноутбуке устанавливает значения внутренних параметров как для "**Linux**".

Мне стало интересно как ОС FreeBSD отвечает на такие запросы, и я выполнил парочку запросов вручную с помощью всё той же утилиты **acpi_call**:

```
rz@hp15s:~ % sudo acpi_call -p \\_OSI -s "Windows 2001"
18446744073709551615

rz@hp15s:~ % sudo acpi_call -p \\_OSI -s "Windows 2001 SP2"
18446744073709551615

rz@hp15s:~ % sudo acpi_call -p \\_OSI -s "Windows 2022"
18446744073709551615

rz@hp15s:~ % sudo acpi_call -p \\_OSI -s "Linux"
0
```

```
rz@hp15s:~ % sudo acpi_call -p \\_OSI -s "FreeBSD"
0
```

На первые три запроса я получил в ответ длинное число **18446744073709551615** — в двоичной форме это число состоит из 64-х единиц, т. е. максимальное значение для 64-х битового числа. Напомню, что таким образом в AML представляется булевое значение **True**, (булевое значение **False** представляется нулем - все биты в «0»).

Получается, что операционной системой FreeBSD «как бы поддерживаются» интерфейсы от ОС «Windows» вплоть до «Windows 2022», но не поддерживается интерфейс ОС «Linux» и, что поразительно, не поддерживается интерфейс самой ОС «FreeBSD». Немного полистав исходные коды ядра операционной системы, я обнаружил в недрах реализации ACPICA, в файле **sys/contrib/dev/acpica/components/utilities/utosi.c**, следующий интересный комментарий:

```
/*
 * Strings supported by the _OSI predefined control method (which is
 * implemented internally within this module.)
 *
 * March 2009: Removed "Linux" as this host no longer wants to respond true
 * for this string. Basically, the only safe OS strings are windows-related
 * and in many or most cases represent the only test path within the
 * BIOS-provided ASL code.
 *
 * The last element of each entry is used to track the newest version of
 * Windows that the BIOS has requested.
 */
```

И далее приводится массив с идентификаторами только для "Windows". Ни "Linux", ни "FreeBSD", ни "OpenVMS" в нём нет! Совершенно аналогичный код присутствует в ядре ОС Linux, так как в обоих этих ОС использована одна и та же «эталонная» реализация «ACPI Component Architecture» выполненная инженерами компании Intel. Да, той самой Intel, которая придумала ACPI и регулярно проталкивает новые спецификации этого стандарта!

Таким образом, вся эта «кухня» с подстройкой под альтернативные ОС с помощью **_OSI** не имеет ни малейшего значения, не смотря на то, что в реализации «Firmware» на многих машинах есть зависимый от этой фичи код. Интересно, знают ли разработчики OEM «Firmware» об этом ? И как они вообще тестировали свой код, если альтернативные пути (альтернативные ОС) всегда отвечают что они якобы представляют собой «Windows» ? ;-)

2.7.2. Метод _OS для запроса типа операционной системы

С самой первой версии ACPI среди служебных методов присутствовал метод **_OS**. Этот метод, аналогично методу **_OSI**, имплементируется внутри операционной системы, но он не принимает на вход никаких параметров, а только лишь возвращает фиксированную строку определяющую название и версию операционной системы. В операционной системе FreeBSD данный метод всегда выдает одну и ту же строку, давайте посмотрим какую именно:

```
rz@hp15s:~ % sudo acpi_call -p \_OS
Microsoft Windows NT
```

Неожиданно, правда ? Откровенно говоря, я тоже ожидал увидеть здесь что-то типа "FreeBSD" или "Linux", но нет. Опять же, подсмотрев в исходный код реализации подсистемы ACPICA внутри ядра FreeBSD, я обнаружил следующий комментарий в файле **sys/contrib/dev/acpica/include/acconfig.h**:

```
/*
 * OS name, used for the _OS object. The _OS object is essentially obsolete,
 * but there is a large base of ASL/AML code in existing machines that check
 * for the string below. The use of this string usually guarantees that
 * the ASL will execute down the most tested code path. Also, there is some
 * code that will not execute the _OSI method unless _OS matches the string
 * below. Therefore, change this string at your own risk.
 */
#define ACPI_OS_NAME                    "Microsoft Windows NT"
```

Как указано в комментарии, метод **_OS** давно и безвозвратно устарел, но многие OEM производители продолжают вставлять его в свой ASL/AML код с целью определить тип операционной системы на которой исполняется их «Firmware». В какой-то момент разработчики опенсорсных системы обнаружили, что несмотря на наличие возможности определения типа операционной системы, единственная ОС, поддержка которой действительно тестируется при разработке «Firmware», это ОС «Windows». Поэтому строку "Microsoft Windows NT" захардкодили и теперь она всегда возвращается при вызове метода **_OS**, во избежания исполнения непроверенного AML кода в «Firmware». На известном сайте ***kernel.org*** есть любопытная статья про использование _OS и _OSI, и о том, почему из ядер опенсорсных операционных систем была убрана поддержка **_OSI("Linux")** и прочих идентификаторов ОС.

Тем не менее, в операционной системе FreeBSD имеется способ передать в ACPICA произвольный тип ОС (произвольную строку с идентификатором). Говорят, что иногда это помогает «вправить мозги» AML коду «Firmware» и решить некоторые проблемы. Для этого в файл конфигурации загрузчика **/boot/loader.conf** необходимо добавить системную переменную **hw.acpi.osname**, присвоив ей символьную строку с идентификатором операционной системы, например:

```
hw.acpi.osname="Linux"
```

Но перед тем, как прибегать к такому способу решения проблемы, следует снять дамп, выполнить декомпиляцию «Firmware» и проанализировать получившийся ASL код на предмет использования метода **_OS**, и попытаться понять какие идентификаторы воспринимает код от OEM производителя. Я просмотрел ASL код полученный с ноутбука «HP Laptop 15s» и нашел в нём упоминание следующих ОС:

```
rz@hp15s:~/ACPI % grep -r _OS, *.dsl
dsdt.dsl:            If (MCTH (_OS, "Microsoft Windows NT"))
dsdt.dsl:            If (MCTH (_OS, "Microsoft Windows"))
dsdt.dsl:            If (MCTH (_OS, "Microsoft WindowsME: Millennium Edition"))
dsdt.dsl:            If (MCTH (_OS, "Linux"))
dsdt.dsl:            If (MCTH (_OS, "FreeBSD"))
dsdt.dsl:            If (MCTH (_OS, "HP-UX"))
dsdt.dsl:            If (MCTH (_OS, "OpenVMS"))
dsdt.dsl:            ElseIf (CondRefOf (\_OS, Local0))
```

```
dsdt.dsl:               If (SEQL (_OS, "Microsoft Windows"))
dsdt.dsl:               ElseIf (SEQL (_OS, "Microsoft WindowsME: Millennium Edition"))
dsdt.dsl:               ElseIf (SEQL (_OS, "Microsoft Windows NT"))
```

Разумеется я испробовал все варианты, в том числе "**FreeBSD**", "**Linux**" и различные варианты "**Windows ***", но это ни как не решило проблему с нерабочей кнопкой «Power».

3. Поиск причины «неисправности» кнопки Power

С целью локализовать проблему неработающей кнопки «Power» на ноутбуке HP15s я перепробовал несколько «наивных» фиксов предложенных в главе «11.16 Using and Debugging FreeBSD ACPI» руководства «FreeBSD Handbook», в том числе подмена версии ОС для методов _OS и _OSI на различные доступные варианты, исправление синтаксических ошибок в DSDT/SSDT таблицах, и т.д. — ничего из предложенного не сработало. Тогда я решил посмотреть что происходит в системе в момент нажатия кнопки «Power». Для этого я загрузил ОС в «многословный» и «однопользовательский» режим (команда загрузчику **boot -v -s** в меню «*3. Escape to loader prompt*»), после чего установил следующие значения системных переменных:

```
# Еще раз, для надежности, включить «многословность» для всех объектов ядра
root@:~ # sysctl debug.bootverbose=1
debug.bootverbose: 1 -> 1 # уже включено опцией boot -v

# Переводить систему в состояние «Power Off» при нажатии кнопки
root@:~ # sysctl hw.acpi.power_button_state=S5
hw.acpi.power_button_state: NONE -> S5

# Выдавать дополнительную информацию от ACPI
root@:~ # sysctl hw.acpi.verbose=1
hw.acpi.verbose: 0 -> 1
```

и несколько раз кратковременно нажал кнопку «Power». Ожидаемо ничего не произошло, в операционной системе нет никакой реакции на событие от этой кнопки. Возник вопрос: формируется ли вообще какое либо событие при нажатии на данную кнопку ?

Независимо от того, каким способом реализована доставка событий в подсистеме ACPI от кнопки «Power» на данной машине (через SMI, SCI, GPE или через _EVT), в операционной системе должно формироваться и обрабатываться прерывание по линии IRQ на которой находится аппаратура ACPI. Выяснить к какой линии IRQ подключена подсистема ACPI в данной машине можно из таблицы FADT (она же "FACP"). Воспользуемся полученной ранее декомпилированной версией таблицы FADT и сделаем в ней поиск поля с именем "Interrupt":

```
rz@hp15s:~/ACPI % grep Interrupt facp.dsl
[02Eh 0046 002h]               SCI Interrupt : 0009
```

Предположительно, ACPI находится на IRQ9. Этот же факт подтверждается из системных переменных:

```
root@:~ # sysctl -a | grep 'irq.*acpi'
...
irq9: acpi0:17 @cpu0(domain0): 5
```

Теперь запросим у операционной системы статистику (счетчик) по прерываниям для линии IRQ9, для чего воспользуемся системной утилитой **vmstat**:

```
root@:~ # vmstat -i | grep irq9
irq9: acpi0                            5          0
```

Счетчик IRQ9 зафиксирован на цифре 5. Сделаем несколько быстрых нажатий на кнопку «Power» и еще раз запросим счетчик обработанных прерываний для этой линии IRQ:

```
root@:~ # vmstat -i | grep irq9
irq9: acpi0                            5          0
```

И получим все то же значение (5), то есть счетчик не увеличивается. Но если подождать секунд 5-7, и еще раз запросить значение счетчика прерываний по линии IRQ9, то можно увидеть некоторое приращение на 1-2 единицы.

Я обратил внимание на то, что на этом ноутбуке счетчик увеличивается на единицу примерно каждые 4-5 секунд. Это говорит о том, что какие-то события в подсистем ACPI всё же проистекают, но это явно не события формируемые нажатием кнопки «Power». Но так ли это ?

3.1. Включение режима отладки для ACPICA

Вспомнив про то, что в подсистеме ACPI есть специальные регистры **PM1x_STS** и **PM1x_EN** в которых имеются биты отвечающие за передачу состояния кнопки «Power» (биты **PWRBTN_STS** и **PWRBTN_EN** соответственно), возникает естественное желание подсмотреть их состояние в разные моменты времени. Возможно, что по какому-то стечению обстоятельств бит разрешения (**PWRBTN_EN**) не установлен или ошибочно сбрасывается системой, соответственно события от кнопки «Power» маскированы (не формируются), а значит и нет прерываний по линии IRQ9 в момент её нажатия.

На эту мысль меня натолкнуло прочтение статьи «Power Button Resurrection on the Surface Go 2» от пользователя *Pascal Brückner* (*sylencecc*), в ней автор боролся и успешно победил аналогичную проблему на планшете «Surface Go 2» под ОС Linux. Статья, кстати, очень занимательная — автор копнул достаточно глубоко в исходный код ACPICA в составе Linux. Ему удалось локализовать проблему и найти простое решение, которое состояло, в том чтобы изменить состояние конфигурационного бита «**Control Method Power Button**» с лог «1» на лог «0» в таблице FADT, собрать кастомную версию таблицы и подсунуть её ОС Linux при загрузке. Мне наивно подумалось, что проблема с кнопкой «Power» на ноутбуке «HP Laptop 15s» может иметь если не точно такое же, то аналогичное решение. К тому же, заглянув в FADT от HP15s, я обнаружил что данный конфигурационный бит также установлен в «1», то есть ACPI подсистемой на данной машине используется AML код для управления кнопкой «Power», но в чем конкретно это выражается, на данном этапе, для меня было не ясно. Может быть AML код просто «сломан» ?

Просмотрев исходный код ACPICA в составе ядра ОС FreeBSD я обнаружил следующие строки выдающие отладочную информацию с состоянием битов в группах регистров **PM1x_STS** и **PM1x_EN**:

```
rz@butterfly:~/FreeBSD-src/src-15.x % grep -B 4 -A 9 ACPI_REGISTER_PM1_STATUS
sys/contrib/dev/acpica/components/events/evevent.c
    /*
     * Read the fixed feature status and enable registers, as all the cases
     * depend on their values. Ignore errors here.
     */
    Status = AcpiHwRegisterRead (ACPI_REGISTER_PM1_STATUS, &FixedStatus);
    Status |= AcpiHwRegisterRead (ACPI_REGISTER_PM1_ENABLE, &FixedEnable);
    if (ACPI_FAILURE (Status))
    {
        return (IntStatus);
    }

    ACPI_DEBUG_PRINT ((ACPI_DB_INTERRUPTS,
        "Fixed Event Block: Enable %08X Status %08X\n",
        FixedEnable, FixedStatus));
```

После прочтения страницы руководства **man 4 acpi** выяснилось, что для того, чтобы задействовать макро **ACPI_DEBUG_PRINT**, придется пересобрать ядро операционной системы из исходных кодов с включенными опциями **options DDB** и **options ACPI_DEBUG**. Но перед этим придется разобраться с тем, как установить исходные коды операционной системы и как осуществлять сборку кастомного ядра ОС FreeBSD.

3.2. Установка исходных кодов ОС FreeBSD 15.1-RELEASE

Имеется несколько способов установить исходные коды операционной системы FreeBSD на локальный диск. На мой взгляд самый простой и самый лучший способ это клонировать официальный репозиторий с помощью утилиты **git**. Если эта утилита еще не установлена в системе, то добавить её из репозитория пакетов можно с помощью команды **pkg install**:

```
rz@hp15s:~ % sudo pkg install -y git
```

После чего создать отдельный рабочий подкаталог:

```
rz@hp15s:~ % mkdir FreeBSD-src

rz@hp15s:~ % cd FreeBSD-src
```

И клонировать в него репозиторий с исходниками FreeBSD версии 15.1-RELEASE с ограничением истории в 1000 последних коммитов, дабы уменьшить объем скачиваемой информации и ускорить процесс (весь объем коммитов этого репозитория огромен и никогда нам не потребуется):

```
rz@hp15s:~/FreeBSD-src % git clone --depth=1000 --branch release/15.1.0
https://git.freebsd.org/src.git src-15.x

Cloning into 'src-15.x'...
```

По завершению исполнения этой команды исходные коды всей системы, включая ядро, будут располагаться в подкаталоге **src-15.x**:

```
rz@hp15s:~/FreeBSD-src % cd src-15.x

rz@hp15s:~/FreeBSD-src/src-15.x % ls -l
total 1092
-rw-r--r--    1 rz rz  11935 Jul 27 03:49 CONTRIBUTING.md
-rw-r--r--    1 rz rz   6070 Jul 27 03:49 COPYRIGHT
-rw-r--r--    1 rz rz    359 Jul 27 03:49 LOCKS
-rw-r--r--    1 rz rz   6794 Jul 27 03:49 MAINTAINERS
-rw-r--r--    1 rz rz  31273 Jul 27 03:49 Makefile
-rw-r--r--    1 rz rz 144552 Jul 27 03:49 Makefile.inc1
-rw-r--r--    1 rz rz   3986 Jul 27 03:49 Makefile.libcompat
-rw-r--r--    1 rz rz   1939 Jul 27 03:49 Makefile.sys.inc
-rw-r--r--    1 rz rz 613031 Jul 27 03:49 ObsoleteFiles.inc
-rw-r--r--    1 rz rz   3165 Jul 27 03:49 README.md
-rw-r--r--    1 rz rz  19119 Jul 27 03:49 RELNOTES
-rw-r--r--    1 rz rz 111192 Jul 27 03:49 UPDATING
drwxr-xr-x   44 rz rz   1024 Jul 27 03:49 bin
drwxr-xr-x   11 rz rz    512 Jul 27 03:49 cddl
drwxr-xr-x   94 rz rz   2048 Jul 27 03:49 contrib
drwxr-xr-x    7 rz rz    512 Jul 27 03:49 crypto
drwxr-xr-x    9 rz rz    512 Jul 27 03:49 etc
drwxr-xr-x    5 rz rz    512 Jul 27 03:49 gnu
drwxr-xr-x   11 rz rz   2560 Jul 27 03:49 include
drwxr-xr-x    8 rz rz    512 Jul 27 03:49 kerberos5
drwxr-xr-x    9 rz rz    512 Jul 27 03:49 krb5
drwxr-xr-x  156 rz rz   3072 Jul 27 03:49 lib
drwxr-xr-x   37 rz rz   1024 Jul 27 03:49 libexec
drwxr-xr-x   12 rz rz    512 Jul 27 03:49 release
```

```
drwxr-xr-x    4 rz rz    512 Jul 27 03:49 rescue
drwxr-xr-x   86 rz rz   2048 Jul 27 03:49 sbin
drwxr-xr-x    8 rz rz    512 Jul 27 03:49 secure
drwxr-xr-x   32 rz rz   1024 Jul 27 03:49 share
drwxr-xr-x   26 rz rz   1024 Jul 27 03:49 stand
drwxr-xr-x   53 rz rz   1024 Jul 27 03:49 sys
drwxr-xr-x    3 rz rz    512 Jul 27 03:49 targets
drwxr-xr-x   10 rz rz    512 Jul 27 03:49 tests
drwxr-xr-x   19 rz rz    512 Jul 27 03:49 tools
drwxr-xr-x  276 rz rz   4608 Jul 27 03:49 usr.bin
drwxr-xr-x  231 rz rz   4608 Jul 27 03:49 usr.sbin
```

Дерево подкаталогов исходных кодов FreeBSD имеет сложную структуру. В двух словах её можно описать так:

- Все, что относится к ядру ОС располагается в подкаталоге **./sys/**.
- Исходные коды системных утили располагаются соответственно в подкаталогах **./bin/**, **./sbin/**, **./usr.bin/** и **./usr.sbin/**.
- В подкаталогах **./contrib/** и **./sys/contrib/** располагаются исходные коды т. н. контрибуций — кода который был заимствован от сторонних разработчиков и внесен в проект FreeBSD.
- Остальные подкаталоги содержат различные конфигурационные файлы и вспомогательные утилиты для выполнения сборки.

В подкаталоге **./sys/** есть свой **./sys/contrib/** — в нём расположены исходные коды для проектов заимствованных у других разработчиков и необходимых для сборки ядра ОС. Компонент ACPICA, предоставленный компанией Intel, расположен в **./sys/contrib/dev/acpica/**.

3.3. Сборка отладочного ядра ОС FreeBSD

Когда-то давно сборка настроенного под свои нужды ядра операционной системы являлась неотъемлемой частью начала пути каждого ~~самурая~~ пользователя UNIX систем. Вызвано это было необходимостью исключить их ядра неиспользуемый код (неиспользуемые драйверы, дополнительные отладочные сообщения, иной неиспользуемый функционал), и добавления кода требуемого для поддержки специфической аппаратуры установленной в машине пользователя. Со временем этот навык был пользователями утерян и сейчас применяется только системными программистами занимающимися разработкой самой ОС, да и то в исключительных случаях. Похоже на то, что наш случай очень близок именно к такому.

Сборка ядра ОС FreeBSD, на первый взгляд сложная задача, на самом деле запускается двумя командами: **make cleankernel** и **make buildkernel**. Но перед запуском процесса сборки, нам потребуется создать конфигурационный файл для будущего ядра. Этот файл обычно располагается в подкаталоге **./sys/amd64/conf/** в дереве каталогов исходных кодов ядра, он содержит перечень драйверов (модулей) которые требуется включить в состав ядра и их различные опции. Так, например, подсистема ACPICA представлена в ядре ОС FreeBSD драйвером **acpi**, который включается опцией "**device acpi**". По умолчанию, в составе исходных кодов ОС FreeBSD уже содержится универсальный конфигурационный файл для получения ядра, которое копируется в систему пользователя при инсталляции ОС. Это ядро имеет название **GENERIC**, а конфигурационный файл соответственно расположен в **./sys/amd64/conf/GENERIC**. Всё, что нам необходимо сделать — скопировать этот файл в новый файл с другим именем и слегка отредактировать его.

Так, как исходные коды ядра предоставлены нам в виде Git репозитория, то перед тем как вносить какие-либо изменения в структуру кода, создадим отдельную ветку (branch) для наших изысканий, это позволит в любой момент откатиться к исходному варианту или произвести сравнение оригинального кода с внесенными нами изменениями (т. е. выполнить **git diff**). Еще одно полезное свойство создания отдельных веток для каждого эксперимента состоит в том, что при сборки ядра имя ветки репозитория будет внесено в версию ядра вместе с идентификатором последнего коммита. Это очень полезно для того, чтобы после загрузки нового ядра ОС мы могли легко убедиться в том, какое именно ядро загружено в данный момент времени — когда количество тестовых ядер станет больше одного, вопрос: «а то ли ядро сейчас загружено?» будет актуален как никогда.

Создать ветку и сразу войти в режим внесения изменений (checkout) можно одной командой:

```
rz@hp15s:~/FreeBSD-src/src-15.x % git checkout -b acpi_test
Switched to a new branch 'acpi_test'
```

Как видно, Git создал нам новую ветку "**acpi_test**" и перешел в неё. Сейчас можно смело вносить изменения в код не боясь что-то испортить. Назовем наше кастомное ядро **CUSTOM** и соответственно создадим конфигурационный файл с таким же именем:

```
rz@hp15s:~/FreeBSD-src/src-15.x % cp ./sys/amd64/conf/GENERIC ./sys/amd64/conf/CUSTOM
```

Внесем в конфигурационный файл требуемые нам опции:

```
rz@hp15s:~/FreeBSD-src/src-15.x % echo 'options DDB' >> ./sys/amd64/conf/CUSTOM

rz@hp15s:~/FreeBSD-src/src-15.x % echo 'options ACPI_DEBUG' >> ./sys/amd64/conf/CUSTOM

rz@hp15s:~/FreeBSD-src/src-15.x % sed -I -e s/GENERIC/CUSTOM/ ./sys/amd64/conf/CUSTOM
```

Последняя (третья) команда редактирует конфигурационный файл и заменяет в нем имя ядра **GENERIC** на **CUSTOM**.

Теперь можно приступать к сборке ядра. Вызовем утилиту **make** указав ей в качестве целей сначала **cleankernel** для очистки от предыдущей сборки (это на всякий случай), а потом **buildkernel** — для запуска сборки нового ядра. Так же укажем с помощью переменной **OBJROOT** путь к подкаталогу в который будут складываться временные объектные файлы, а с помощью переменной **KERNCONF** — имя конфигурационного файла собираемого ядра:

```
# Удалить результат предыдущей сборки ядра
rz@hp15s:~/FreeBSD-src/src-15.x % make cleankernel OBJROOT=$PWD/obj/

# Запустить новую сборку ядра CUSTOM
rz@hp15s:~/FreeBSD-src/src-15.x % make -j12 buildkernel KERNCONF=CUSTOM OBJROOT=$PWD/obj/

...

--- kernel.full ---
linking kernel.full
ctfmerge -L VERSION -g -o kernel.full ...
      text      data       bss        dec         hex   filename
  20675466   1673041   8814208   31162715   0x1db815b   kernel.full
--- kernel.debug ---
objcopy --only-keep-debug kernel.full kernel.debug
--- kernel ---
objcopy --strip-debug --add-gnu-debuglink=kernel.debug  kernel.full kernel
      336.61 real      3134.65 user       217.75 sys
-------------------------------------------------------------
>>> Kernel build for GENERIC completed on Tue Jul 28 03:27:14 +05 2026
-------------------------------------------------------------
```

```
>>> Kernel(s)  GENERIC built in 337 seconds, ncpu: 12, make -j12
----------------------------------------------------------------
```

Важно: для сборки ядра ОС FreeBSD 15.1-RELEASE потребуется около 3 ГБ свободного дискового пространства! Процесс достаточно долгий и зависит от ресурсов вашей рабочей машины. Как видно из приведенной выше статистики, на моём ноутбуке «Lenovo Ideapad 3» с процессором Ryzen 5 (12 потоков) и 32 ГБ ОЗУ, процесс сборки ядра занимает около 6 минут (337 seconds).

Параметр **OBJROOT=$PWD/obj/** задает путь к подкаталогу в который будут складываться объектные файлы (в том числе и собранное ядро) в процессе выполнения сборки. В данном случае все объектники будут складываться в подкаталог **./obj/** в текущем каталоге. Если параметр **OBJROOT** не указывать, то сборочный скрипт попытается складывать временные файлы в системный подкаталог **/usr/src/obj/**, что обычно доступно только пользователю **root**.

После того, как у нас готово новое ядро, его необходимо установить и загрузить систему с его участием. Но перед этим я настоятельно рекомендую на целевой машине (то есть на той, для которой собирается ядро) сделать копию старой версии ядра для того, чтобы иметь возможность быстро вернуть систему к предыдущему состоянию. Всё компоненты ядра ОС FreeBSD: само ядро, модули ядра, системный загрузчик (**loader**) и его скрипты находятся в подкаталоге **/boot** расположенном в самом корне файловой системы. Поэтому достаточно просто сделать копию этого подкаталога, например вот так:

```
rz@hp15s:~/FreeBSD-src/src-15.x % sudo cp -r /boot /boot.good
```

Теперь можно смело инсталлировать новое, только что собранное, ядро системы не боясь что-то испортить:

```
rz@hp15s:~/FreeBSD-src/src-15.x % sudo make installkernel KERNCONF=CUSTOM OBJROOT=$PWD/obj/
```

В виду особенностей скрипта установщика (он будет пытаться изменять принадлежность файлов и подкаталогов), команду установки придется выполнить от имени суперпользователя!

Файл с новым ядром ОС будет располагаться в **/boot/kernel/kernel**:

```
rz@hp15s:~/FreeBSD-src/src-15.x % file /boot/kernel/kernel
/boot/kernel/kernel: ELF 64-bit LSB executable, x86-64, version 1 (FreeBSD), dynamically
linked, interpreter /red/herring, BuildID[sha1]=78f473b6c191251a9d47422756891152fc4ec15d,
not stripped

rz@hp15s:~/FreeBSD-src/src-15.x % ls -l /boot/kernel/kernel
-r--r--r--  1 root wheel 31247440 Jul 29 04:20 /boot/kernel/kernel
```

Если Вы надумаете делать сборку ядра на другой (более быстрой) машине, как это делаю я, то для переноса нового ядра необходимо сначала выполнить его установку во временный каталог на рабоче машине, а потом перенести всё содержимое этого каталога на целевую машину, при этом указав путь к временном каталогу с помощью переменной **DESTDIR**:

```
rz@butterfly:~/FreeBSD-src/src-15.x % sudo make installkernel KERNCONF=CUSTOM
OBJROOT=$PWD/obj/ DESTDIR=$PWD/install/
```

```
# Результат установки (ядро и его модули) будут в помещены в ./install/boot
rz@butterfly:~/FreeBSD-src/src-15.x % du -d ./install/boot
5783016 .

# Копировать ./install/boot на целевую машину с помощью scp
rz@butterfly:~/FreeBSD-src/src-15.x % scp -rp ./install/boot rz@hp15s:/
```

На целевой машине выполнить:

```
#Копировать новое ядро на целевой машине в корень файловой системы:
rz@hp15s:~ % sudo cp -r ~/boot /
```

После установки нового (кастомного) ядра, необходимо перезагрузить операционную систему в однопользовательский режим, то есть после команды **reboot** в меню загрузчика нажать «**S**».

Если что-то пошло не так, что старое ядро можно загрузить указав к нему путь в промпте загрузчика (меню «*3. Escape to loader prompt*»):

```
OK boot /boot.good/kernel/kernel
```

3.4. Получение отладочной информации от ACPICA

Теперь можно приступить к «трассировке» ACPICA чтобы выяснить состояние битов в регистрах **PM1x_STS** и **PM1x_EN**, но перед этим необходимо проделать еще пару операций.

Во-первых, необходимо убедиться в том, что мы загрузились именно с тем ядром, сборку которого выполняли:

```
root@:~ # uname -a
FreeBSD  15.1-RELEASE FreeBSD 15.1-RELEASE #3 acpi_test-96841ea08dcf-dirty: Tue Aug  4 03:02:06 +05 2026
rz@megamini:/usr/obj/usr/src/amd64.amd64/sys/CUSTOM amd64
```

Система сообщает нам, что текущее (загруженное) ядро имеет номер версии сборки **acpi_test-96841ea08dcf-dirty** — это как раз то, что требуется, так как в нём присутствует строка "acpi_test" соответствующая названию «ветки» исходного кода.

Далее проверим присутствует ли в собранном нами кастомном ядре код для вывода отладочных сообщений компонента ACPICA, то есть убедимся что опция конфигурации ядра **ACPI_DEBUG** возымела действие. Сделать это можно путем поиска части строки с сообщением в бинарном файле ядра. Для этого на целевой машине выполним команду поиска:

```
root@:~ # grep -r 'Fixed Event Block: Enable %08X Status %08X' /boot/
Binary file /boot/kernel/kernel matches
```

Вывод команды **grep** говорит о том, что такая строка символов действительно присутствует внутри бинарного файла ядра. Если **grep** не выдает никакого сообщение, значит такой строки не найдено, а следовательно ядро собранно без опции **ACPI_DEBUG**. В этом случае необходимо выяснить на каком этапе закралась ошибка, исправить её и еще раз пересобрать и установить ядро ОС на целевую машину.

Код подсистемы ACPICA имеет ряд механизмов для регулирования потока отладочных сообщений. Чтобы разрешить системе выводить отладочные сообщения от ACPICA в системный буфер (**dmesg**), необходимо указать от какого из компонентов (или их группы) требуется получать сообщения, а также сообщения какого уровня: информационные, отладочные, сообщения об ошибках и т. д. Информацию о перечне доступных компонентов и уровней можно почерпнуть из того же системного руководства (команда **man 4 acpi**). После прочтения этой странички руководства, а также после ряда экспериментов, я выяснил, что для включения вывода отладочных сообщений от компонента **ACPI_EVENTS**, в котором располагается обработчик прерываний, нам потребуется установить две системных переменных **debug.acpi.layer** и **debug.acpi.level** в следующие значения:

```
root@:~ # sysctl debug.acpi.layer=ACPI_EVENTS
debug.acpi.layer: ACPI_UTILITIES ACPI_HARDWARE ACPI_EVENTS ACPI_TABLES ACPI_NAMESPACE
ACPI_PARSER ACPI_DISPATCHER ACPI_EXECUTER ACPI_RESOURCES AC -> ACPI_EVENTS

root@:~ # sysctl debug.acpi.level=ACPI_LV_VERBOSITY3
debug.acpi.level: NONE -> ACPI_LV_INIT ACPI_LV_DEBUG_OBJECT ACPI_LV_INFO ACPI_LV_REPAIR
ACPI_LV_ALL_EXCEPTIONS ACPI_LV_PARSE ACPI_LV_DISPATCH ACPI_LV_EXE
```

С этого момента на системную консоль могут начать выводиться объемные сообщения от ACPICA генерируемые в процессе обработки событий, так что будьте готовы! Отключить вывод можно командой:

```
root@:~ # sysctl debug.acpi.level=NONE
```

На ноутбуке HP15s, сразу после включения вывода отладочных сообщений, я ничего не увидел. Я тут же несколько раз нажал кнопку «Power» и также не получил ни одного сообщения на системной консоле. Но спустя секунд 5 на консоле появились следующее сообщение:

Листинг 6. Отладочные сообщения от компонента ACPI_EVENT.

```
    evsci-0243 EvSciXruptHandler     : ----Entry
  evevent-0365 EvFixedEventDetect    : Fixed Event Block: Enable 00004120 Status 00000111
    evgpe-0846 EvDetectGpe           : ----Entry
    evgpe-0886 EvDetectGpe           : Read registers for GPE 00: Status=00, Enable=00,
RunEnable=08, WakeEnable=00
    evgpe-0846 EvDetectGpe           : ----Entry
    evgpe-0886 EvDetectGpe           : Read registers for GPE 01: Status=00, Enable=00,
RunEnable=08, WakeEnable=00
    evgpe-0846 EvDetectGpe           : ----Entry
    evgpe-0886 EvDetectGpe           : Read registers for GPE 02: Status=00, Enable=00,
RunEnable=08, WakeEnable=00
    evgpe-0846 EvDetectGpe           : ----Entry
    evgpe-0886 EvDetectGpe           : Read registers for GPE 03: Status=00, Enable=08,
RunEnable=08, WakeEnable=00
    evgpe-0846 EvDetectGpe           : ----Entry
    evgpe-0886 EvDetectGpe           : Read registers for GPE 04: Status=00, Enable=00,
RunEnable=08, WakeEnable=00
    evgpe-0846 EvDetectGpe           : ----Entry
    evgpe-0886 EvDetectGpe           : Read registers for GPE 05: Status=00, Enable=00,
RunEnable=08, WakeEnable=00
    evgpe-0846 EvDetectGpe           : ----Entry
    evgpe-0886 EvDetectGpe           : Read registers for GPE 06: Status=00, Enable=00,
RunEnable=08, WakeEnable=00
    evgpe-0846 EvDetectGpe           : ----Entry
    evgpe-0886 EvDetectGpe           : Read registers for GPE 07: Status=00, Enable=00,
RunEnable=08, WakeEnable=00
```

```
    evgpe-0595 EvGpeDetect           : Ignore disabled registers for GPE 08-0F:
RunEnable=00, WakeEnable=00
    evgpe-0595 EvGpeDetect           : Ignore disabled registers for GPE 10-17:
RunEnable=00, WakeEnable=00
    evgpe-0846 EvDetectGpe           : ----Entry
    evgpe-0886 EvDetectGpe           : Read registers for GPE 18: Status=00, Enable=00,
RunEnable=40, WakeEnable=40
    evgpe-0846 EvDetectGpe           : ----Entry
    evgpe-0886 EvDetectGpe           : Read registers for GPE 19: Status=00, Enable=00,
RunEnable=40, WakeEnable=40
    evgpe-0846 EvDetectGpe           : ----Entry
    evgpe-0886 EvDetectGpe           : Read registers for GPE 1A: Status=00, Enable=00,
RunEnable=40, WakeEnable=40
    evgpe-0846 EvDetectGpe           : ----Entry
    evgpe-0886 EvDetectGpe           : Read registers for GPE 1B: Status=00, Enable=00,
RunEnable=40, WakeEnable=40
    evgpe-0846 EvDetectGpe           : ----Entry
    evgpe-0886 EvDetectGpe           : Read registers for GPE 1C: Status=00, Enable=00,
RunEnable=40, WakeEnable=40
    evgpe-0846 EvDetectGpe           : ----Entry
    evgpe-0886 EvDetectGpe           : Read registers for GPE 1D: Status=00, Enable=00,
RunEnable=40, WakeEnable=40
    evgpe-0846 EvDetectGpe           : ----Entry
    evgpe-0886 EvDetectGpe           : Read registers for GPE 1E: Status=00, Enable=40,
RunEnable=40, WakeEnable=40
    evgpe-0846 EvDetectGpe           : ----Entry
    evgpe-0886 EvDetectGpe           : Read registers for GPE 1F: Status=00, Enable=00,
RunEnable=40, WakeEnable=40
    evsci-0268 EvSciXruptHandler     : ----Exit- 0000000000000001
```

В этом сообщении просматривается строка с состоянием битов в регистрах **PM1x_STS** и **PM1x_EN**, вот как она выглядит:

```
evevent-0365 EvFixedEventDetect    : Fixed Event Block: Enable 00000101 Status 00000001
```

Я обратил внимание на то, что такие большие отладочные сообщения от компонента EVENTS подсистемы ACPICA на данном ноутбуке появляются с периодичностью примерно в 5 сек. Из состояния бита 0 регистра статуса можно догадаться, что происходит периодическое срабатывание таймера **PM_TIMER**, который на данной машине имеет разрядность 24 бита, что при частоте 3,579545 МГц дает срабатывание прерывания по переполнению примерно раз в 4,66 сек. Собственно это событие мы и наблюдаем.

Но вернемся к нашей проблеме. Не смотря на то, что бит **PWRBTN_EN** (бит номер 8) в регистре **PM1x_EN** установлен в единицу, бит **PWRBTN_STS** (так же имеющий номер 8) в регистре статуса **PM1x_STS** никогда не появляется в состоянии «1». То есть прерывания от кнопки «Power» разрешены, но сигнала по этой линии не приходит! Похоже, что кнопка «Power» вообще не привязана к ACPI на данной машине и обрабатывается отдельно, как обычное GPIO. Косвенным подтверждением этой теории является то, что в таблице FADT установлен флаг:

```
Control Method Power Button (V1) : 1
```

Что согласно спецификации ACPI говорит о том, что событие PWRB формируется программно внутри AML кода, а источником для его возникновения должен быть какой-то

компонент операционной системы вызывающий один из AML методов. Чтобы проверить эту теорию, нам придется заглянуть в дисассемблированный код таблиц DSDT/SSDT.

3.5. Анализ кода ПО ACPI и поиск причины отсутствия события PWRB

Нам известно, что при изменении статуса кнопки «Power», AML код обязан отправить в операционную систему нотификационное событие **_SB.PWRB**, которое воспринимается операционной системой как сигнал к действию. Также нам известно, что изменение состояния кнопки «Power» кодируется числом 0x80 - «Device Status Changed» (см. главу «*1.7.3. Оператор Notify() для отправки нотификаций в OSPM*»). Таким образом нам надо найти все участки кода которые имеют вызов оператора **Notify(PWRB, 0x80)** и попытаться понять их логику работы. Как выяснилось, таких участков кода всего два. Один из них находится в таблице DSDT:

Листинг 7а. Первый участок ASL кода формирующий нотификацию для PWRB (DSDT).

```
    Scope (_SB.PCI0.SBRG)
    {
        Device (EC0)
        {
            Name (_HID, EisaId ("PNP0C09") /* Embedded Controller Device */)  // _HID:
Hardware ID
            Name (_UID, One)  // _UID: Unique ID

            ...

            Method (_Q28, 0, NotSerialized)  // _Qxx: EC Query, xx=0x00-0xFF
            {
                Notify (PWRB, 0x80) // Status Change
            }
            ...
        }
    }
```

Второй участок находится в таблице SSDT14:

Листинг 7б. Второй участок ASL кода формирующий нотификацию для PWRB (SSDT14).

```
    Scope (\_SB.GPIO)
    {
        Method (_EVT, 1, Serialized)  // _EVT: Event
        {
            Switch (ToInteger (Arg0))
            {
                Case (Zero)
                {
                    MSTP (0x3900)
                    Notify (\_SB.PWRB, 0x80) // Status Change
                }
                ...
            }
        }
     }
```

3.5.1. Обработка событий от Embedded Controller (метод EC0._Qxx)

Попробуем разобраться для чего нужен первый фрагмент кода. В нём объявляется устройство EC0 представляющее собой первый (и единственный) «Embedded Controller», а к данному устройству — несколько методов вида **_Qxx**, где xx — шестнадцатеричный код. Если посмотреть код вокруг, то можно догадаться, что набор этих методов есть не что иное как обработка пронумерованных запросов от EC контроллера. Запрос с номером 0x28 (**_Q28**) формирует требуемое нам нотификационное событие информирующее операционную систему о смене состояния (нажатии) кнопки «Power». Но кто и когда вызывается эти **_Qxx** методы ? Логично предположить, что этим «заведует» драйвер EC контроллера исходный код которого расположен в файле **sys/dev/acpica/acpi_ec.c**.

Если заглянуть в текст драйвера, то можно увидеть, что он регистрирует в подсистеме ACPICA обработчик GPE события. Напомню, что «General Purpose Events» (GPEs) — это одни из способов формирования событий от ACPI аппаратуры (который работает одновременно с другим способом — «Fixed Feature», привязанного к битам в **PM1x_STS/EN** регистрах). И действительно, в выводе **dmesg** на ноутбуке HP15s имеется следующая строка:

```
acpi_ec0: <Embedded Controller: GPE 0x3> port 0x62,0x66 on acpi0
```

сообщающая нам о том, что события от EC0 будут поступать через GPE регистр номер 3. Сам обработчик GPE устроен достаточно просто: он отправляет команду **EC_COMMAND_QUERY** в EC контроллер чтобы выяснить какой номер метода требуется вызвать, после чего использует полученное данное как индекс для вызова метода **_Qxx**. Код вызова ACPI метода находится в функции **EcGpeQueryHandlerSub()** драйвера **acpi_ec** и выглядит следующим образом:

Листинг 8. Фрагмент кода функции EcGpeQueryHandlerSub() вызывающий метод _Qxx.

```
/*
 * Send a query command to the EC to find out which _Qxx call it
 * wants to make.  This command clears the SCI bit and also the
 * interrupt source since we are edge-triggered.  To prevent the GPE
 * that may arise from running the query from causing another query
 * to be queued, we clear the pending flag only after running it.
 */
for (retry = 0; retry < 2; retry++) {
    Status = EcCommand(sc, EC_COMMAND_QUERY);
    if (ACPI_SUCCESS(Status))
        break;
    if (ACPI_FAILURE(EcCheckStatus(sc, "retr_check",
        EC_EVENT_INPUT_BUFFER_EMPTY)))
        break;
}

...

Data = EC_GET_DATA(sc);

...

/* Evaluate _Qxx to respond to the controller. */
snprintf(qxx, sizeof(qxx), "_Q%02X", Data);
AcpiUtStrupr(qxx);
Status = AcpiEvaluateObject(sc->ec_handle, qxx, NULL, NULL);
if (ACPI_FAILURE(Status) && Status != AE_NOT_FOUND) {
    device_printf(sc->ec_dev, "evaluation of query method %s failed: %s\n",
        qxx, AcpiFormatException(Status));
}
```

Иными словами, схема передачи события от контроллера EC0 в AML код следующая:

1. Аппаратура EC контроллера устанавливает бит готовности для регистра GPE с номером 0x03, что приводит к срабатыванию аппаратного прерывания SCI (линия IRQ9 в нашем случае).
2. Обработчик прерывания по линии IRQ9 перебирает флаги статуса в GPE регистрах и вызывает соответствующие зарегистрированные обработчики событий.
3. Драйвер контроллера, предварительно зарегистрировав свой обработчик событий **EcGpeQueryHandler()** на GPE 0x3, производит прямой запрос в EC контроллер (отправляет команду **EC_COMMAND_QUERY**) с помощью которого выясняет номер **_Qxx** запроса.
4. Получив ответ от EC контроллера, драйвер производит вызов ACPI метода **_SB.PCI0.SBRG.EC0._Qxx**, где xx — есть данное полученное от EC контроллера.
5. Вызванная таким образом AML программа выполняет функцию ассоциированную с полученным запросом.

С тем, кто и как вызывает методы **_SB.PCI0.SBRG.EC0._Qxx** мы разобрались. Теперь нужно понять что скрывается именно за запросом номер 0x28 контроллера EC. К сожалению, мне не удалось найти четкого описания Q-кодов EC контроллера установленного в этом ноутбуке, но поиск в Сети навел меня на мысль о том, что это «запасной» (или даже аварийный) механизм выключения питания машины управляемый от EC контроллера. Но это не точно! В любом случае, из отладочных сообщений полученных от компонента EVENTS подсистемы ACPICA не прослеживается ни одного GPE 0x03 события в моменты до и после нескольких нажатий кнопки «Power», о чем говорит следующая строка:

```
evgpe-0886 EvDetectGpe          : Read registers for GPE 03: Status=00, Enable=08,
```

Т.е. формирование событий по GPE 0x03 разрешено, но статус всегда нулевой и события по кнопке «Power» отсутствуют.

Чтобы убедиться в том, что EC контроллер функционирует и способен формировать GPE события, достаточно отключить от ноутбука (и подключить обратно) шнур блока питания AC/DC. При этом, на консоле в этот же момент появятся сообщения содержащие строки вида:

```
    evgpe-0886 EvDetectGpe          : Read registers for GPE 03: Status=08, Enable=08,
RunEnable=08, WakeEnable=00
    evgpe-0974 EvGpeDispatch        : ----Entry
    evgpe-1067 EvGpeDispatch        : ----Exit- 0000000000000001
    evgpe-0846 EvDetectGpe          : ----Entry
```

указывающие на то, что события от GPE 0x03 поступают и для них вызывается функция **EvGpeDispatch()**.

3.5.2. Обработка событий от GPIO (метод GPIO._EVT)

Теперь разберемся со вторым участком кода. Не сложно догадаться, что он реализует задокументированный в спецификации ACPI метод **_SB.GPIO._EVT**, назначение которого описано в главе «*1.4. Поддержка событий через GPIO*». Напомню, что данный метод предназначен для трансляции состояния аппаратных линий GPIO в AML код, где и происходит их обработка. Метод **_EVT** принимает один целочисленный параметр который

рассматривается как номер логической линии GPIO контроллера, состояние которой претерпело изменение.

Из приведенного в листинге кода видно, что если параметр равен нулю, то происходит вызов оператора **Notify (_SB.PWRB, 0x80)** для передачи нотификации в операционную систему об изменении состояния кнопки «Power». Отсюда можно сделать предположение, что на ноутбуке HP15s кнопка «Power» каким-то образом связана с линией **GPIO0**. Попытаемся проверить эту теорию, попутно выясним какой компонент ядра ОС вызывает ACPI метод **_SB.GPIO._EVT** и в какой момент.

Если сейчас перезагрузить операционную систему в нормальный (многопользовательский) режим, сделать поиск по списку зарегистрированных в системе устройств и выбрать из него все устройства в названии которых присутствует строка "gpio", то получим следующее:

```
rz@hp15s:~ % devinfo -v | grep -i gpio
    unknown pnpinfo _HID=AMDI0030 _UID=0 _CID=AMDI0030 at handle=\_SB_.GPIO
```

Команда **devinfo** сообщает нам, что в вычислительной системе присутствует аппаратура с HID идентификатором **AMDI0030**, но драйвера к ней не зарегистрировано. Тут же видно, что данная аппаратура должна быть связана драйвером с объектом **_SB_.GPIO** в подсистеме ACPICA. Быстрый поиск в Сети показывает что данное устройство является GPIO контроллером на машинах с процессорами **AMD Ryzen**, на котором основывается данный ноутбук. Но почему к этому контроллеру не загружен драйвер ? И присутствуете ли таковой в ОС FreeBSD 15.x ?

На странице руководства вызываемого командой **man 4 gpio** можно прочитать подробности про устройство GPIO фреймворка внутри операционной системы FreeBSD. В том числе на этой странице руководства приводится информация о поддерживаемых GPIO контроллерах от различных производителей, однако, никакого упоминания про GPIO контроллеры от AMD, на момент написания этой статьи, в руководстве нет. Этот факт наводит на печальный мысли о том, что в ОС FreeBSD 15.x не поддерживается данный GPIO контроллер, что собственно и является первопричиной нерабочей кнопки «Power». Во всяком случае, в том «дефолтном» состоянии, в котором сейчас загружена ОС на данной машине, не загружено программного компонента способного считывать статус линий GPIO и передавать его в подсистему ACPICA для дальнейшей обработки и формирования системных событий. А это Беда!

3.6. Драйвер GPIO контроллера для процессоров AMD (amdgpio)

Озадаченный таким поворотом судьбы я принялся изучать посты на профильном форуме по ОС FreeBSD, и быстро выяснил, что драйвер для GPIO контроллера для процессоров AMD всё же в ОС FreeBSD имеется. Называется он **amdgpio**. Почему информация о нём отсутствует в системном руководстве, осталось мной невыясненным. Попробуем воспользоваться новым знанием и подключить этот драйвер.

Для чистоты эксперимента загрузим операционную систему в «однопользовательский» режим (кнопка «S» в загрузчике), после чего проверим имеется ли в подкаталоге **/boot** модуль ядра с близким названием:

```
root@:~ # find /boot/ -name '*gpio*'
/boot/kernel/amdgpio.ko
/boot/kernel/bytgpio.ko
/boot/kernel/chvgpio.ko
/boot/kernel/ftgpio.ko
/boot/kernel/gpiobus.ko
/boot/kernel/gpioiic.ko
/boot/kernel/gpioled.ko
/boot/kernel/gpiospi.ko
/boot/kernel/gpioths.ko
/boot/kernel/gpioaei.ko
/boot/kernel/nctgpio.ko
```

Первой же строкой видим файл с именем **amdgpio.ko**. Файлы с суффиксом **.ko** являются загружаемыми объектными кодами ядра операционной системы (Kernel Loadable Object), или другими словами — загружаемыми модулями или «драйверами». Хотя термин «драйвер» не совсем верный, так как в загружаемом модуле может быть реализован практически любой функционал, а не только лишь драйвер устройства. Для загрузки и выгрузки **.ko** модулей в системе имеются утилиты **kldload** и **kldunload**, а для просмотра списка загруженных модулей — утилита **kldstat**. Воспользуемся последней для того, чтобы посмотреть какие модули сейчас загружены:

```
root@:~ # kldstat
Id Refs Address                Size Name
1    1 0xffffffff80200000  2512a30 kernel
```

Утилита **kldstat** показывает нам всего один модуль с именем **kernel** — это есть само ядро операционной системы. В «однопользовательском» режиме так и должно быть. Остальные модули (драйверы) загружаются постепенно, по мере необходимости, в процессе исполнения **rc** скриптов. Утилита **kldstat** помимо имени модуля выдает его размер и адрес расположения в виртуальной памяти ядра ОС, а так же число ссылок на данный модуль — **Refs**. Дело в том, что загружаемые объекты ядра в ОС FreeBSD образуют иерархическую структуру и таким образом можно проследить дерево зависимостей одного модуля от другого. Отсюда вытекает одно важное следствие — если число ссылок на модуль больше единицы, то выгрузить из памяти его отдельно нельзя, а требуется выгрузить сначала все зависимые от него модули (примечание: Refs также увеличивается, если драйвер/модуль имеет активных пользователей, например, открытый файл).

Что-ж, с помощью утилиты **kldload** попробуем загрузить модуль драйвера **amdgpio.ko**. Имя файла модуля, если он расположен в системном каталоге **/boot/kernel/** или **/boot/modules**, можно указывать без суффикса:

```
root@:~ #:~ % kldload amdgpio
gpio0: <AMD GPIO Controller> iomem 0xfed81500-0xfed818ff,0xfed81200-0xfed812ff irq 7 on acpi0
gpiobus0: <GPIO bus (ACPI-hinted)> on gpio0
gpioc0: <GPIO controller> at pins 0-255 on gpiobus0
gpiobus0: <unknown device> at pins 0,44,61-62,58-59,18
```

Выводимые на консоль сообщения показывают, что в системе зарегистрировано несколько новых логических устройств: **gpiobus0**, **gpioc0** и **gpio0**. Посмотрим какие сейчас модули присутствуют в памяти ядра:

```
root@:~ # kldstat
Id Refs Address                Size Name
 1    9 0xffffffff80200000  2512a30 kernel
 2    1 0xffffffff82e20000     22b0 amdgpio.ko
 3    1 0xffffffff82e23000     6760 gpiobus.ko
```

Видно, что модуль **amdgpio.ko** подтянул за собой другой модуль с именем **gpiobus.ko**. Ядро ОС отслеживает зависимости между различными модулями и автоматически подгружает недостающие модули. О том, как система узнаёт о таких зависимостях, мы рассмотрим далее при создании своего драйвера. А сейчас попробуем разобраться в том, для чего все этим логические устройства необходимы.

3.7. Взаимодействие с GPIO в ОС FreeBSD (GPIO Bus)

В ОС FreeBSD реализован отдельный фреймворк для работы с GPIO, который называется «GPIO Bus». Этот фреймворк представляет собой логическую «шину» к которой, с одной стороны, могут быть подключенный потребители (пользовательские приложения или другие модуля ядра), а с другой — устройства (физические или логически), реализующие взаимодействие с конкретной аппаратурой или её эмуляцию. В данном случае мы имеем: логическое устройство **gpiobus0**, представляющее точку подключения к «шине» GPIO; логическое устройство **gpioc0** - контроллер (GPIO Controller), экспортирующее на шину некоторое количество линий GPIO (pins); и драйвер физического устройства «AMD GPIO Controller» - **gpio0**, который реализует взаимодействие непосредственно с аппаратурой процессора. Если сейчас с помощью утилиты **devinfo** запросить структуру дерева устройств, то получим вот такую иерархию:

```
root@:~ # devinfo -rv | grep gpio
    gpio0 <AMD GPIO Controller> pnpinfo _HID=AMDI0030 _UID=0 _CID=AMDI0030 at
handle=\_SB_.GPIO
      gpiobus0 <GPIO bus (ACPI-hinted)>
        gpioc0 <GPIO controller> at pins=0-255 handle=(unknown)
```

Мы видим, что в системе представлено 256 линий (пинов) GPIO имеющих логическую нумерацию от 0 до 255. При этом, если ориентироваться на вывод команды **kldload**, для работы доступны линии с номерами: **0,44,61-62,58-59** и **18**. Эти линии на данный момент не имеют потребителей (сообщение "**unknown device**" от устройства **gpiobus0** говорит об этом), то есть свободны и никак не обрабатываются системой. В этот список попадает и пин с номером **0**, к которому, предположительно, подключена кнопка «Power».

А если пин **0** свободен, то мы имеем возможность поработать с ним прямо из пользовательского окружения. Во-первых, мы можем запросить у ядра ОС информацию о настройках данной линии GPIO. Для этого воспользуемся утилитой **gpioctl** и выведем информацию о первых 10-ти линиях GPIO на контроллере **gpioc0**:

```
root@:~ # gpioctl -f /dev/gpioc0 -lv | head
pin 00: 1       AMDGPIO0_PIN_0<IN>, caps:<IN,OUT,PU,PD>
pin 01: 1       AMDGPIO0_PIN_1<IN>, caps:<IN,OUT,PU,PD>
pin 02: 1       AMDGPIO0_PIN_2<IN>, caps:<IN,OUT,PU,PD>
pin 03: 1       AMDGPIO0_PIN_3<IN>, caps:<IN,OUT,PU,PD>
pin 04: 1       AMDGPIO0_PIN_4<IN>, caps:<IN,OUT,PU,PD>
pin 05: 0       AMDGPIO0_PIN_5<IN>, caps:<IN,OUT,PU,PD>
pin 06: 0       AMDGPIO0_PIN_6<IN>, caps:<IN,OUT,PU,PD>
pin 07: 1       AMDGPIO0_PIN_7<IN>, caps:<IN,OUT,PU,PD>
pin 08: 1       AMDGPIO0_PIN_8<IN,PU>, caps:<IN,OUT,PU,PD>
pin 09: 1       AMDGPIO0_PIN_9<IN>, caps:<IN,OUT,PU,PD>
```

Из данного вывода вино, что нулевая линия (**pin 00**) может быть сконфигурирована: как входной сигнал (**IN**), как выходной (**OUT**), с подтяжками «вверх» (**PU**) к питанию и «вниз» (**PD**) к «земле». Также утилита **gpioctl** показывает нам текущее состояние линий — нулевая линия сконфигурирована как **IN** (входной сигнал) и её текущее состояние равно логической «**1**». Для тех читателей, которые далеки от тонкостей разработки цифровой электроники,

поясню, что подтягивание линии GPIO к логической «1» в пассивном состоянии является обычным решением — сработка сигнала в таком случае выражается в замыканием линии GPIO на «землю», что с программной стороны видно как смена состояния линии с логической «1» на логический «0». Такой режим работы принято называть «Active Low» (активный сигнал — в нуле).

Во-вторых, с помощью этой же утилиты мы можем организовать цикл опроса состояния интересующей нас линии с выводом на терминал. Опросить конкретную линию GPIO можно следующим образом:

```
root@:~ # gpioctl -f /dev/gpioc0 0
1
```

Теперь мы можем создать небольшой shell-скрипт для непрерывного опроса состояния линии GPIO с номером 0, чтобы пронаблюдать изменение состояния этой линии в динамике.
Создадим файл **gpio0_test.sh** со следующим кодом:

```
root@:~ # cat gpio0_test.sh
#!/bin/sh

while [ 1 ]
do
        gpioctl -f /dev/gpioc0 0
        sleep 0.2
done
```

и поставим его на исполнение:

```
root@:~ # sh gpio0_test.sh
1
1
1
…
```

В момент выполнения скрипта несколько раз нажмем и отпустим кнопку «Power», вывод на консоль изменится как-то вот так:

```
1
1
0
0
1
1
0
1
0
1
1
0
1
1
1
1
1
1
^C
```

Каждый раз когда кнопка «Power» нажата, состояние линии GPIO меняется на логический «0». И наоборот, при отпускании кнопки состояние линии возвращается в пассивное, т. е. в логическую «1».

Достигнув этого результата я вскрикнул «Yesss!» и чуть было не свалился с дивана. Ведь это первый тест за прошедшие почти две недели исследований, в котором мне удалось

получить хоть какой-то отклик от кнопки «Power» на ноутбуке «HP Laptop 15s» и программно выяснить её состояние, при этом используя только стандартные средства операционной системы FreeBSD!

Опытный пользователь наверное уже почувствовал, что решение проблемы на подходе — ведь самым простым и очевидным решением было бы создать и поместить в «автозапуск» отдельный shell-скрип, который циклически проверял бы состояние нулевой линии GPIO с помощью утилиты **gpioctl** и посылал операционной системе сигнал для входа в режим энергосбережения с помощью команды **acpiconf -s 3**. Скрипт из трех строк, всё в духе UNIX, что может быть лучше ? :-) Такое решение действительно рабочее, но у него есть ряд существенных недостатков:

1. Скрипты запускаются только в «многопользовательском» режиме, а это значит, что в «однопользовательском» режиме кнопка «Power» работать не будет.
2. Скрипт потребляет ресурсы — память, процессорное время при каждом запуске утилиты **gpioctl**. Может быть это не так много по современным меркам, но на мой взгляд слишком много для такой мелкой задачи.
3. И самое важное — приложениям пользователя кнопка «Power» будет недоступна. Обычно в графических средах имеется специальное приложение, которое отображает пользователю меню для выбора действия при нажатии кнопки «Power»: «Suspend», «Reboot» или «Shutdown». Такое приложение работать не будет, что создаст неудобство для пользователя.

И в целом хотелось докопаться до сути, получить ответ на вопрос почему сигналы смены состояния линий GPIO не транслируются в ACPICA, ведь драйвер **amdgpio** загружен! Возможно, требуется еще какой-то промежуточный модуль. Поэтому я продолжил изыскания.

3.8. Вызов метода _EVT при смене состояния линий GPIO (модуль gpioaei)

Сообщение от устройства **gpiobus0** вида «**unknown device**» как бы намекает на то, что в операционной системе должен присутствовать еще какой-то компонент обрабатывающий состояния линий GPIO. Аналогичный намёк подает нам сообщение «**handle=(unknown)**» от устройства **gpioc0**. В последнем, под термином **handle** понимается путь в структуре иерархии объектов ACPI, который на данный момент не указан. Немного поразмыслив, я сделал поиск строки "**_EVT**" по всему дереву исходных кодов ядра ОС FreeBSD — какой-то компонент системы все-таки должен вызывать этот ACPI метод.

В результате поиска обнаружился модуль с именем **gpioaei**, исходный код которого расположен в файле **sys/dev/gpio/gpioaei.c**. Странички системного руководства для этого модуля также не нашлось, но из просмотра исходных кодов стало понятно его назначение.

Назначение модуля **gpioaei** следующее: при инициализации выяснить какие линии GPIO относятся к ACPI и для всех таких линий установить обработчик прерываний, а в обработчике вызывать ACPI метод **_EVT**, **_Exx** или **_Lxx** в зависимости о того, что требует ACPI. Ниже в листинге приведен код реализации обработчика из этого модуля:

Листинг 9а. Функция gpio_aei_intr() вызывающая метод _EVT.

```
static void
gpio_aei_intr(void * arg)
{
        struct gpio_aei_ctx * ctx = arg;

        /* Ask ACPI to run the appropriate _EVT, _Exx or _Lxx method. */
        if (ctx->type == ACPI_AEI_TYPE_EVT)
                acpi_SetInteger(ctx->handle, NULL, ctx->pin);
        else
                AcpiEvaluateObject(ctx->handle, NULL, NULL, NULL);
}
```

К слову сказать, AEI расшифровывается как «ACPI Event Information» — способ передачи информации о произошедшем событии из ядра операционной системы в подсистему ACPICA. Вот что говорится по этому поводу в спецификации ACPI:

> *5.6.5.2 _AEI Object for GPIO-signaled Events*
>
> *The _AEI object designates those GPIO interrupts that shall be handled by OSPM as ACPI events (see Section 5.6.5). This object appears within the scope of the GPIO controller device whose pins are to be used as GPIO-signaled events.*
>
> *Arguments:*
> *None*
> *Return Value:*
> *A resource template Buffer containing only GPIO Interrupt Connection descriptors.*

То есть в составе «Firmware», у объекта **_SB.GPIO**, предполагается наличие метода **_AEI**, который может вызвать операционная система (OSPM) для того, чтобы запросить список линий GPIO состояние которых требуется передавать в AML код. И действительно в таблице SSDT14 обнаружилась реализация такого метода:

Листинг 9б. Фрагмент ASL кода реализующий метод _AEI для выдачи конфигурации GPIO.

```
Scope (\_SB.GPIO)
  {
      Method (_AEI, 0, NotSerialized)  // _AEI: ACPI Event Interrupts
      {
          Name (BUF0, Buffer (0xF7)
          {
              /* 0000 */  0x8C, 0x20, 0x00, 0x01, 0x00, 0x01, 0x00, 0x11,  // . ......
              /* 0008 */  0x00, 0x00, 0x00, 0x00, 0x88, 0x13, 0x17, 0x00,  // ........
              /* 0010 */  0x00, 0x19, 0x00, 0x23, 0x00, 0x00, 0x00, 0x00,  // ...#....
              /* 0018 */  0x00, 0x5C, 0x5F, 0x53, 0x42, 0x2E, 0x47, 0x50,  // .\_SB.GP
              /* 0020 */  0x49, 0x4F, 0x00, 0x8C, 0x20, 0x00, 0x01, 0x00,  // IO.. ...
              /* 0028 */  0x01, 0x00, 0x11, 0x00, 0x03, 0x00, 0x00, 0x00,  // ........
              /* 0030 */  0x00, 0x17, 0x00, 0x00, 0x19, 0x00, 0x23, 0x00,  // ......#.
              ...

              /* 00F0 */  0x47, 0x50, 0x49, 0x4F, 0x00, 0x79, 0x00         // GPIO.y.
          })
          Return (BUF0) /* \_SB_.GPIO._AEI.BUF0 */
      }
      ...
  }
```

Данная реализация метода **_AEI** возвращает структуру **BUF0** размером 247 (0xF7) байт с описанием линий GPIO требуемых для работы ACPI «Firmware». Эта структура разбирается

компонентом ACPICA и преобразуется во внутреннюю структуру **ACPI_RESOURCE_GPIO**, описание которой дано в файле **sys/contrib/dev/acpica/include/acrestyp.h**.

Но вернемся к модулю **gpioaei**. Получается, что всё, что нам требуется это просто загрузить модуль **gpioaei** в память и кнопка «Power» должна автоматически заработать. Так и сделаем:

```
root@:~ # kldload gpioaei
gpio_aei0 at pins 0,44,61-62,58-59,18 on gpiobus0
gpio_aei0: Cannot allocate an IRQ for pin 0x0
gpio_aei0: Cannot set pin flags for pin 0x2c
gpio_aei0: Cannot allocate an IRQ for pin 0x3d
gpio_aei0: Cannot allocate an IRQ for pin 0x3e
gpio_aei0: Cannot allocate an IRQ for pin 0x3a
gpio_aei0: Cannot allocate an IRQ for pin 0x3b
gpio_aei0: Cannot allocate an IRQ for pin 0x12
```

Модуль загрузился, но тут же выдал несколько сообщений об ошибке "**Cannot allocate an IRQ for pin**", в том числе для важной для нас линии GPIO пин 0. Данное сообщение говорит о том, что установить обработчик прерываний для указанного ряда линий GPIO не удалось, но о причинах ничего не сообщается. Разумеется, я тут же попробовал несколько раз нажать кнопку «Power», однако, искомого результата не последовало. Будем выяснять по какой причине модуль **gpioaei** не может получить доступ к прерываниям от GPIO чтобы установить свой обработчик.

3.9. Анализ кода обработки прерываний от GPIO

Еще раз заглянем в исходный код модуля **gpioaei**, находящийся в файле **sys/dev/gpio/gpioaei.c**, и найдем участок формирующий приведенное выше сообщение об ошибке. Располагается он в функции **gpio_aei_enumerate()**, вызываемой из функции инициализации **gpio_aei_attach()**, и выглядит следующим образом:

Листинг 10а. Фрагмент функции gpio_aei_enumerate() устанавливающий обработчик прерываний на линию GPIO.

```
ctx->intr_rid = 0;
ctx->intr_res = gpio_alloc_intr_resource(sc->dev,
    &ctx->intr_rid, RF_ACTIVE, ctx->gpio,
    flags & GPIO_INTR_MASK);
if (ctx->intr_res == NULL) {
        device_printf(sc->dev,
            "Cannot allocate an IRQ for pin 0x%x\n", pin);

        gpio_pin_release(ctx->gpio);
        free(ctx, M_DEVBUF);
        continue;
}
```

В этом фрагменте кода происходит вызов функции **gpio_alloc_intr_resource()** с целью получить доступ к ресурсу обработки прерываний (аппаратные прерывания с точки зрения ОС это тоже ресурс), и, если ресурс получить не удалось (т. е. функция возвращает **NULL**), то выводится соответствующее диагностическое сообщение.

Функция **gpio_alloc_intr_resource()** является частью фреймворка «GPIO Bus», её реализация находится в модуле **gpiobus**, в файле **sys/dev/gpio/gpiobus.c**. Поэтому следующим шагом заглянем в этот файл, где обнаружим код следующей структуры:

Листинг 11а. Фрагмент кода обработки прерываний в gpiobus.

```
#ifdef INTRNG

struct resource *
gpio_alloc_intr_resource(device_t consumer_dev, int *rid, u_int alloc_flags,
    gpio_pin_t pin, uint32_t intr_mode)
{
        u_int irq;
        struct intr_map_data_gpio *gpio_data;
        struct resource *res;

        gpio_data = (struct intr_map_data_gpio *)intr_alloc_map_data(
            INTR_MAP_DATA_GPIO, sizeof(*gpio_data), M_WAITOK | M_ZERO);
        gpio_data->gpio_pin_num = pin->pin;
        gpio_data->gpio_pin_flags = pin->flags;
        gpio_data->gpio_intr_mode = intr_mode;

        irq = intr_map_irq(pin->dev, 0, (struct intr_map_data *)gpio_data);
        res = bus_alloc_resource(consumer_dev, SYS_RES_IRQ, rid, irq, irq, 1,
            alloc_flags);
        if (res == NULL) {
                intr_unmap_irq(irq);
                return (NULL);
        }
        return (res);
}

#else

struct resource *
gpio_alloc_intr_resource(device_t consumer_dev, int *rid, u_int alloc_flags,
    gpio_pin_t pin, uint32_t intr_mode)
{
        return (NULL);
}
#endif
```

Здесь мы видим условную компиляцию определяемую макро-переменной **INTRNG** активирующей одну из двух реализаций функции **gpio_alloc_intr_resource()**, при этом одна из реализаций (активная если **INTRNG** не установлена) — пустая заглушка!

Сразу возникает несколько вопросов: что такое **INTRNG**, в каком состоянии эта макро-переменная находится при сборке нашего кастомного ядра, и, соответственно, какая реализация функции **gpio_alloc_intr_resource()** является действительной. Не может ли оказаться так, что мы имеем дело с каким-то не полностью реализованным функционалом модуля **gpiobus** ?

Для меня, как для практика, проще всего оказалось сначала найти ответ на последний вопрос. Для этого я вставил команду препроцессора **#error DEADBEEF** в тело обеих реализаций функции **gpio_alloc_intr_resource()** чтобы сгенерировать аварийный останов компилятора, и еще раз запустил сборку ядра ОС FreeBSD.

Выдержка из файла **sys/dev/gpio/gpiobus.c** содержащая добавленные мной команды **#error**, с указанием номеров строк приведены в Листинге 11б:

Листинг 11б. Фрагмент кода обработки прерываний с внесенным оператором #error.

```
 92 #ifdef INTRNG
 93
 94 struct resource *
 95 gpio_alloc_intr_resource(device_t consumer_dev, int *rid, u_int alloc_flags,
 96     gpio_pin_t pin, uint32_t intr_mode)
```

```
 97 {
 98 #error DEADBEEF
 99         u_int irq;
100         struct intr_map_data_gpio *gpio_data;
101         struct resource *res;
102 
103         gpio_data = (struct intr_map_data_gpio *)intr_alloc_map_data(
104             INTR_MAP_DATA_GPIO, sizeof(*gpio_data), M_WAITOK | M_ZERO);
105         gpio_data->gpio_pin_num = pin->pin;
106         gpio_data->gpio_pin_flags = pin->flags;
107         gpio_data->gpio_intr_mode = intr_mode;
108 
109         irq = intr_map_irq(pin->dev, 0, (struct intr_map_data *)gpio_data);
110         res = bus_alloc_resource(consumer_dev, SYS_RES_IRQ, rid, irq, irq, 1,
111             alloc_flags);
112         if (res == NULL) {
113                 intr_unmap_irq(irq);
114                 return (NULL);
115         }
116         return (res);
117 }
118 #else
119 struct resource *
120 gpio_alloc_intr_resource(device_t consumer_dev, int *rid, u_int alloc_flags,
121     gpio_pin_t pin, uint32_t intr_mode)
122 {
123 
124 #error DEADBEEF
125         return (NULL);
126 }
127 #endif
```

После запуска сборки ядра я получил следующее сообщение об «ошибке»:

```
rz@hp15s:~/FreeBSD-src/src-15.x % make buildkernel KERNCONF=CUSTOM OBJROOT=$PWD/obj
/home/rz/FreeBSD-src/src-15.x/sys/dev/gpio//gpiobus.c:124:2: error: DEADBEEF 124 | #error
DEADBEEF |  ^ 1 error generated. *** Error code 1
```

Как я и предполагал, «ошибка» сработала в строке номер 124, то есть в той ветке кода которая реализует заглушку. А это в свою очередь говорит о том, что поддержка прерываний в модуле **gpiobus** либо не имплементирована, либо не включена по умолчанию. Но почему ?

Наивно предполагать, что кто-то из разработчиков ОС FreeBSD забыл включить в конфиге **GENERIC** опцию ядра устанавливающую макро-переменную **INTRNG**. Кстати, опция ядра **options INTRNG** действительно существует, но как выяснилось, доступна она только для платформ ARM, ARM64 и RISC-V, о чем говорит следующий результат поиска по конфигурационным файлам:

```
rz@hp15s:~/FreeBSD-src/src-15.x % grep -r 'option.*INTRNG' sys/
sys/arm/conf/DEFAULTS:options   INTRNG            # All arm systems use INTRNG these days
sys/arm64/conf/DEFAULTS:options         INTRNG
sys/riscv/conf/DEFAULTS:options         INTRNG
```

Складывается впечатление, что INTRNG — это фреймворк внутри ядра ОС FreeBSD отвечающий за обработку аппаратных прерываний, который реализован для ряда платформ, но не реализован для столь популярной платформы как x86/amd64. Чтобы понять почему так получилось, нам придется слегка погрузиться в механизма обработки аппаратных прерываний ядра ОС FreeBSD и его API.

3.10. Подсистема обработки аппаратных прерываний в ОС FreeBSD

К сожалению, в сети Интернет не так много материалов (статей, книг или руководств) освещающих внутренний мир ядра ОС FreeBSD, подсистемы обработки прерываний в частности, и того как эта подсистема развивалась. У нас есть ряд страниц системного руководства (**man 9 intro**) посвященных ядру, есть презентация «Interrupt filtering» от 2006 года за авторством Paolo Pisati описывающая новый (на тот момент) API, а также обрывки дискуссий в списках рассылок freebsd-*@FreeBSD.org.

Потратив изрядное количество времени на изучение этих материалов, на чтение исходных кодов ядра и экспериментов с ним, у меня сложилось некоторое общее понимание того, как работает подсистема прерываний в ОС FreeBSD, какие имеются API (фреймворки) и как они появились. Часть этой информации я попытаюсь в очень сжатой форме изложить в данной главе.

3.10.1. API intr_event

Небольшая заметка на странице системного руководства **man 9 intro** рассказывает о том, что в ядре ОС существует разделение API для обработки прерываний на «машинозависимого» (MD) и «машинонезависимого» (MI), а в центре находится внутрисистемный API называемый **intr_event** соединяющий MI и MD API между собой. К **intr_event**, с одно стороны, подключаются контроллеры прерываний (их драйверы) представляющие архитектурно зависимые (MD) устройства, такие как PIC/APIC/IOAPIC на платформе x86/amd64, GIC на ARM, XICS/XIVE на PowerPC и т.д. Они поставляют в подсистему прерываний поток «прерывательных событий» или «interrupt events» (уж извините, не знаю как это правильно перевести на русский). С другой стороны к **intr_event** подключаются MI потребители событий, которыми обычно являются драйверы различных устройств инициирующих аппаратные прерывания в процессе своей жизнедеятельности.

Задача API **intr_event** состоит в том, чтобы организовывать входящий поток событий от различных источников прерываний в очереди, разбрасывая их между имеющимися процессорным ядрами (учитывая приоритет и привязанность к процессорному ядру — «CPU affinity»), и своевременно вызывать зарегистрированные, связанные с конкретным источником события, «обработчики». Под «обработчиком» в **intr_events** понимается структура с таким же названием (**struct intr_event**) содержащая список указателей на две сущности: **filter** и **handler**, где **filter** — это функция фильтрации выполняемая в общем контексте с остальными обработчиками событий, а **handler —** процедура обработки исполняемая в отдельном потоке внутри ядра ОС со своим независимым контекстом. Таким образом, обработка входящих прерываний разделяется на две части: быструю и легковесную фильтрацию в функции **filter** и тяжеловесную работу в отдельно выделенном потоке **ithread** исполняемым, как правило, с приоритетом REAL_TIME.

Функция фильтра должна быть максимально короткой по времени исполнения, в ней не разрешается использовать функции и фичи ядра приводящие к блокировке: семафоры, мьютексы, sleep, блокирующий ввод/вывод. Также нежелательно в фильтре осуществлять копирование блоков данных, так как это занимает большое время исполнения и может вызвать «голодание» («irq starving») других обработчиков. Всё, что допускается в фильтре это: изменение счетчика, установка флага и вращение указателей в циклическом буфере. Для разграничения доступа к общей переменной фильтр может использовать «spinlock» — способ организации доступа к переменной через атомарные операции процессора.

У функции фильтра есть своя обязанность — сообщить контроллеру прерываний (и устройству сформировавшее прерывание) о том, что прерывание обработано, то есть выполнить процедуру «End of Interrupt» (EOI) путем сброса флага ожидания обработки

(pending interrupt flag) или путем отправки специальной команды в контроллер прерываний (в случае с MSI/MSI-X). Если этого не сделать, то это может привести к тому, что контроллер либо перестанет присылать запросы по данному источнику (состояние «irq stall»), либо наоборот — завалит систему непрерывным потоком прерываний (состояние «irq storm»). Конкретный исход в данной ситуации зависит от того как сконфигурирована линия источника прерываний: если прерывания происходят по фронту сигнала (INTR_EDGE_RAISING или INTR_EDGE_FALLING), то получим ситуацию «irq stall», если же по уровню сигнала (INTR_LEVEL_HIGH или INTR_LEVEL_LOW), то получим «irq storm».

Если из фильтра перезапустить контроллер по данному источнику прерываний не позволяет логика работы с устройством, то процедуру «EOI» допускается выполнить внутри потока **ithread** в процедуре **handler**, но при этом фильтр должен временно маскировать (заблокировать) поступление прерываний по данному источнику. Для этого MD API контроллера прерываний предоставляет набор функций-примитивов: **pic_mask()** и **pic_unmask()**.

Когда функция фильтрации **filter** завершается, она возвращает в ядро одно из следующих значений, которое определяет дальнейший ход обработки данного события:

- **FILTER_STRAY** — событие не обработано, если фильтр посчитал, что данное прерывание поступило не от обслуживаемого им устройства. Т.е. возможны случаи когда одна линия прерывания используется одновременно несколькими устройствами (случай «shared irq» задается флагом **RF_SHAREABLE**);
- **FILTER_HANDLED** — событие полностью обработано, можно вызывать «EOI» и завершать обработку события.
- **FILTER_SCHEDULE_THREAD** — событие обработано не полностью, требуется запустить поток **ithread** с соответствующим обработчиком **handler**.

Поточная процедура **handler** может выполнять свою работу сколько угодно долго, лишь бы успеть до поступления следующего события от данного источника, чтобы не переполнить внутреннюю очередь и буфера входных данных. Если **handler** по каким-то причинам не справляется с задачей (не успевает обрабатывать поступающие данные), то возникает ситуация «lost irq» приводящая к частичной потере входных данных, что в некоторых случаях может быть некритично, но в целом - нежелательно. Поток может использовать мьютексы, блокируемый ввод/вывод и даже временно засыпать — система сама его разбудит при поступлении события.

Если логика обработки прерывания не позволяет выполнить процедуру «EOI» внутри фильтра, то она должна быть выполнена по завершению работы соответствующего потока **ithread**.

Суммируя всё выше сказанное, API **intr_event** можно свести к нескольким основным функциям описанным в заголовочном файле **sys/sys/interrupt.h**:

- **intr_event_create()** — регистрирует в системе новое событие по заданному источнику прерывания, устанавливает функции **pre_ithread** и **post_ithread**;
- **intr_event_destroy()** — удаляет из системы ранее зарегистрированный источник событий;
- **intr_event_add_handler()** — добавляет процедуры обработчика (**filter** и **handler**) к зарегистрированному источнику событий, создает в системе новый поток исполнения **ithread** если **handler** не NULL;
- **intr_event_remove_handler()** — удаляет ранее добавленный обработчик, уничтожает **ithread** если больше нет зарегистрированных **handler**-ов;
- **intr_event_handle()** — основная функция инициирующая вызов процедур обработчика, должна вызываться из низкоуровневого API контроллера прерываний.

Функция добавления обработчика **intr_event_add_handler()** возвращает некоторое произвольное число, которое рассматривается как **cookie** для ссылки на зарегистрированный обработчик (комплект **filter** и **handler**). Удаление обработчика осуществляется передачей полученного ранее значения **cookie** в функцию **intr_event_remove_handler()**.

Структура **struct intr_event**, по мимо уже упомянутых списков обработчиков, содержит еще два опциональных указателя на процедуры с именем **pre_ithread** и **post_ithread**, которые устанавливаются при вызове **intr_event_create()**. Эти указатели предназначены для MD драйвера контроллера прерываний, они вызываются (если не NULL) соответственно перед и после выполнения потоковых обработчиков **handler**, и служат для временного маскирования и демаскирования данного источника событий. В руководстве указано, что **pre_ithread** и **post_ithread** вызываются только в том случае, если процедурой фильтром определен вызов хотя бы одного **handler** для данного источник, то есть для него уже создан и запущен отдельный поток **ithread**.

У данного API есть еще одна особенность — возможность создавать программные прерывания (SWI). Никто не мешает вызывать функцию **intr_event_handle()** не из драйвера контроллера прерываний, а, скажем, из таймера или какого-то системного исключения (exception), или даже из приложения пользователя через псевдо-драйвер. Для работы с SWI предусматриваются соответствующие функции **swi_add()**, **swi_remove()** и **swi_sched()** описанные на странице руководства **man 9 swi**.

Подробности использования API **intr_event**, с примером кода, можно прочесть на странице руководства **man 9 intr_event**.

3.10.2. API bus_setup_intr

В рамках иерархической системы устройств принятой в ядре ОС FreeBSD, каждый драйвер устройства может быть потребителем услуг, поставщиком услуг или реализовывать сразу два амплуа. Драйвер который предоставляет услуги (т. е. имеет подключенных потребителей), а сам подключается к другому драйверу, принято называть «шиной». Например, модуль **gpiobus** является драйвером шины GPIO, у него могут быть потребители — другие драйверы, такие как **gpioaei** или **gpioled**. Драйвер шины заведует и распоряжается ресурсами своей шины, в том числе линиями аппаратных прерываний, либо транслирует эту обязанность вверх по иерархии.

Для подключения потребителя к шине, в части обработки прерываний, имеется отдельный API описанный на странице руководства **man 9 bus_setup_intr**. Этот API состоит из трех основных функций:

- **bus_alloc_resource()** — для запроса ресурса (источника прерываний);
- **bus_setup_intr()** — для регистрации обработчика принадлежащего потребителю;
- **bus_teardown_intr()** — для снятия регистрации обработчика принадлежащего потребителю;

В этом API при регистрации и снятия обработчиков, точно так же как в API **intr_event**, используются **cookie** для ссылки.

3.10.3. Аппаратные прерывания как виртуальный ресурс (global IRQ)

Когда драйвер хочет получить доступ к источнику аппаратных прерываний, он вызывает функцию **bus_alloc_resource()** в которую передает тип ресурса **SYS_RES_IRQ,**

номер первой единицы ресурса **start**, номер последней **end** и количество запрашиваемых ресурсов **count** (параметр **count** требуется потому, что некоторые ресурсы выделяются блоками по 4, 8 или 4096 штук — как-то страница памяти). В ответ функция возвращает указатель на структуру **struct resource**, а также заполняет переменную **rid** помещая в неё порядковый номер ресурса.

Если ресурсом является аппаратная линия прерывания, то параметры **start** и **end** должны указывать на некий **глобальный номер** линии прерывания ассоциированный с данным источником события. Чтобы преобразовать реальный номер физической линии прерывания (например GPIO пин 0), к глобальному, API предоставляет специальную функцию **intr_map_irq()** в которую передается непрозрачная структура **struct intr_map_data**. Непрозрачной она является потому, что её внутреннее устройство известно лишь потребителю (драйверу который запрашивает ресурс) и конечному контроллеру прерываний (PIC) который будет обрабатывать запросы на переднем фланге. Драйвер потребитель должен правильно заполнить структуру **struct intr_map_data** указав номер линии прерывания, флаги и режимы работы этого источника, так, чтобы это было понятно драйверу контроллера прерываний.

В листинге 11б мы как раз видим такой запрос формируемый функцией **gpio_alloc_intr_resource()** в составе драйвера **gpiobus**. В нём сначала происходит выделение памяти и заполнение структуры **struct intr_map_data_gpio** (строки 105-107 листинга), являющейся частным случаем непрозрачной структуры **struct intr_map_data.** После чего, указатель на эту структуру преобразуется в **struct intr_map_data*** и передается в функцию **intr_map_irq()** (строка 109 листинга) для выполнения преобразования номера физической линии (пина) GPIO в глобальный номер **irq**.

Такая «глобализация» линий прерываний позволяет ядру операционной системы абстрагироваться от физической природы источника прерываний. В нашем случае с драйвером **gpiobus**, огромное количество линий GPIO контроллера, способных генерировать аппаратные прерывания и имеющих свою внутреннюю нумерацию (а также свойства), спрятаны за GPIO контроллером, который висит на одной физической линии **IRQ7** подключенной к контроллеру прерываний IOAPIC процессора архитектуры x86/amd64. Подобная же история происходит с прерываниями от PCI и PCIe устройств которые формируются не физическим сигналом, а передачей MSI/MSI-X сообщения по шине, источник каждого из которых может иметь свой глобальный номер линии прерывания.

Внутри подсистемы прерывания ядра ОС номера глобальных линий **irq** хранятся в таблице **irq_map**, добавляются в неё по запросу функцией **intr_map_irq()** и удаляются с помощью **intr_unmap_irq()**. Это означает, что один и тот же номер глобального **irq** может быть использован различными устройствами в разные моменты времени. Драйвер **gpiobus** вызывает **intr_unmap_irq()** каждый раз, когда его потребитель высвобождает ресурс вызовом функции **gpiobus_release_resource()**.

Такой метод работы с прерываниям в ОС FreeBSD получил название «***New-style Interrupt Framework***», или сокращенно фреймворк **INTRNG**, описание функций его API находится в файле **sys/sys/intr.h**. Появился он в ядре ОС FreeBSD не сразу.

Минутка истории. Напомню, что предшественником ОС FreeBSD была ОС 386BSD разработанная парой William Jolitz и Lynne Jolitz в 1989-1992 годах в рамках проекта переноса 4.3BSD на платформу IBM PC/AT i386 (x86). В то время машины с i386 процессором комплектовались достаточно простым контроллером прерываний состоящим из двух микросхем i8259A «Programmable Interrupt Controller» (PIC) соединенных каскадом, каждая из которых предоставляла по 8 линий IRQ (всего было доступно 15 линий IRQ). ОС FreeBSD унаследовала от ОС 386BSD код подсистемы обработки прерываний которая была

весьма проста, частично написана с использованием встроенного ассемблера и, следовательно, заточена под этот тип PIC контроллера. Появившийся в середине 90-х процессор Intel Pentium имел в своем составе более сложный контроллер прерываний «Advanced Programmable Interrupt Controller» (APIC), но добавление его поддержки не сильно повлияло на архитектуру ядра ОС. В конце 1990-х в ядро ОС FreeBSD начали добавлять поддержку других платформ, в частности DEC Alpha (AXP) и, чуть позже, PowerPC. Это привело к необходимости разделить все внутриядерные API на два класса - низкоуровневый машинозависимый (MD) и высокоуровневый независимый от архитектуры (MI). Вместе с этим, с версией 5.0-RELEASE, появился первый фреймворк для организации обработки прерываний с использованием отдельных легковесных потоков исполнения, содержащий функции **ithread_create()**, **ithread_add_handler()**, **ithread_schedule()** и т.д. Позже, с версии 6.2-RELEASE, этот API и всего функции переименовали в **intr_event_*** , который, как мы видели, существует по сей день. Не смотря на разделения API на MD и MI, в ядре ОС источники прерываний всё еще были жестко привязаны к физическим линиями IRQ. Проблему создавала только шина PCI в которой аппаратные события передавались с помощью MSI сообщений, для её решения был добавлена прослойка преобразующая MSI сообщения в номера несуществующих IRQ линий с номерами 24 и выше.

Появление поддержки микропроцессоров ARM и дальнейшее развитие PowerPC привело к необходимости пересмотреть устройство подсистемы обработки прерываний. Вызвано это было тем, что используемые контроллеры прерываний GIC (в ARM) и XIVE (в PowerPC) позволяли создавать очень сложные древовидные и перенастраиваемые пути соединения источников прерываний, а XIVE внес в эту тему элемент виртуализации. Еще одной сложность стало то, что на «embedded» ARM платформах для конфигурации аппаратуры используется технология «Flattened Device Tree» (FDT), которую пришлось позаимствовать из ОС Linux. Всё это привело к появлению в 2005 году фреймворка INTRNG, который в 2016-м году, с версии 11.0-RELEASE, был существенно переработан и получил название «New-style INTRNG». Основная цель переработки состояла в «глобализации» прерываний и поддержки сложных топологий. На мой взгляд здесь больше подходит термин «виртуализация» обработки прерываний, но разработчики ядра почему-то использовали слово «global», см цитату ниже. Описание усовершенствованного фреймворка INTRNG можно прочитать в комментариях к патчу https://reviews.freebsd.org/D5730, ниже приведена небольшая выдержка из него:

> *(A) INTRNG works only with **global interrupt numbers**. Each of them is associated with some interrupt source (ISRC) on some interrupt controller (PIC). And only a PIC can know which global interrupt numbers are associated with its ISRCs. So, if someone wants to know a global interrupt number of some specific interrupt, the query must always end in some PIC.*
> *Basically, a query should be done thru INTRNG framework. However, in some special cases when a PIC is an integral part of another device, it could be done within that device (not thru INTRNG framework). For example, someone could ask a GPIO framework for an interrupt for pin 20 on gpio1. If a PIC is a part of that gpio1 device, the query may be resolved internally.*
>
> *(B) On modern SOCs, a description of an interrupt may be quite complex and may differ from PIC to PIC. It identifies an interrupt itself as well as its configuration. A description is not only associated with an interrupt, but also and mainly with a device. An interrupt may be utilized by more devices either at the same time or separately on user demand.*

Про сложности реализации такой обработки прерываний для различных платформ можно прочесть в одной из старых Wiki статей на сайте проекта FreeBSD озаглавленной «Complicated Interrupts».

В классической (legacy) схеме, когда устройству выделялась одна физическая линия IRQ, часто разделяемая с другими устройствами, есть ряд изъянов влияющих на производительность. Дело в том, что современные устройства имеют сложную структуру с множеством подчиненных устройств в своём составе. В случае когда все эти подчиненные устройства спрятаны за одной линией IRQ, у обработчика прерываний (в составе драйвера) возникает необходимость выяснять какое именно подчиненное устройство требует внимания. Для этого обработчик вынужден сначала опросить ряд регистров статуса и, далее основываясь на полученных данных, вызвать одну из (или несколько, но по очереди) внутренних подпрограмм обработки, соответствующую данному источнику прерываний. Например, в случае с Ethernet контроллером, имеется два подчиненных устройства — передающий тракт (TX queue) и приемный (RX queue). При каждой сработке прерывания, в классической схеме, драйвер вынужден тратить драгоценное процессорное время на выяснение какую из подпрограмм вызвать — для приема входящего пакета или для передачи исходящего. Если таких подчиненных устройств много (несколько десятков приемных и передающих трактов, как у современных сетевых адаптеров), то при гигантском потоке прерываний драйвер будет расходовать существенное количество вычислительных ресурсов только на внутреннюю диспетчеризацию, создавая при этом ненужную работу (overhead) и замедляя пропускную способность сетевого интерфейса. В современных PCI/PCIe устройствах все подчиненные устройства имеют возможность генерировать т. н. MSI (или MSI-X) сообщения, которые доставляются контроллеру прерываний, который, в свою очередь, инициирует сработку прерывания центрального процессора — ему передается номер источника сообщения, который преобразуется операционной системой в глобальный номер irq (вектор) и производится вызов индивидуального для этого источника обработчика событий. Иными словами, *диспетчеризация переносится на аппаратуру!*

Еще одно положительное свойство «глобализации» прерываний состоит в том, что индивидуальные обработчики (исполняемые в потоках **ithread**) можно привязывать к конкретным вычислительным ядрам процессора (CPU affinity). Таким образом, с одной стороны резервируя под массивную нагрузку часть вычислительных ресурсов системы, а с другой — минимизируя накладные издержки связанные с переключением контекста и переносом данных кэша между процессорными ядрами. В случае с классической схемой, когда за одной линией IRQ находится множество подчиненных устройство, такая привязка либо невозможна, либо требует дополнительных накладных расходов внутри обработчика.

Но почему «New-style INTRNG» все еще не поддерживает x86/amd64 архитектуру ? Ответ прост — для этого потребуется слишком много переделок, внедрить которые «незаметно» для пользователя не получится. Старый код обработки прерываний, который все еще используется в ОС FreeBSD на x86 архитектуре, не стоял на месте — его постоянно развивали, в нём есть поддержка MSI/MSI-X, «CPU affinity» и многих других современных фич. Он хорошо оптимизирован с точки зрения производительности, что является одной из отличительных черт ОС FreeBSD. А так, как основными пользователями этой ОС, на данный момент, являются компании эксплуатирующие высоконагруженные серверы для доставки контента (CDN), преимущественно архитектуры x86, то перевод подсистемы прерываний на новый фреймворк с большой вероятностью приведет к неожиданным сбоям и простоям на их сетях при внедрении новых версий системы, со всеми негативными последствиями для её популярности и, скорее всего, к сокращению финансирования разработки.

Тем не менее, необходимость такая назрела и попытки перетащить машинозависимые части x86 ядра FreeBSD на INTRNG фреймворк предпринимаются. Об этом я расскажу в одной из последних глав. Главной движущей силой тут является необходимость поддержки технологии «Sleep-to-Idle» (S0ix) и ноутбуков с Hardware-Reduced ACPI.

На этом закончим наш экскурс в устройство подсистемы прерываний ОС FreeBSD и приступим к решению первоначальной проблемы — исправлению «сломаной» кнопки «Power», для чего нам придется написать свой собственный драйвер.

4. Разработка модуля «gpiopwrb» для протезирования прерываний от кнопки «Power»

Для тех читателей, которые пропустили несколько предыдущих глав или просто запутались в моём витиеватом повествовании, коротко поясню в чём суть проблемы, решение для которой нам далее предстоит реализовать.

Как выяснилось в результате исследования, кнопка включения/выключения питания «Power» на ноутбуке «HP Laptop 15s» подключена к линии (пину) 0 контроллера GPIO находящегося в составе процессора AMD Ryzen. У контроллера AMD GPIO имеется большое количество линий ввода/вывода которые можно настроить на формирование аппаратного прерывания при изменении состояния линии, в том числе это доступно и для линии к которой подключена кнопка «Power». Программное обеспечение ACPI «Firmware» этого ноутбука (AML код исполняемый в виртуальной машине компонента ACPICA) разработано с таким расчетом, что при изменении состояния кнопки «Power», операционная система вызовет метод **_SB_.GPIO._EVT** с нулевым параметром равным номеру пина GPIO (в нашем случае пин 0). За вызов этого метода отвечает модуль ядра **gpioaei** который использует API интерфейс предоставляемый модулем **gpiobus** для работы с линиями GPIO, в том числе для конфигурирования линии как источника прерывания и регистрации соответствующего обработчика прерываний. Модуль **gpiobus**, в свою очередь, зависит о модуля **amdgpio** реализующий машинозависимый (MD) интерфейс для взаимодействия с контроллером AMD GPIO. При погружении в исходный код модуля **amdgpio** выяснилось, что в нём не имплементирована работа с прерываниями, так как более высокоуровневый API обработки прерываний предоставляемый модулем **gpiobus** требует поддержки т.н. «New-style INTRNG» фреймворка для работы с прерываниями, а этот фреймворк не доступен для платформ x86 и amd64.

Наиболее верное решение данной проблемы состоит в переводе подсистемы обработки прерываний, реализованной в ядре ОС FreeBSD для x86/amd64 платформы, на новый фреймворк INTRNG и добавлении соответствующей поддержки в код **amdgpio**. Обсуждение этой темы в багтрекере на сайте проекта FreeBSD показало, что в обозримом будущем такой переход не случится, так как это весьма большой объем работы затрагивающий хорошо отлаженный и стабильно работающий десятилетиями код.

Остается не совсем красивое, но рабочее решение — протезировать отсутствующее прерывание от линии GPIO 0 путем её регулярного опроса (poll) из ядра ОС и формировании вызова ACPI метода **_SB_.GPIO._EVT** в случае если статус линии изменился (кнопка «Power» нажата). Этот способ мы уже фактически реализовали с помощью shell-скрипта, теперь же мы реализуем его с помощью загружаемого модуля ядра (драйвера). Такой способ гораздо экономичнее с точки зрения потребляемых ресурсов и лишен ряда недостатков описанных в главе «*3.7. Взаимодействие с GPIO в ОС FreeBSD (GPIO Bus)*».

Но перед тем, как разрабатывать «драйвер кнопки Power», который далее будем называть **gpiopwrb**, неплохо было бы ознакомиться с тем, как устроены загружаемые модуля ядра в ОС FreeBSD. Об этом и пойдет речь в следующей главе.

4.1. Структура драйвера (модуля) в ОС FreeBSD

Как отмечалось в главе *2.4*, модуль ядра в ОС FreeBSD представляет собой объект («Kernel Object» или «KObj») расширяющий возможности системы, для чего модуль экспортирует свой набор методов или даже свой интерфейс (шину) если он предполагает обслуживание подключений от других модулей. Все модули в системе связаны между собой в единую иерархическую структуру. Регистрацией и отслеживанием загруженных в систему

объектов занимается специальный фреймворк — «Newbus API», в составе которого устройство с именем **nexus0** является корнем дерева зарегистрированных объектов, само дерево тоже называется «Nexus». Пример фрагмента дерева Nexus приведен на рис. 5, его текущее состояние можно отобразить с помощью команды **devinfo**.

```
nexus0
  └─ pci0
       └─ ehci0   (or xhci0, depending on the host controller)
            └─ usbus0
                 └─ uhub0   (the root hub)
                      └─ uhub1 (a downstream hub, if present)
                           └─ [class driver]
```

Рис. 5. Фрагмент древовидной структуры объектов (модулей) Nexus в ОС FreeBSD.

Далее по тексту я буду использовать термины «объект», «модуль» и «драйвер» как взаимозаменяемые, хотя в общем случае это не так: модуль может реализовывать драйвер устройства, а может и не быть драйвером либо быть драйвером шины (bus). Также один модуль может содержать несколько объектов составляющих драйвер сложного устройства. Однако, для упрощения картины будем предполагать, что модуль который нам предстоит разработать является драйвером состоящим из одного объекта KObj.

4.1.1. Контекст (softc)

Каждый объект (модуль) регистрируемый в системе должен где-то сохранять свои внутренние данные и своё состояние. Эту, зарезервированную под конкретный объект, область памяти принято называть «контекстом» или в терминах разработчиков ядра ОС - «softc» (от «Software Context»). «Softc» в общем случае представляет собой непрозрачную, с точки зрения ОС, структуру данных, внутреннее устройство которой известно только драйверу (модулю) работающему с ней. Как правило, для каждой копии объекта, т. е. для каждого отдельного устройства обслуживаемого драйвером, резервируется своя копия структуры «softc», указатель на которую, представленный как тип **void*,** передается в качества параметра в метод инициализации объекта. Структура «softc» может располагаться как в глобальной области памяти ядра, то есть объявлена статически, так и динамически аллоцироваться (и уничтожаться) с помощью внутриядерных функций **malloc()** и **free()**. Да, всё верно, внутри ядра BSD систем имеется свой набор средства работы с динамической памятью представленный функциями **malloc()**, **realloc(), free()** и т. д., которые по поведению аналогичны своим собратьям из пользовательской библиотеки LIBC, но ведут себя гораздо более хитрее за счет дополнительного параметра — тип аллоцируемого объекта. Более подробно о динамическом распределении памяти в ядре ОС можно прочить на странице системного руководства по команде **man 9 malloc**.

4.1.2. Методы устройства (device)

Для описания нового объекта в системе предоставляется API (макро) **DEFINE_CLASS()**, с помощью него модуль сообщает операционной системе имя нового класса, указывает перечень методов которые он желает переопределить, а также методы которые будут экспортироваться в систему этим модулем. Как правило, типичный модуль реализующий драйвер должен переопределить следующие методы относящихся к классу **device**:

Метод **device_probe** — вызывается родительским классом автоматически при выполнении процедуры перенумерации устройств в дереве Nexus, например, в момент загрузки ОС или в момент обнаружения (подключения) нового устройства. В методе **device_probe** модуль должен определить, может ли он работать с новым устройством, указатель на структуру типа **device_t** с описанием которого передается в качестве входного параметра. Если данный модуль распознал поддерживаемое им устройство и готов работать с ним, то он вызывает функцию **device_set_desc()** чтобы присвоить новому устройству имя и завершается с возвратом значения **BUS_PROBE_GENERIC**. Если модуль (драйвер) не опознал новое устройство, то он завершается с возвратом значения ошибки **ENXIO**.

Метод **device_identify** — является разновидностью метода **device_probe**, он вызывается родительским классом в том случае, если подключаемый к нему драйвер заявляет об устройстве о котором родительский класс ничего не знает, т. е. информация о нём заранее не известна родительскому классу и выполнить автоматическую процедуру перенумерации с участием этого устройства невозможно. Обычно такие устройства находятся на аппаратной шине лишенной механизмов автоматического сканирования и распознавания подключенных устройств (например, на шине ISA, I^2C или на линиях GPIO). Драйвер такого устройства самостоятельно выполняет процедуру подключения к родительскому классу используя оберточную функцию **BUS_ADD_CHILD()** из метода **device_identify**.

Метод **device_attach —** вызывается системой следом за методом **device_probe** (или **device_identify**) если модуль рапортовал о готовности обслуживать новое устройство. В этом методе модуль должен запросить у системы все необходимые ресурсы (порты ввода/вывода, прерывания, блоки конфигурационных данных ACPI), выдели память под контекст, подготовить и запустить потоки обработчиков заданий, установить обработчик прерываний, выполнить инициализацию устройства и завершиться с кодом **0**. Либо вернуть код ошибки если что-то пошло не так. Важно чтобы метод **device_attach** выполнил высвобождение ранее запрошенных ресурсов (памяти, шины и т.д.) если в процессе возникла ошибка. Если этого не сделать, то возникнет утечка ресурсов и в следующий раз система может отказать модулю в выделении ресурса.

Метод **device_detach** — вызывается в том случае, если устройство отключилось, либо модуль выгружен системой или системным администратором. В данном методе модуль обязан произвести все операции деинициализации в обратном порядке: деинициализировать (успокоить) аппаратуру устройства, маскировать (заблокировать) прерывания, снять с регистрации обработчик прерываний, остановить потоки обработки заданий и высвободить память выделенную под контекст и буфера. В случае успеха метод обязан вернуть код **0**, иначе — вернуть код соответствующий возникшей ошибке.

Более подробная информация о методах класса **device** можно получить на страницах системного руководства **man 9 device** и **man 9 driver**.

4.1.3. Методы экспортируемого интерфейса (bus interface)

Класс описанный модулем может экспортировать для потребителей свой набор специфических методов. Этот список методов принято называть «интерфейсом шины» (bus interface). Описание интерфейса производится на специализированном языке «М» - Си подобном языке с использованием структур данных предопределенных в заголовочных **.h**-файлах. Код интерфейса помещается в файл вида **foo_if.m**, где **foo** — название класса. Компиляция интерфейсных файлов производится с помощью специализированного Perl-скрипта, он парсит М-файл и генерирует из него Си-шный файл **foo_if.h**, который в свою очередь участвует в сборке модуля, а также может быть использован при компиляции модулей-потребителей зависимых от модуля **foo**. Вызов скрипта для парсинга М-файлов интегрирован в систему сборки ядра, всё что требуется сделать — это добавить **foo_if.m** в

список файлов исходного кода модуля сборочного скрипта. Ниже, в качестве примера, в листинге 12а приведена выдержка из файла **sys/dev/gpio/gpiobus_if.m** описывающая часть интерфейсных методов модуля шины **gpiobus**.

Листинг 12а. Фрагмент файла описания интерфейса gpiobus_if.m.

```
INTERFACE gpiobus;

#
# Dedicate the gpio bus control for a child
#
METHOD int acquire_bus {
        device_t busdev;
        device_t dev;
        int how;
};

#
# Release the bus
#
METHOD void release_bus {
        device_t busdev;
        device_t dev;
};

#
# Set value of pin specified by pin_num
#
METHOD int pin_set {
        device_t dev;
        device_t child;
        uint32_t pin_num;
        uint32_t pin_value;
};
```

На стадии парсинга М-файла, к имени каждого метода задаваемого оператором **METHOD** будет добавлено имя класса задаваемого в этом же файле оператором **INTERFACE**. В приведенном листинге имя интерфейса определено как **gpiobus**, значить в результате парсинга будут экспортированы следующие методы:

```
static int gpiobus_acquire_bus(device_t, device_t, int);
static void gpiobus_release_bus(device_t, device_t);
static int gpiobus_pin_set(device_t, device_t, uint32_t, unsigned int);
```

и т. д.

Реализация этих методов на языке Си, как правило, располагается в главном файле модуля, в данном случае в файле **sys/dev/gpio/gpiobus.c**. Функции реализующие интерфейсные методы должны быть зарегистрированы в дереве Nexus, для этого в коде модуля используется структура типа **device_method_t** содержащая массив указателей на функции имплементирующие соответствующие методы. В листинге 12б приведен фрагмент кода демонстрирующий выполнение этого требования.

Листинг 12б. Регистрация функций имплементирующих методы интрефейса gpiobus.

```
static device_method_t gpiobus_methods[] = {
        …
        /* GPIO protocol */
        DEVMETHOD(gpiobus_acquire_bus,  gpiobus_acquire_bus),
        DEVMETHOD(gpiobus_release_bus,  gpiobus_release_bus),
        DEVMETHOD(gpiobus_pin_set,      gpiobus_pin_set),

        …
        /* Device interface */
        DEVMETHOD(device_probe,         gpiobus_probe),
```

```
        DEVMETHOD(device_attach,        gpiobus_attach),
        DEVMETHOD(device_detach,        gpiobus_detach),
        …

        DEVMETHOD_END
};
```

В приведенном выше листинге производится привязка (фактически — заполнение указателей) методов, имена которых указанны в первом параметре, с имплементирующими функциями, имена которых заданны во втором параметре макро **DEVMETHOD**. Очень часто имя метода и имя функции совпадают, поэтому мы видим в обоих параметрах макро **DEVMETHOD** одно и то же значение. Однако, это требование не обязательное и имя имплементирующей функции может быть любым. Макро **DEVMETHOD_END** является признаком конца массива (фактически — это запись вида **`{ NULL, NULL }`**).

4.1.4. Регистрация модуля (драйвера) в Nexus

Для регистрация нового объекта заданного класса, с подготовленными методами и интерфейсом, служит макро **DRIVER_MODULE()** — помощью него осуществляется связка нового объекта с деревом Nexus и устанавливается предок данного объекта. Здесь же резервируется память под «контекст» и устанавливается процедура-обработчик внутренних событий, которые Nexus распространяет эпизодически по своему дереву сигнализируя объектам о событиях как-то загрузка или выгрузка модуля, вход в режим «сна» и т.д.

Для объектов, которые имеют соединения с несколькими другими объектами, предоставляется макро **MODULE_DEPEND()** указывающее системе на существование таких зависимостей. Напомню, что операционная система может автоматически подгружать недостающие модули если у неё имеется информация о зависимостях.

Листинг 12в. Регистрация модуля gpiobus в системе.

```
driver_t gpiobus_driver = {
        "gpiobus",
        gpiobus_methods,
        sizeof(struct gpiobus_softc)
};

DRIVER_MODULE(gpiobus, gpio, gpiobus_driver, 0, 0);
```

В приведенном выше листинге 12в регистрируется класс **gpiobus** который является наследником класса **gpio**, его методы передаются в оберточной структуре **gpiobus_driver**. Эта же структура содержит строковое имя драйвера **"gpiobus"** (под этим именем драйвер будет виден в дереве Nexus) и размер структуры контекста **gpiobus_softc** получаемый с помощью оператора **sizeof()**. Сама структура контекста для данного модуля очень объемная, она определена в заголовочном файле **sys/dev/gpio/gpiobusvar.h** и здесь мы её рассматривать не будем.

4.1.5. Сборочный скрипт (Makefile)

Процесс сборки модуля из исходных кодов весьма запутанный процесс, одна только компиляция интерфейсных M-файлов требует вызова различных скриптов и утилит. Чтобы упросить жизнь разработчикам драйверов и нивелировать возникновение ошибки, а также чтобы убрать зависимость сборки модулей от версии ядра ОС и версии системы сборки (которая может претерпевать изменения от версии к версии), разработчики операционной системы FreeBSD подготовили множество шаблонных **.mk**-скриптов которые можно

подключать в Makefile своего проекта с помощью оператора **.include** и решать типовые задачи сборки. Системная утилита **make** настроена таким образом, что она находи **.mk**-скрипты в системном подкаталоге **/usr/share/mk/**.

Для сборки модуля ядра в этом подкаталоге имеется скрипт с именем **bsd.kmod.mk**, перед вызовом ему требуется задать несколько входных параметров: в переменной **SRCS** указать список файлов исходного кода (в том числе М-файлов) участвующих в сборке, а переменной **KMOD** указать имея результирующего бинарного файла (**.ko**-файла) модуля. В файле **/usr/share/mk/bsd.README** дано небольшое описание всех имеющихся **.mk**-скриптов и их входных параметров.

Таким образом сборка модуля вызывается простым запуском команды **make**, в листинге 12г приведен типовой Makefile для сборки модуля ядра **foo**.

Листинг 12г. Типовой Makefile для сборки модуля foo.

```
# Build script for FOO driver module
#
# Required for Newbus drivers: generates device_if.h and bus_if.h

KMOD= foo
SRCS= foo.c foo_if.m device_if.h bus_if.h

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

На странице руководства **man 9 module** представлена более подробная информация о структуре моделей ядра, там же даются ссылки на другие страницы руководства с детальным описанием используемых макро и их параметров. В базовой поставке ОС FreeBSD имеются примеры (исходные коды на языке Си) готовых модулей для изучения, находятся они в подкаталоге **/usr/share/examples/kld/**.

На этом закончим наш краткий обзор структуры модулей ядра в ОС FreeBSD и приступим к разработке драйвера. Для желающих более плотно погрузиться в тему, рекомендую к прочтению открытую книгу «FreeBSD Device Drivers From First Steps to Kernel Mastery» за авторством Edson Brandi, она же «FDD». Книга свободно доступна в PDF формате на Github.

4.2. Структура разрабатываемого модуля gpiopwrb

Если сократить до двух строк описание алгоритма, который приведен в начале главы 4, то его можно выразить следующим образом: модуль ядра **gpiopwrb** при инициализации стартует отдельный поток исполнения **kthread** в котором непрерывно, с заданным интервалом времени, опрашивает пин GPIO и, если обнаруживает что состояние пина изменилось, то делает вызов ACPI метода **_EVT**. Отсюда следует, что модуль **gpiopwrb** должен содержать следующее:

- имплементацию стандартного набора методов **device_probe**, **device_identify**, **device_attach** и **device_detach** представленных функциями **gpiopwrb_probe()**, **gpiopwrb_identify()**, **gpiopwrb_attach()** и **gpiopwrb_detach()** соответственно;
- процедуру **gpiopwrb_daemon()** для опроса пина GPIO исполняемую в отдельном потоке **kthread**;
- процедуру **gpiopwrb_call_acpi()** для вызова ACPI метода **_EVT**;
- и структуру **struct gpiopwrb_softc** содержащую контекст.

Так как модуль **gpiopwrb** будет представлять собой конечный объект (leaf) в длинной цепочке Nexus, что не предполагает экспорта каких либо методов для взаимодействия с другими объектами, то необходимости в создании своего M-файла с описанием интерфейса для этого модуля у нас не возникает. Соответственно не возникает и необходимости экспортировать какие-то дополнительные методы в Nexus, кроме выше перечисленных.

Наш модуль **gpiopwrb** для своей работы требует доступ к API «GPIO Bus» предоставляемый модулем **gpiobus**, который, в свою очередь, использует низкоуровневый API предоставляемый модулем **amdgpio**. Также наш модуль будет использовать API для работы с ACPI предоставляемый модулем **acpi**. Таким образом, имеем три зависимости от других модулей, который следует внести в код описания модуля.

Основываясь на этой информации о структуре модуля, уже сейчас можно описать формальную часть кода выполняющую регистрацию класса **gpiopwrb** и одноименного драйвера в дереве Nexus. Приведенный ниже листинг 13а содержит весь формальный код для разрабатываемого модуля.

Листинг 13а. Регистрация класса и модуля gpiopwrb в Nexus.

```
# STEP-1
static device_method_t gpiopwrb_methods[] = {
       DEVMETHOD(device_identify, gpiopwrb_identify),
       DEVMETHOD(device_probe, gpiopwrb_probe),
       DEVMETHOD(device_attach, gpiopwrb_attach),
       DEVMETHOD(device_detach, gpiopwrb_detach),
       DEVMETHOD_END
};

# STEP-2
DEFINE_CLASS_0(gpiopwrb, gpiopwrb_driver, gpiopwrb_methods,
       sizeof(struct gpiopwrb_softc));

# STEP-3
DRIVER_MODULE(gpiopwrb, gpiobus, gpiopwrb_driver, 0, 0);

# STEP-4
MODULE_DEPEND(gpiopwrb, gpiobus, 1, 1, 0);
MODULE_DEPEND(gpiopwrb, acpi, 1, 1, 0);
MODULE_DEPEND(gpiopwrb, amdgpio, 1, 1, 0);

# STEP-5
MODULE_VERSION(gpiopwrb, 1);
```

Для удобства описания я разделил выше приведенный код на «шаги» обозначенные как «STEP-1» ... «STEP-5», рассмотрим его более подробно:

На шаге «STEP-1» с помощью макро **DEVMETHOD** производится заполнение массива **gpiopwrb_methods** содержащего список методов нового класса. В нашем случае происходит переопределение интерфейсных методов от класса **device**, который представляет собой стандартный драйвер устройства. Эти методы всегда наследуются по всему дереву Nexus для всех объектов и не требуют явного указания класса-предка.

На шаге «STEP-2» с помощью макро **DEFINE_CLASS_0** производится регистрация нового класса. На вход в это макро передаются: имя нового класса — **gpiopwrb**, имя переменной для хранения описания класса — **gpiopwrb_driver**; имя переменной содержащей список методов — **gpiopwrb_methods**; и размер структуры контекста **gpiopwrb_softc**. Фактически, данное макро производит объявление и заполнение оберточной структуры на которую ссылается переменная **gpiopwrb_driver**. Макро **DEFINE_CLASS_0** объявляет новый класс без базового (без предка). Для объявления нового класса с наследованием интерфейса от одного базового класса служит макро **DEFINE_CLASS_1**. Аналогично

имеется макро для наследования от двух и более классов. Данные макро определены в заголовочном файле **/usr/include/sys/kobj.h**.

На шаге «STEP-3» производится регистрация объекта (модуля) заданного класса в дереве устройств Nexus. Для этого в макро **DRIVER_MODULE** передаются: имя нового объекта — **gpiopwrb**; имя объекта «шины» в иерархии Nexus к которому будет подключен новый объект — **gpiobus**; переменная ссылающая на оберточную структуру с описанием класса — **gpiopwrb_driver**; ссылка на функцию обработчика событий для этого объекта и произвольный параметр для этого функции. В нашем случае последние два параметра не используются и указаны как 0.

На шаге «STEP-4» с помощью макро **MODULE_DEPEND** задается список зависимостей от других модулей. Ядру операционной системы необходимо знать какие модули требуется автоматически подгрузить когда данный модуль регистрируется в системе. В макро **MODULE_DEPEND** передаются: имя текущего модуля — **gpiopwrb**; имя модуля от которого зависит данный модуль (в нашем случае это модули **gpiobus**, **acpi** и **amdgpio**); минимальный номер версии зависимого модуля, желаемый номер версии и максимальный номер версии. В качестве минимально и желаемого номера версии устанавливаются 1, а в качества максимального — 0. Такой набор параметров, по сути, отключает проверку версий.

На шаге «STEP-5» задается номер версии текучего модуля с помощью макро **MODULE_VERSION**.

4.3. Контекст модуля gpiopwrb (struct gpiopwrb_softc)

Теперь попробуем описать структуру контекста **gpiopwrb_softc**. Так как наш модуль является драйвером устройства, то нам необходимо сохранить указатель на структуру типа **device_t** передаваемую от фреймворка «Newbus» при инициализации (вызове метода **device_attach**) — это структура по которой наш драйвер идентифицируется в иерархии Nexus, сохраним указатель на него в переменной **main_dev**.

Наш драйвер будет подключаться к шине GPIO модуля **gpiobus** вызовом оберточного метода **BUS_ADD_CHILD()**, для чего нам потребуется еще два указателя на эту же структуру **device_t**: первый будет ссылаться на устройство предка, второй — на дочернее (подключенное) устройство возвращаемое методом **BUS_ADD_CHILD()**. Заведем в контексте для этих указателей две переменные: **gpiobus_dev** и **child_dev**. Обычно, при подключении к шине, в качестве дочернего устройства возвращается текущее устройство (то есть **child_dev** совпадает с **main_dev**), но это справедливо только в том случае, если драйвер инициирует подключение одного «инстанса» (одной копии объекта). Если драйвер делает два и более подключений к шине, то на каждую копию будет сформирована своя копия структуры **device_t**, а значит со второй копии указатель **child_dev** будет уже отличаться. Не смотря на то, что наш драйвер будет инстранциировать только одну копию, для проформы будем хранить указатели в разных переменных. Кнопка «Power» на ноутбуке у нас одна, но вдруг мы заходим добавить поддержку кнопки «Sleep» ? ;-)

Следующее что нам потребуется, это хендлер на GPIO пин 0 который мы запросим у **gpiobus**. Этот хендлер имеет тип **gpio_pin_t**, сохраним его в переменной **gpio_pin**. Запрос GPIO пина у шины **gpiobus** осуществляется специализированным методом **gpio_pin_get_by_bus_pinnum()**, на вход в который передается номер пина. Сохраним номер пина в переменной **pin_num** типа **int**. О том, откуда наш модуль узнает номер GPIO пина мы поговорим чуть позже, сейчас можно задать его значение по-умолчанию с помощью константы **GPIO_PWRB_PIN_NUM** и приравнять к нулю.

Аналогично нам потребуется переменная хендлер **acpi_handle** типа **ACPI_HANDLE** для вызова метода ACPI. Этот хендлер нам придется запросить с помощью метода **AcpiGetHandle()** у шины **acpi**, на вход которого передается путь в дереве ACPI Namepath

представляющий собой 0-терминированную строку ASCII символов. Для сохранения пути введем в контекст переменную **acpi_path** типа массив типов **char**. Сам путь к методу ACPI может отличаться на разных машинах и нам потребуется предусмотреть механизм передачи его в модуль перед загрузкой (аналогично номеру GPIO пина). Сохраним строку с ACPI путём по-умолчанию в константу **GPIO_PWRB_ACPI_PATH** и приравняем к **"_SB.GPIO._EVT"**.

Наш модуль предусматривает запуск потока исполнения **kthread**, для этого в контексте необходимо зарезервировать место под хендлер поток. Будем сохранять этот хендлер в переменной **daemon** типа **struct thread ***. Также нам потребуется переменная состояния потока **thread_terminate** типа **int** для его корректного завершения. Процесс запуска и останова потока обсудим далее при реализации функций **gpiopwrb_attach()** и **gpiopwrb_detach()**.

Ну и последнее что обязательно должно присутствовать в контексте для много-поточного модуля это мьютекс для разграничения доступа к самой структуре контекста. Для этого введем в контекст переменную **thread_mtx** типа **struct mtx**.

Еще один не решенный вопрос состоит в том, где хранить заполненную структуру контекста **gpiopwrb_softc**. Напомню, что у нас есть два способа: объявить глобальную статическую переменную и использовать динамическое выделение памяти. На мой взгляд структура **gpiopwrb_softc** достаточно мала по размеру чтобы её размещать в динамической области памяти где минимальный объём выделяемого блока составляет 2^N байт, поэтому просто объявим глобальную переменную **gpiopwrb_softc_sc** типа **struct gpiopwrb_softc**. Ниже в листинге 13б дано определение структуры контекста **struct gpiopwrb_softc** и объявление статической переменной для её хранения.

Листинг 13б. Объявление структуры контекста и глобальной переменной для неё.

```
#define GPIO_PWRB_PIN_NUM       0                // Default pin number
#define GPIO_PWRB_ACPI_PATH     "\\_SB.GPIO._EVT" // Default ACPI method to call
#define ACPI_PATH_SIZE          32               // Buf size for ACPI pathname

static struct gpiopwrb_softc {
        device_t main_dev;      /* this device */
        device_t gpiobus_dev;   /* gpiobus parent */
        device_t child_dev;     /* gpiobus child dev (me again) */
        gpio_pin_t gpio_pin;    /* GPIO in handler */
        ACPI_HANDLE acpi_handle;
        struct mtx thread_mtx;
        int thread_terminate;   /* 0 - init, 1 - running, 2 - terminated */
        int pin_num;            /* Power Button pin num, usually 0 */
        struct thread *daemon;
        char acpi_path[ACPI_PATH_SIZE];
} gpiopwrb_softc_sc;
```

4.4. Инициализация и деинициализация модуля gpiopwrb

4.4.1. Реализация метода device_identify

Перед тем, как мы приступим к реализации функций **gpiopwrb_probe()**, и **gpiopwrb_attach()**, отдельно следует рассказать про функцию **gpiopwrb_identify()** реализующую метод **device_identify**, а так же про необходимость этого метода в контексте нашего драйвера. Как мы знаем, одна из задач подсистемы ACPI состоит в том, чтобы снабдить операционную систему списком всех устройство и их настроек в объеме необходимом для выполнения инициализации. В ACPI, в том числе, предусматривается передача в систему конфигурационной информации об устройствах подключенных к линиям GPIO. Для этого в «firmware» коде реализуется управляющий метод **_SB_.GPIO._AEI**, запросив который операционная система получит в ответ массив (тип **Buffer**) с описанием GPIO устройств и используемых ими ресурсов. Как отмечалось выше в главе 3.8, в ОС FreeBSD имеется модуль ядра **gpioaei** задача которого состоит в том, чтобы запросить у «firmware» конфигурационную информацию об устройствах находящихся на линиях GPIO и инициировать их перенумерацию (вызов методов **device_probe** и **device_attach** у соответствующих драйверов). Как было показано в той же главе, в нашем случае при загрузке модуля **gpioaei** выдается ошибка вида "**gpio_aei0: Cannot allocate an IRQ for pin 0x0**", что препятствует дальнейшей инициализации, а значит никаких методов **device_probe** и **device_attach** для нашего драйвера вызвано автоматически не будет (иначе, наш драйвер был бы не нужен).

Для того, чтобы у нашего драйвера **gpiopwrb** были вызваны методы инициализации, мы должны снабдить его еще одним методом — методом идентификации **device_identify**. Наличие такого метода укажет родительскому классу на то, что к нему подключено еще одно «неопознанное» устройство, а вся процедура «опознания» произойдет внутри этого метода. Подробно эта процедура описана на странице руководства **man 9 DEVICE_IDENTIFY**. Вкратце, для нашего драйвера она состоит из следующих шагов:

1. Получить из внешнего источника информацию о линии GPIO (номере пина) к которой подключена кнопка «Power»;
2. Получить из внешнего источника информацию о том, какой управляющий ACPI метод необходимо вызывать для сигнализации о смене состояния этой линии GPIO;
3. С помощью функции **BUS_ADD_CHILD()** выполнить подключение к родительскому классу (к шине «GPIO bus»);
4. Запомнить возвращаемый идентификатор **child_dev** типа **device_t** в контекстной структуре **gpiopwrb_softc_sc**.

Тут возникает резонный вопрос: а что может служить внешним источником информации для драйвера ? Каким образом системный администратор должен сообщить драйверу **gpiopwrb** на какой именно линии находится кнопка «Power» на его машине ?

Для таких случаев в ОС FreeBSD предусмотрена система «подсказок» (**hints**), являющихся частью переменных окружения ядра. Эти переменные с подсказками очень похожи на системные переменные управляемые утилитой **sysctl**, но не пересекаются с ними. Для управления подсказками используется утилита **kenv** (страница руководства доступна по **man 1 kenv**), она позволяет читать и устанавливать подсказки «на ходу», также их можно заносить в файл конфигурации загрузчика **/boot/loader.conf**. Начинаются переменные подсказок со сроки символов "**hint.XXXX.Y.**", где XXXX это имя модуля (драйвера) для которого предназначена подсказка, а Y — номер инстанса, начиная с 0. Иногда эти подсказки

путают с «tunables» системными переменными ядра, так как они тоже могу находится в файле конфигурации загрузчика, но повторюсь — это не одно и то же!

Для нашего драйвера **gpiopwrb** мы добавим распознавание следующих подсказок:

- **hint.gpiopwrb.0.pin_num=0** — будет содержать число представляющее номер пина GPIO с кнопкой «Power»;
- **hint.gpiopwrb.0.acpi_path=""** — будет содержать строку символов к вызываемому ACPI методу.

Для чтения подсказки ядро операционной системы предоставляет функцию **kern_getenv()** на вход в которой передается имя переменной подсказки (указатель **const char ***), а на выходе она выдает указатель типа **char*** на строку символов со значением переменной. Этот возвращаемый указатель после использования необходимо высвободить вызовом **freeenv()**. Фукнция **kern_getenv()** возвращает **NULL** есть переменной окружения с таким именем не обнаружено. На случай, если системный администратор не позаботился об установки соответствующих подсказок мы будем использовать значения по умолчанию задаваемые константами **GPIO_PWRB_PIN_NUM** и **GPIO_PWRB_ACPI_PATH**.

Про функцию подключения к **BUS_ADD_CHILD()** я уже упоминал в главе 4.3 при описании структуры контекста. Она принимает на вход четыре параметра: указатель на текущее устройство **dev** типа **device_t**, параметр **order** определяющий в какой момент будут вызываться методы инициализации, указатель на строку **const char *name** задающую имя нашему устройству и целочисленный параметр **unit** — номер инстанса данного устройства (объекта) в системе. В нашем случает для параметра **name** мы укажем строку символов "**gpiopwrb**", для параметра **order** укажем **0** — последовательность вызова не имеет значения, а для параметра **unit** укажем **-1** чтобы драйвер шины сам назначил нам свой номер инстанса. В контексте данного драйвера этот номер не важен, но если нам захочется добавить две управляемые по GPIO кнопки, то придется эти номера запомнить и обработать.

У нас все готово для реализации функции идентификации для нашего драйвера. Ниже в листинге 14а приведен весь его код на языке Си.

Листинг 14а. Реализация идентификационного метода device_identify.

```
static void
gpiopwrb_identify(driver_t *driver, device_t parent)
{
        dprintf("gpiopwrb: identify\n");

        struct gpiopwrb_softc *sc = &gpiopwrb_softc_sc;

        KASSERT(sc != NULL, ("softc is NULL."));

        if (sc->child_dev) {
                dprintf("gpiopwrb: child_dev = %p already set for parrent = %p\n",
                        sc->child_dev, parent);
                return;
        }

        sc->child_dev = BUS_ADD_CHILD(parent, 0, "gpiopwrb", -1);
        sc->gpiobus_dev = parent;

        /* Fetch GPIO pin number from Hints */

        char hint_str[32];
        char *hint_value;

        snprintf(hint_str, 32, "hint.%s.%d.pin_num",
            device_get_name(sc->child_dev),
```

```
            device_get_unit(sc->child_dev));

        if ((hint_value = kern_getenv(hint_str)) != NULL) {
                sc->pin_num = strtol(hint_value, NULL, 0);
                freeenv(hint_value);
        }
        else
                sc->pin_num = GPIO_PWRB_PIN_NUM;

        dprintf("gpiopwrb: new child_dev = %p for parrent = %p, main_dev = %p, "
                "pin_num = %d, %s = %s\n",
                sc->child_dev, parent, sc->main_dev, sc->pin_num, hint_str, hint_value);

        /* Fetch ACPI path from Hints */

        snprintf(hint_str, 32, "hint.%s.%d.acpi_path",
            device_get_name(sc->child_dev),
            device_get_unit(sc->child_dev));

        if ((hint_value = kern_getenv(hint_str)) != NULL) {
                strlcpy(sc->acpi_path, hint_value, ACPI_PATH_SIZE);
                freeenv(hint_value);
        }
        else
                strlcpy(sc->acpi_path, GPIO_PWRB_ACPI_PATH, ACPI_PATH_SIZE);

        dprintf("gpiopwrb: acpi path hint = %s, value = %s\n",
                hint_str, sc->acpi_path);

        device_set_softc(sc->child_dev, sc);
}
```

В приведенном выше лисntинге используются вспомогательные функции **device_get_name()** и **device_get_unit()** для получения от системы имени драйвера и его номера инстанса. Вспомогательная функция **device_set_softc()**, как можно догадаться, позволяет ассоциировать (и запомнить) в системе контекстную структуру, указатель которой находится в переменной **sc**, с текущим устройством в переменной **sc->child_dev**. Далее мы сможем запрашивать у системы указатель на контекст для нашего драйвера с помощью вспомогательной функции **device_get_softc()**. Хотя, этого можно и не делать, так как контекст находится в глобальной переменной **gpiopwrb_softc_sc** имя которой всегда известно данному модулю.

4.4.2. Реализация метода device_probe

Так как наш драйвер не выполняет каких либо проверок на совместимость с устройством, то реализация метода **device_probe** будет очень лаконичной, его полный листинг приведен ниже в Листинге 14б. Всё что нам потребуется сделать это проверить поддерживается ли системой ACPI, после чего установить человеко-читаемое название драйвера (оно будет отображаться пользователю в **dmesg** при отработки метода **device_attach**) и завершиться с успешным кодом **BUS_PROBE_DEFAULT**.

Для проверки доступности подсистемы ACPI в ядре ОС FreeBSD предусмотрен механизм позволяющий отключать поддержку ACPI для отдельных её частей и модулей. Реализуется это следующим образом: модуль, в реализации метода инициализации, вызывает вспомогательную функцию **acpi_disabled()** передавая на вход указатель на строку символов содержащую название этого модуля (или группы модулей) и, в зависимости от возвращаемого результата, продолжает работу или завершается с ошибкой **ENXIO**. Внутри функции **acpi_disabled()** производится проверка на вхождение переданной строки с названием модуля в строку определяемую переменной окружения ядра (**kenv**) с именем **debug.acpi.disabled**. Таким образом, администратор системы может временно и выборочно отключать поддержку ACPI для некоторых частей системы добавив эту переменную окружения в файл конфигурации загрузчика **/boot/loader.conf**.

Листинг 14б. Реализация метода device_probe.

```
static int
gpiopwrb_probe(device_t dev)
{
        if (acpi_disabled("gpiopwrb")) {
                dprintf("pwrb: ACPI is disabled\n");
                return (ENXIO);
        }

        device_set_desc(dev, "GPIO Power Button Monitor");

        dprintf("gpiopwrb: probed\n");

        return (BUS_PROBE_DEFAULT);
}
```

4.4.3. Реализация метода device_attach

После того как пройдет идентификация модуля, операционная система осуществит вызов метода **device_attach**. В этом методе модуль должен запросить (аллоцировать) все необходимые ему ресурсы чтобы больше не возвращаться к этому вопросу в процессе работы. Драйверу **gpiopwrb** потребуются следующие ресурсы:

1. Хендлер типа **ACPI_HANDLE** для вызова управляющего метода ACPI по заданному пути. Получить этот хендлер можно вызовом метода **AcpiGetHandle()** предоставляемого модулем **acpi**. На вход этого метода передается ACPI путь ранее полученный от подсказок (hints) и сохраненный в контекстной структуре.
2. Доступ к линии GPIO осуществляется через хендлер типа **gpio_pin_t**, получить который можно вызовом метода **gpio_pin_get_by_bus_pinnum()** предоставляемого объектом шины GPIO (**gpiobus**).
3. Мьютекст для разграничения доступа к контекстной структуре. Получить новый мьютекс можно с помощью сервисной функции **mtx_init()**, на вход в которую передается указатель на структуру типа **struct mtx**.

Помимо ресурсов нам потребуется запустить поток **kthread** для выполнения основной работы — регулярного опроса состояния линии GPIO. Запуск нового потока осуществляется вызовом сервисной функции **kthread_add()**. Эта функция принимает на вход минимум семь параметров:

- **func** — указатель на функцию тела исполняемого потока. В нашем случае это будет указатель на функцию **gpiopwrb_daemon()** реализацию которой мы обсудим немного позже.
- **arg** — указатель на произвольный аргумент передаваемый в функцию **func** при запуске потока. В нашем случае это указатель на структуру контекста. Его предварительно потребуется запросить у системы через сервисную функцию **device_get_softc()**.
- **procp** — указатель на хэндлер описания процесса к которому будет добавлен новый поток. Мы будет указывать **NULL**, что означает что наш поток будет добавлен к системному процессу **proc0** внутри ядра ОС.
- **tdprt** — указатель на хенлер потока (**struct thread***), в эта переменная будет заполнена операционной системой.
- **flags** — битовое поле определяющее состояние потока в момент запуска и некоторые его свойства. В нашем случае будем передавать **0**, чтобы система установила значения по умолчанию.

- **procname** — указатель на строку символов (**const char ***) содержащую человеко-читаемое имя потока, отображаемое системному администратору. Мы укажем здесь строку символов "**gpiopwrb**".

Согласно требованиям, если в процессе выделения ресурсов в методе **device_attach** возникает ошибка препятствующая дальнейшей работы модуля, то метод должен завершиться с кодом **ENXIO**. Иначе, в случае успеха, метод должен вернут код **0**.

Исходя из изложенного выше, у меня получился представленный на Листинге 14в код функции **gpiopwrb_attach()** реализующий метод **device_attach** в модуле **gpiopwrb**.

Листинг 14в. Реализация метода device_attach.

```
static int
gpiopwrb_attach(device_t dev)
{
        int error;
        struct gpiopwrb_softc *sc = device_get_softc(dev);

        KASSERT(sc != NULL, ("softc is NULL."));

        sc->main_dev = dev;

        ACPI_STATUS acpi_status = AcpiGetHandle(NULL, sc->acpi_path,
                &sc->acpi_handle);

        if (ACPI_FAILURE(acpi_status)) {
                printf("gpiopwrb: failed to get ACPI handle for %s: %s\n",
                        sc->acpi_path,
                        AcpiFormatException(acpi_status));
                return (ENXIO);
        }

        dprintf("gpiopwrb: attached to gpiobus = %s, child = %s, main_dev = %s\n",
                device_get_name(sc->gpiobus_dev),
                device_get_name(sc->child_dev),
                device_get_name(sc->main_dev));

        error = gpio_pin_get_by_bus_pinnum(sc->gpiobus_dev,
                sc->pin_num, &sc->gpio_pin);

        dprintf("gpiopwrb: gpio_pin_get_by_bus_pin_num error = %d, "
                "gpio_pin->pin = %d, gpio_pin->flags = %x, "
                "gpio_pin->dev = %p, devname = %s\n",
                error,
                sc->gpio_pin->pin, sc->gpio_pin->flags, sc→gpio_pin→dev,
                device_get_name(sc->gpio_pin->dev));

        if (error) {
                printf("gpiopwrb: failed to get GPIO pin %d value, gpiobus = %p, "
                        "child = %p, error = %d\n",
                        sc->pin_num, sc->gpiobus_dev, sc->child_dev, error);
                return (ENXIO);
        }

        sc->thread_terminate = 0;

        mtx_init(&sc->thread_mtx, "pwrb thread mtx", NULL, MTX_DEF);

        error = kthread_add(gpiopwrb_daemon, sc, NULL,
                &sc->gpiopwrb_daemon_thread,
                0, 0, "pwrb_daemon");

        if (error) {
                printf("gpiopwrb: failed to start kthread, error = %d\n", error);
                return (error);
        }

        device_set_softc(dev, sc);
```

```
        printf("gpiopwrb: attached to gpio pin %d\n", sc->pin_num);

        return (0);
}
```

В приведенном выше коде, перед запуском потока, производится установка значения **0** во внутреннюю сигнальную переменную **thread_terminate**. Это говорит о том, что поток запущен. Внутри функции **gpiopwrb_daemon()** (тело потока) будем регулярно, в основном цикле, проверять эту переменную на значение отличное от **0**. При достижении этого условия, цикл будет остановлен, функция завершится и поток остановится. Такой механизм понадобиться нам для того, чтобы сигнализировать потоку о корректном завершении при выгрузке модуля (при вызове системой метода **device_detach**).

4.4.4. Реализация метода device_detach

Как упоминалось в главе 4.1.2 метод **device_detach**, по сути, должен выполнить действия обратные тем, которые выполняются в методе **device_attach**. Для нашего драйвера **gpiopwrb** это означает следующую последовательность действий:

1. Инициировать останов потока **gpiopwrb_daemon()**.
2. Дождаться окончания его исполнения.
3. Высвободить хэндлеры **acpi_handle**, **gpio_pin** и **thread_mtx**.
4. Завершиться с кодом **0**.

Начнем с остановки потока. Тут надо отметить, что API **kthread** ядра ОС FreeBSD не предусматривает принудительного завершения потока из вне (из другого потока). Вместо этого, поток должен самостоятельно завершиться когда посчитает нужным, а перед самым выходом из его главной функции он обязан вызвать сервисную функцию **kthread_exit()**, которая никогда не возвращается. Если этого не сделать, то ядро может запаниковать. Выше мы договорились, что для управления потоком введем сигнальную переменную **thread_terminate**, расположенную в структуре контекста. Если поток обнаружит, что значение этой переменной не равно **0**, то он самостоятельно закончит своё выполнение и вызовет функцию **kthread_exit()**. Поэтому, в реализации метода **device_detach** для инициирования остановки потока следует просто увеличить значение переменной **thread_terminate** на **1**. Но тут есть нюанс: а что, если поток находится в приостановленном состоянии ? В некоторых случаях операционная система может остановить исполнение потока (усыпить его) на неопределенное время. Попытка ожидания завершения потока приостановленного системой, с большой вероятностью, приведет к блокировке и «зависанию» процесса выгрузки нашего модуля. Чтобы избежать такой участи, перед тем как сигнализировать потоку об остановке, мы принудительно разбудим его вызовом сервисной функции **kthread_resume()**. Попытка разбудить уже разбуженный поток никакого вреда не принесет.

Ожидание завершения потока. Тут процесс обратный — после того, как поток вышел из цикла он должен передать сигнал в обратную сторону, для этого он просто увеличивает содержимое переменной **thread_terminate** на 1 (еще раз). В методе **device_detach** мы будем в цикле ожидать когда переменная **thread_terminate** примет значение более 1 — это и будет сигналом того, что поток завершился и больше не использует ресурсы контекста.

Разграничение доступа к переменной **thread_terminate**. Каждый раз когда в коде потока или в коде **device_detach** мы будем изменять или проверять содержимое переменной **thread_terminate**, нам следует делать это эксклюзивно — то есть так, чтобы предотвратить ситуацию одновременного чтения или изменения содержимого переменной в двух разных потоках. Если такого разграничения не сделать, то в переменной **thread_terminate** может

образоваться «мусорное» значение и вся наша логика управления будет нарушена. Для эксклюзивного доступа к ресурсу (переменной) используется технология мьютексов (mutex). В ядре ОС FreeBSD она реализуется двумя сервисными функциями: **mtx_lock()** и **mtx_unlock()** - захватывающие и высвобождающие ресурс соответственно. О том, как работает этот механизм можно прочесть на странице руководства доступного по команде **man 9 mutex**. Воспользуемся этим и «обвернем» доступ к переменной **thread_terminate** двумя этими функциями.

Высвобождение хендлеров. Как выяснилось, высвобождение хендлера **acpi_handle** не требуется. Во всяком случае я не нашел, ни в документации, ни в коде ACPICA, механизмов для этого. А вот хендлер на GPIO пин содержащийся в переменной **gpio_pin** высвободить следует, иначе при следующем подключении к этому пину (при следующей загрузке) наш модуль не получит в нему доступ. Для высвобождения GPIO пина воспользуемся методом **gpio_pin_release()** предоставляемого модулем **gpiobus**. Также нам следует высвободить хендлер для более неиспользуемого мьютекса, для чего воспользуемся сервисной функцией **mtx_destroy()**.

Выразив всё выше сказанное на языке Си получим приведенный на Листинге 14г код функции **gpiopwrb_detach()** реализующей метод **device_detach**.

Листинг 14г. Реализация метода device_detach.

```
static int
gpiopwrb_detach(device_t dev)
{
	struct gpiopwrb_softc *sc = device_get_softc(dev);

	KASSERT(sc != NULL, ("softc is NULL."));

	kthread_resume(sc->gpiopwrb_daemon_thread);

	/* signal kthread to terminate */
	mtx_lock(&sc->thread_mtx);
	sc->thread_terminate++;
	mtx_unlock(&sc->thread_mtx);

	dprintf("gpiopwrb: waiting thread termination...\n");

	for (;;) {
		mtx_lock(&sc->thread_mtx);
		int tmp = sc->thread_terminate;
		mtx_unlock(&sc->thread_mtx);

		if (tmp > 1)
			break;

		pause("wterm", hz / 100);
	}

	gpio_pin_release(sc->gpio_pin);

	mtx_lock(&sc->thread_mtx);
	mtx_destroy(&sc->thread_mtx);

	dprintf("gpiopwrb: detached from gpio pin %d\n", sc->pin_num);

	return (0);
}
```

В приведенном выше коде используется сервисная функция **pause()** в для приостановки исполнения текущего потока на **hz / 100** квантов времени, то есть на 10 миллисекунд. Это необходимо в цикле для того, чтобы текущий поток не загружал без толку отведенный ему процессор, а также для того, чтобы поток в функции **gpiopwrb_daemon()** имел шанс захватить мьютекс.

После того, как цикл ожидания завершится, происходит разрушение запрошенных ранее ресурсов, в том числе мьютекса. И тут скрыт еще один интересный момент — если поток **gpiopwrb_daemon()** слегка запоздает, то может случиться ситуация когда **mtx_destroy()** сработает на мьютексе который занят в данный момент. Обычно такие ситуации в многозадачных и многопроцессорный операционных системах приводят к краху. Чтобы обойти эту ситуацию, перед вызовом **mtx_destroy()** нам потребуется захватить мьютекс еще раз. В руководстве на функцию **mtx_destroy()** имеется следующее интересное замечание:

> *It is permissible to have a single hold count on a mutex when it is destroyed. It is not permissible to hold the mutex recursively, or have another thread blocked on the mutex when it is destroyed.*

Это позволяет нам занять мьютекс, таким образом дождавшись завершения его использования в **gpiopwrb_daemon()**, и тут же разрушить его. Красота!

4.5. Реализация потока gpiopwrb_daemon

Итак, мы постепенно подобрались к самому главному участку нашего драйвера — реализации поточной функции в которой происходит вся интересующая нас логика драйвера. Напомню, что нам требуется регулярно опрашивать состояние линии GPIO, хенлер на которую храниться в контекстной структуре в поле **gpio_pin**, и, если статус изменился, то вызывать управляющий ACPI метод, хендлер на который сохранен в поле **acpi_handle**.

Для начала опишем нашу встроенную сервисную функцию **gpiopwrb_call_acpi()** которая собственно производит вызов управляющего ACPI метода, её текст приведен ниже в Листинге 15а. Всё, что она делает — вызывает метод **AcpiEvaluateObject()** предоставляемый модулем **acpi**, предварительно подготовив массив входных параметров.

Листинг 15а. Сервисная функция для вызова управляющего ACPI метода.

```
static void
gpiopwrb_call_acpi(struct gpiopwrb_softc *sc)
{
        ACPI_STATUS status;
        ACPI_OBJECT args[2];
        ACPI_OBJECT_LIST arglist;

        memset(args, 0, sizeof(args));
        memset(&arglist, 0, sizeof(arglist));

        /*
         * ACPI event \_SB.GPIO._EVT usually takes two arguments:
         * arg0 - event source number (GPIO pin number) that has
         *        status changed.
         *
         */

        args[0].Type = ACPI_TYPE_INTEGER;
        args[0].Integer.Value = (ACPI_INTEGER) (INT64) sc->pin_num;
        arglist.Count = 1;
        arglist.Pointer = args;

        status = AcpiEvaluateObject(sc->acpi_handle, NULL, &arglist, NULL);

        if (ACPI_FAILURE(status))
                printf("gpiopwrb: failed to call AcpiEvaluateObject: %s\n",
                        AcpiFormatException(status));
}
```

Метод **AcpiEvaluateObject()** определен в компоненте ACPICA в файле **sys/contrib/dev/acpica/include/acpixf.h** следующим образом:

```
ACPI_STATUS
AcpiEvaluateObject (
    ACPI_HANDLE             Object,
    ACPI_STRING             Pathname,
    ACPI_OBJECT_LIST        *ParameterObjects,
    ACPI_BUFFER             *ReturnObjectBuffer)) ;
```

Как можно догадаться из названия, метод **AcpiEvaluateObject()** предназначен для исполнения функции закрепленной за объектом на который ссылается хэндлер **Object** (первый параметр), плюс смещение пути задаваемое вторым параметром **Pathname**. Если конечным объектом пути является управляющий метод ACPI, как в нашем случае, то вызов **AcpiEvaluateObject()** приведет к исполнению этого управляющего метода в AML коде.

AcpiEvaluateObject() предусматривает передачу объекту фиксированного списка параметров. Для этого сначала формируется массив структур типа **ACPI_OBJECT**, каждый

элемент которого представляет собой обертку для передаваемого параметра (см. переменную **args** в Листинге 15а). В нашем случае первым и единственным параметром передается значение номера пина GPIO сохраненного в переменной **sc->pin_num**. Далее, указатель на этот массив, вместе с информацией о его размере, помещается в еще одну оберточную структуру типа **ACPI_OBJECT_LIST** (переменная **arglist**), после чего указатель уже на эту оберточную структуру передается в метод **AcpiEvaluateObject()** третьим параметром. Четвертым параметром передается указатель на буфер типа **ACPI_BUFFER**, он заполняется возвращаемым значением. Так как нас не интересуют данные возвращаемые методом из AML, то в качестве четвертого параметра передаем **NULL**.

Сам метод **AcpiEvaluateObject()** возвращает код статуса исполнения, его можно проверить с помощью макро **ACPI_FAILURE()**, которое принимает значение «true» если была зарегистрирована ошибка исполнения. Описание ошибки в текстовом виде можно получить с помощью метода **AcpiFormatException()** передав на вход статусную переменную.

Ну, а теперь перейдем к рассмотрению главной функции потока, её текст приведен в Листинге 15б. Сразу после запуска потока, в ней производится проверка готовности используемых её переменны контекста, в частности на **NULL** проверяются указатель контекста **sc** (он передается в главною функцию в качестве параметра) и хендрел **sc->gpio_pin**. После этого запоминается текущее (начальное) состояние линии GPIO с помощью метода **gpio_pin_is_active()** и сохраняется в локальную переменную **gpio_state_prev**. Это начальное значение состояния линии будем дальше рассматривать как пассивное, то есть кнопка «Power» не нажата.

Далее, в «вечном» цикле периодически запрашивается текущее состояние линии GPIO и сравнивается со значением сохраненным в **gpio_state_prev**. Если эти значения не совпадают, то фиксируется факт смены состояния (нажатие кнопки «Power») во флаговой переменной **changed**, и если флаг **changed** установлен, то производится вызов сервисной функции **gpiopwrb_call_acpi()**. На этом основная логика заканчивается.

Листинг 15б. Главная функция потока исполнения модуля gpiopwrb.

```
static void
gpiopwrb_daemon(void *arg)
{
	struct gpiopwrb_softc *sc = (struct gpiopwrb_softc *) arg;

	if (sc == NULL) {
		printf("gpiopwrb: bogus softc, thread stopped!\n");
		goto stop_daemon;
	}

	dprintf("gpiopwrb: kthread %p started, sc = %p\n",
		sc->gpiopwrb_daemon_thread, sc);

	if (sc->gpio_pin == NULL) {
		printf("gpiopwrb: gpio pin %d had not been acquired "
			"during attach!\n", sc->pin_num);
		goto stop_daemon;
	}

	bool gpio_state, gpio_state_prev;

	gpio_pin_is_active(sc->gpio_pin, &gpio_state);
	gpio_state_prev = gpio_state;

	dprintf("gpiopwrb: gpio pin %d state = %d\n", sc->pin_num, gpio_state);

	for (;;) {
		int terminate;

		mtx_lock(&sc->thread_mtx);
		terminate = sc->thread_terminate;
```

```
                mtx_unlock(&sc->thread_mtx);

                if (terminate > 0)
                        break;

                gpio_pin_is_active(sc->gpio_pin, &gpio_state);

                bool changed = (gpio_state != gpio_state_prev);

                dprintf("gpiopwrb: poll... gpio pin %d state = %d, changed = %s\n",
                        sc->pin_num, gpio_state, changed ? "YES" : "no");

                if (changed) {
                        printf("gpiopwrb: GPIO pin %d changed, "
                                "emmiting ACPI event %s\n",
                                sc->pin_num, sc->acpi_path);
                        gpiopwrb_call_acpi(sc);
                }

                gpio_state_prev = gpio_state;

                pause_sig("gpiopwrb", GPIO_POLL_TIMEO);
        }

        stop_daemon:

        dprintf("gpiopwrb: kthread %p exited\n", sc->gpiopwrb_daemon_thread);

        mtx_lock(&sc->thread_mtx);
        sc->thread_terminate++;
        mtx_unlock(&sc->thread_mtx);

        kthread_exit();
}
```

Теперь рассмотрим вспомогательную логику потока.

Для передачи системе процессорных тиков (бездействия потока), в цикле используется функция **pause_sig()** - это разновидность функции **pause()**, которая может завершаться «досточно» в случае, если поток получил любой сигнал. Это нужно для того, чтобы при переходе операционной системы в состоянии «suspend» или «shutdown» ядро операционной системы не тратило лишние клоки исполняя предписанный таймаут. Если этого не делать, то при смене состояния системы каждый поток отнимет свой маленький кусочек бесполезного времени, что в сумме сложится в десятки секунд или даже минуты, а пользователь очень не любит ждать.

Как и в методе **device_detach**, в коде потока необходимо использовать мьютекс в местах, где идет обращение к разделяемым переменным контекста. В нашем случае это сигнальная переменная **sc->thread_terminate**. При проверке сигнала его значение перекладывается во временную локальную переменную **terminate**, это позволяет сократить время нахождения мьютекса в заблокированном состоянии, а потока в т. н. «критической секции» кода.

Перед самым завершением главной функции потока производиться вызов сервисной функции **kthread_exit()**, она полностью останавливает поток и снимает его с исполнения. Возврат из **kthread_exit()** невозможен.

4.6. Сборка и тестирование модуля gpiopwrb

Так повелось, что отладка любой программы в UNIX системах невозможна без вывода отладочной информации в консоль, для чего в ядре ОС FreeBSD можно использовать стандартную функцию **printf()** и все её производные. Для того, чтобы отключать лишние отладочные сообщения на стадии компиляции, принято использовать небольшую макро-обертку вокруг этой функции, выглядит она следующим образом:

Листинг 16а. Макро dprintf() для вывода отладочной ифнормации.

```
#ifdef GPIOPWRB_DEBUG
#define dprintf printf
#else
#define dprintf(x, arg...)
#endif
```

Приведенное в листинге 16а макро **dprintf()** ведет себя точно так же как обычная функция **printf()**, но только если предварительно была определена константа **GPIOPWRB_DEBUG** . Если этой константы не задано, то макро **dprintf()** редуцируется до пустого места. Это позволяет нам управлять выводом отладочной информации на стадии сборки модуля, задавая любое значение в макро **GPIOPWRB_DEBUG** . Макро **dprintf()** использовано в коде модуля в тех местах где требуется информировать программиста о происходящем процессе или о состоянии каких-то переменных. Если же драйверу требуется информировать пользователя, а не программиста, о каком-то аварийном состоянии (или о состоянии готовности), то следует использовать обычную функцию **printf()**, она будет присутствовать в коде драйвера всегда. Именно так и сделано во всех в приведенных мной участка кода модуля **gpiopwrb**.

Чтобы добавить в процесс еще немного гибкости, введем в сборочный скрипт **Makefile** переменную окружения **DEBUG**, которая будет устанавливать константу **GPIOPWRB_DEBUG** при вызове компилятора так, как это показано ниже.

Листинг 16б. Вариант сборочного скрипта Makefile с пременной DEBUG.

```
# Build script for GPIO Power Button driver module

KMOD=   gpiopwrb
SRCS=   gpiopwrb.c device_if.h bus_if.h gpio_if.h acpi_if.h opt_acpi.h opt_platform.h

.if defined(DEBUG)
CFLAGS+=       -DGPIOPWRB_DEBUG=1
.endif

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

С таким сборочным скриптом мы теперь можем инициировать сборку модуля с выводом отладочной информации путем вызова команды:

```
rz@butterfly:~/gpiopwrb % make clean && make DEBUG=1

rz@butterfly:~/gpiopwrb % ls -al
...
-rw-r--r--  1 rz  rz  18400 Oct  6 01:39 gpiopwrb.ko
```

Или как обычно, для сборки без отладочного вывода:

```
rz@butterfly:~/gpiopwrb % make clean && make

rz@butterfly:~/gpiopwrb % ls -al
...
-rw-r--r--  1 rz  rz  17488 Oct  6 01:40 gpiopwrb.ko
```

В обоих случаях после сборки мы получим результирующий файл с именем **gpiopwrb.ko**, но файл с отладочным выводом почти на 1 килоайт больше, так как содержит дополнительные строки текста.

Теперь перенесем файл модуля **gpiopwrb.ko**, собранного с отладочным выводом, на целевую машину в подкаталог **/boot/modules/**, загрузим её в «однопользовательский» режим (см. главу 3), и попробуем загрузить модуль. Но перед этим объясним системе, что переходить в режим

«сна» или «перезагрузки» по кнопке «Power» не требуется. Вдруг наш модуль сработает как-то неверно и система моментально уйдет в «suspend» или «shutdown» ? Для этого временно установим переменную ядра **hw.acpi.power_button_state** в значение **NONE**:

```
root@:~ # sysctl hw.acpi.power_button_state=NONE

hw.acpi.power_button_state: S5 -> NONE
```

Теперь выясним, присутствует ли наш модуль в ожидаемом подкаталоге **/boot/modules**:

```
root@:~ # find /boot/ -name gpiopwrb.ko

/boot/modules/gpiopwrb.ko
```

Все в порядке, загружаем модуль командой **kldload**:

```
root@:~ # kldload gpiopwrb.ko

gpio0: <AMD GPIO Controller> iomem 0xfed81500-0xfed818ff irq 7 on acpi0
gpiobus0: <GPIO bus (ACPI-hinted)> on gpio0
gpioc0: <GPIO controller> at pins 0-255 on gpiobus0
gpiobus0: <unknown device> at pins 0,61-62,58-59,2-3,17-18,24,54
gpiopwrb: identify
gpiopwrb: new child_dev = 0xfffff800013d1200 for parrent = 0xfffff800017bd500, main_dev = 0,
pin_num = 0, hint.gpiopwrb.0.pin_num = (null)
gpiopwrb: acpi path hint = hint.gpiopwrb.0.acpi_path, value = \_SB.GPIO._EVT
gpiopwrb: probed
gpiopwrb: probed
gpiopwrb0: <GPIO Power Button Monitor> on gpiobus0
gpiopwrb: attached to gpiobus = gpiobus, child = gpiopwrb, main_dev = gpiopwrb
gpiopwrb: gpio_pin_get_by_bus_pin_num error = 0, gpio_pin->pin = 0, gpio_pin->flags = 0,
gpio_pin->dev = 0xfffff80001612e00, devname = gpio
gpiopwrb: attached to gpio pin 0
gpiopwrb: kthread 0xfffff80102e1e000 started, sc = 0xffffffff831fa198
gpiopwrb: gpio pin 0 state = 1
gpiopwrb: poll... gpio pin 0 state = 1, changed = no
root@:~ #
gpiopwrb: poll... gpio pin 0 state = 1, changed = no
gpiopwrb: poll... gpio pin 0 state = 1, changed = no
gpiopwrb: poll... gpio pin 0 state = 1, changed = no
gpiopwrb: poll... gpio pin 0 state = 1, changed = no
…
```

Сразу видно, что наш модуль загрузился и подтянул за собой еще несколько модулей от которых он зависит. В частности мы видим информационный вывод от модулей **amdgpio** и **gpiobus** сообщающих нам о количестве обнаруженных GPIO портов и линий на каждом из них. Также мы видим вывод от нашего модуля где он сообщает об успешном подключении к GPIO шине и получении доступа к GPIO пину 0 (сообщение "**gpiopwrb: attached to gpio pin 0** ").

После того, как наш модуль **gpiopwrb** загрузился и подключился к требуемым ресурсам, он запускает поток исполнения из которого на консоль постоянно выводится строка интересующая нас о состоянии линии GPIO и значении флаговой переменной **changed**. Сделав несколько кратковременных нажатий на кнопки «Power» мы будем наблюдать как изменяется значение этой переменной:

```
gpiopwrb: poll... gpio pin 0 state = 1, changed = no
gpiopwrb: poll... gpio pin 0 state = 1, changed = no
gpiopwrb: poll... gpio pin 0 state = 0, changed = YES
gpiopwrb: GPIO pin 0 changed, emmiting ACPI event \_SB.GPIO._EVT
gpiopwrb: poll... gpio pin 0 state = 1, changed = no
gpiopwrb: poll... gpio pin 0 state = 1, changed = no
gpiopwrb: poll... gpio pin 0 state = 1, changed = YES
gpiopwrb: GPIO pin 0 changed, emmiting ACPI event \_SB.GPIO._EVT
gpiopwrb: poll... gpio pin 0 state = 1, changed = no
gpiopwrb: poll... gpio pin 0 state = 1, changed = no
```

Вместе с изменение флаговой переменной (**changed = YES**) модуль информирует нас о вызове ACPI метода сообщением "**emmiting ACPI event _SB.GPIO._EVT** ". Похоже, что модуль работает как задумывалось.

Теперь проверим как отрабатывает выгрузка модуля. Для этого введем команду **kldunload**. Вводить её придется вслепую, так как вывод отладочной информации от нашего модуля заполняет консоль и скрывает промпт командной строки. Имя выгружаемого модуля можно указывать без суффикса **.ko**:

```
kldunload gpiopwrb
gpiopwrb: poll... gpio pin 0 state = 1, changed = no
gpiopwrb: poll... gpio pin 0 state = 1, changed = no

gpiopwrb: waiting thread termination...
gpiopwrb: kthread 0xfffff800093c3780 exited
gpiopwrb: detached from gpio pin 0
gpiopwrb0: detached
gpiobus0: <unknown device> at pin
gpioc0: detached
gpiobus0: detached
gpio0: detached
```

Наш модуль освободил занимаемые им ресурсы, успешно выгрузился системой и выгрузил за собой все затянутые ранее зависимости. Проверим какие модули остались загруженными в системе:

```
root@:~ # kldstat
Id Refs Address                Size Name
1    1 0xffffffff80200000  2512a30 kernel
```

Команда **kldstat** показывает нам только один модуль **kernel** — само ядро операционной системы. Больше ничего нет, и это замечательно.

Всё выглядит так, будь-то бы наш драйвер действительно работает корректно. Если бы что-то пошло не так, то мы, во-первых, увидели бы множественные сообщения об ошибках, и, во-вторых, система скорее всего ушла бы в panic! Так, как ничего подобного не наблюдается, то попробует разрешить системе обработать события от кнопки «Power» - вернем прежнее значение переменной ядра **hw.acpi.power_button_state** = **S5**:

```
root@:~ # sysctl hw.acpi.power_button_state=S5

hw.acpi.power_button_state: NONE -> S5
```

Напомню, что согласно спецификации ACPI переход системы в состояние **S5** это полный «Shut Down» с отключением питания. Загружаем модуль еще раз, нажимаем кнопку «Power» и смотрим что происходит:

```
root@:~ # kldload gpiopwrb.ko
...
gpiopwrb: attached to gpio pin 0
...
gpiopwrb: poll... gpio pin 0 state = 1, changed = no
gpiopwrb: poll... gpio pin 0 state = 1, changed = no
gpiopwrb: poll... gpio pin 0 state = 1, changed = no
gpiopwrb: poll... gpio pin 0 state = 0, changed = YES
gpiopwrb: GPIO pin 0 changed, emmiting ACPI event \_SB.GPIO._EVT

Waiting (max 60 seconds) for system process `vnlru' to stop... done
Waiting (max 60 seconds) for system process `syncer' to stop...
Syncing disks, vnodes remaining... 0
```

Как видно из вывода, операционная система завершила свою работу, синхронизировала дисковые кэши и обесточила машину. Красота! Вот теперь можно констатировать, что искомый результат достигнут. Всё, что нам еще потребуется сделать, перед тем как идти в магазин за бутылкой шампанского, — это пересобрать модуль без отладочного вывода, чтобы он вел себя тихо и спокойно, как все остальные драйверы. После чего, перенести его на целевую машину и добавить в конфиг системного загрузчика командой:

```
root@:~ # sysrc -f /boot/loader.conf gpiopwrb_load="YES"
gpiopwrb_load: NO -> YES
```

Но на этом разработка модуля не заканчивается! Согласно устоявшейся традиции, в ОС FreeBSD каждая утилита, каждая библиотека и каждый модуль должны быть снабжены страницей документации доступной через утилиту **man**. Созданием своей маленькой man-страницы мы займемся в следующей главе.

4.7. Подготовка man-странички руководства для gpiopwrb

Для описание страниц системного руководства во многих UNIX системах традиционно используется формат разметки текста на основе языка **roff**, к появлению которого приложил руку лично Dennis M. Richie — один из создателей ОС UNIX и языка программирования Си. Язык **roff**, в свою очередь, основывается на системе верстки документов под названием **RUNOFF** придуманной еще до появления ОС UNIX. Ниже небольшая историческая справка взятая со странички руководства **man roff**:

> *HISTORY*
>
> *The RUNOFF typesetting system, whose input forms the basis for roff, was written in MAD and FAP for the CTSS operating system by Jerome E. Saltzer in 1964. Doug McIlroy rewrote it in BCPL in 1969, renaming it roff. Dennis M. Ritchie rewrote McIlroy's roff in PDP-11 assembly for Version 1 AT&T UNIX, Joseph F. Ossanna improved roff and renamed it nroff for Version 2 AT&T UNIX, then ported nroff to C as troff, which Brian W. Kernighan released with Version 7 AT&T UNIX. In 1989, James Clark re-implemented troff in C++, naming it groff.*

Вы только вникнитесь в цепочку преобразований, в следствии которой **RUNOFF** превратился в **groff**: RUNOFF → roff на BCPL → roff на ASM для PDP-11 → nroff → troff на Си → groff на C++. Чувствуете запах фанталина ? А что делать, традиции!

Но и это еще не всё. Язык **roff** был расширен набором макрокоманд специфичных для написания стилизованных системных руководства (общий стиль и семантика). Эта расширенная версия получила название «MDOC Markup Language» или просто **mdoc**. Первоначально для преобразования страниц **mdoc** в визуальный формат выводимый на печать (или на экран терминала) использовалась утилита **groff** (GNU roff), но позже в BSD системах её заменили на утилиту **mandoc**. Далее мы будем пользоваться именно этой утилитой для просмотра подготовленной нами страницы руководства.

Язык разметки **mdoc**, в отличии, скажем, от широко используемого языка **Markdown**, является семантическим. Это означает, что в **mdoc** все выделения (подчеркивания, подсветка текста и табуляции) появляются в зависимости от смысла и места расположения текста на

странице, а не от желания автора текста «подсветить» какой-то фрагмент. Возможность произвольной «раскраски» текста в **mdoc** сохраняется, но не приветствуется. Более того, на автора накладываются определенные ограничения по стилю изложения и требования к содержанию руководства.

Отличное руководство по написанию руководств (извините за тавтологию) находится в Главе 11 проекта «*FreeBSD Documentation Project Primer for New Contributors*» в узких кругах известного также как «FDP». Данный сборник сочинений представляет собой набор статей различных авторов собранных с общей целью — помочь новым разработчикам и контрибьюторам ОС FreeBSD сориентироваться в море негласных стандартов и традиций. Далее я постараюсь очень кратно изложить требования и формат файла **mdoc**. Разумеется, исчерпывающее справочное руководство по языку разметки можно всегда получиться по команде **man mdoc**.

4.7.1. Подсистема встроенной документации mdoc

В операционной системе FreeBSD, только в её базовой части, насчитывается более **11500** страниц руководства, а каждая установленная из пакетов или портов утилита добавляет свой набор в общую копилку. Эти страницы текста на языке **mdoc** сжимаются архиватором **gzip** и складываются в каталог **/usr/share/man/** для базовых системных утилит, и в **/usr/local/share/man/** для установленных из вне (из портов, пакетов или вручную). Внутри каждого располагаются подкаталоги с пронумерованными именами **man1/ - man9/**. Каждый из этих подкаталогов содержит страницы относящиеся к определенному тематическому разделу (Manual Section), в ОС FreeBSD таких разделов ровно девять:

```
The sections of the manual are:
      1.   FreeBSD General Commands Manual
      2.   FreeBSD System Calls Manual
      3.   FreeBSD Library Functions Manual
      4.   FreeBSD Kernel Interfaces Manual
      5.   FreeBSD File Formats Manual
      6.   FreeBSD Games Manual
      7.   FreeBSD Miscellaneous Information Manual
      8.   FreeBSD System Manager's Manual
      9.   FreeBSD Kernel Developer's Manual
```

Каждый файл внутри подкаталога **manX/** имеет имя в конце которого через символ «точка» добавляется суффикс с номером раздела, а также суффикс **.gz** если страница сжата архиватором. Первое требование — имя файла man-стриницы должно совпадать с названием утилиты, библиотеки, функции или имени драйвера описание которого она содержит, так как по этому имени утилита **man** находит соответствующую страницу.
Для описания драйверов и прочих модулей ядра в ОС FreeBSD отводится раздел с номером 4 — «FreeBSD Kernel Interface», поэтому файл с руководством для нашего драйвера будет иметь название **gpiopwrb.4**, а его сжатый вариант для размещения в системе — **gpiopwrb.4.gz**. После установки в систему нашей man-странички её можно будет вызвать командой **man 4 gpiopwrb** (или сокращенно: **man gpiopwrb**).

Внутри man-страницы находиться текст на упрощенном Американском Английском языке (например, слова типа «colour» и «programme» должны записываться как «color» и «program»), преимущественно с короткими предложениями без трехэтажных деепричастных оборотов, в обезличенной неопределенной форме. По содержанию текст должен быть лаконичным и сохранять баланс между ясностью изложения, краткостью и полнотой. Такие требования предъявляет к документации проект FDP.

Текст обильно разбавляется макрокомандами языка **mdoc**. Макрокоманда обычно начинается с символа «точка» в самом начале строки, за которым следую два или три

символа названия макрокоманды, далее, через пробел, следую аргументы этой макрокоманды. Ниже приведен список некоторых часто используемых макрокоманды:

- **.\" some comment** — до конца строки располагается текст комментария **some comment text**, этот текст не участвует в выводе на печать.
- **.Dd Month Day, Year** — задает дату создания документа, где **Month** — название месяца на Английском, **Day** и **Year** — соответственно числа. Вместо даты можно указать макропеременную **$Mdocdate$**, вместо которой будет подставлена дата последнего изменения этого файла.
- **.Dt Command X** — задает название команды **Command** (утилиты, функции, модуля, и т. д.), описание которой содержится в данном документе, а так же номер раздела **X**, к которому она относится.
- **.Nt Command** — первый вызов данной макрокоманды запоминает название команды содержащейся в строке **Command** (должно совпадать с тем, что задано в **.Dt**), последующие вызовы без параметра приводят к подстановке имени команды в текст.
- **.Os** — подставляет версию ОС для которой был написан документ.
- **.Sh Section** — открывает поименованный раздел **Section** внутри документа. Каждая man-страница содержит несколько разделов, часть из них являются обязательными. Об этом чуть далее.
- **.Pp** — начало параграфа (добавляет строку отступа).
- **.Pa pathname** — путь к файлу в файловой системе (отображается подчеркнутым).
- **.Ev ENV** — отображает текст **ENV** как переменную окружения.
- **.Bd params** и .**Ed** — обрамляют блок текста, параметры которого задаются **params**.
- **.Bl params** и **.El** — обрамляют списки, параметр списка задается в **params**.
- **.Op Fl x Ar file** — добавляет элемент перечня флагов командной строки, где **x** — имя флага, **file** — имя файла или путь. Возможны вариации типа **.Op Fl x** — только флаг, или **.Op Ar file** — только файл, или даже **.Op Ar** — напечатает вместо имени файла три точки «...».
- **.In file** — добавляет элемент перечня include-файл **file**.
- **.Fn func "arg1" "arg2"** — добавляет элемент перечня имя функции **func** с входными параметрами **arg1** и **arg2**.
- **.Xr name X** — ссылка на другую страницу руководства с именем **name** в разделе **X**.

Приведенного выше списка макрокоманд вполне достаточно для создания хороших man-страниц.

4.7.2. Структура документа mdoc

Теперь рассмотрим собственно структуру man-страницы. Каждая страница руководства состоит из «пролога» (типового заголовка) и нескольких обязательных, а иногда и ряда дополнительных, секций.

Пролог обязан содержать небольшой комментарий с описанием лицензии, за которым идет следующий набор макрокоманд: **.Dd**, **.Dt**, и **.Os**. Для нашего драйвера выглядит следующим образом:

Листинг 17а. Пролог (заголовок) для man-стриницы драйвера gpiopwrb.

```
.\"
.\" SPDX-License-Identifier: BSD-2-Clause
.\"
.\" Copyright (c) 2026, Ruslan Zalata <rz@fabmicro.ru>
.\" All rights reserved.
```

```
.Dd $Mdocdate$
.Dt GPIOPWRB 4
.Os
```

Сразу за прологом следую обязательные секции в следующем порядке:

- **.Sh NAME** — определяет имя команды, утилиты, библиотеки, функции или модуля. Здесь же содержится однострочное описание предмета.
- **.Sh SYNOPSIS** — описывает способ сборки, установки, синтаксис командной строки или синтаксис вызова функции. Кратко о том, как пользоваться предметом.
- **.Sh DESCRIPTION** — детальное описание предметной области, с расшифровкой переметров командной строки или параметров вызова функции.

Дополнительные секции:

- **.Sh NOTE** — содержит примечания или некоторые особенности на которые стоит обратить внимание пользователю.
- **.Sh SEE ALSO** — кросс-ссылки на другие страницы руководства.
- **.Sh OBSOLETE** — указывает, когда и при каких условиях данный предмет (утилита, функция или модуль) потеряют смысл.
- **.Sh HISTORY** — история создания и появления описанного предмета, а также история изменения данного руководства.

Завершающая обязательная секция:

- **.Sh AUTHORS** — информация об авторах (отдельно для предмета и руководства).

Ниже в Листинге 17б я приведу полный текст руководства для нашего драйвера **gpiopwrb** так, как вижу его я. При желании читатель может прислать мне дополнения и корректировки в виде «Pull Request» на сайте Github.

Листинг 17б. Основной текст man-стриницы драйвера gpiopwrb.

```
.Sh NAME
.Nm gpiopwrb
.Nd GPIO-based Power/Sleep Button driver that polls GPIO pin status and
raises ACPI event.
.Sh SYNOPSIS
To compile and install this driver, use
.Bd -literal -offset indent
make
make install
.Ed
.Pp
as superuser. Then, add this module to
.Ev kld_list
variable in your
.Pa /etc/rc.conf
.Sh DESCRIPTION
.Pp
On many PCs with Hardware-Reduced ACPI equipped with AMD Ryzen CPUs, which
are mostly modern laptops, Power Button is wired to AMD GPIO controller.
FreeBSD has support for AMD GPIO since 2018 implemented by
.Pa amdgpio
module.  Unfortunately, it does not support interrupts on amd64 platform,
it's simply not implemented because of lack of INTRNG support in upstream
.Pa gpiobus
module. Hence, pressing Power Button will never trigger poweroff or sleep
event.
.Pp
This
.Nm
module provides a quick and dirty hack to allow Power Button to work on
such PCs by periodically polling state of given GPIO pin and calling
```

```
associated ACPI method to raise event.
.Pp
The default behaviour of this module is to poll state of GPIO pin 0 and
to call
.Pa \\\_SB.GPIO._EVT
ACPI method. The polling interval is 250 milliseconds.

On some machines GPIO pin number and/or ACPI method name implemented in
AML firmware can be different. In this case it's possible to configure
.Nm
module to use these hint values:
.Bl -tag -width ".Va hint.gpiopwrb.0.pin_num"
.It Va hint.gpiopwrb.0.pin_num
GPIO pin number (decimal or 0x hex). Default: 0.
.It Va hint.gpiopwrb.0.acpi_path
ACPI pathname to call on pin state change. Default: \\\_SB.GPIO._EVT
.El
.Pp
Add them to your
.Pa /boot/loader.conf
like:
.Pp
.Bd -literal
hint.gpiopwrb.0.pin_num=0 # <-- set GPIO pin number for Power Button
hint.gpiopwrb.0.acpi_path="\\\\_SB.GPIO._EVT" # <-- call this  ACPI event
.Ed
.Pp
or use
.Xr kenv 1
utility to set hints temporarily before loading
.Nm
module manually with
.Xr kldload 8
like in the following:
.Bd -literal
# kenv hint.gpiopwrb.0.pin_num=0
# kenv hint.gpiopwrb.0.acpi_path='\\\_SB.GPIO._EVT'
.Ed
.Pp
Use single backslash and single quote character while in shell.
.Pp
To learn exact GPIO pin number one can use
.Xr gpioctl 8
utility to get pin list and their current states.

To learn ACPI pathname one has to decompile and analyze AML firmware.
There are
.Xr acpidump 8
utility to dump DSDT (AML code) and
.Xr iasl 8
compiler/decompiler to convert AML byte-code to C-style source code
provided by FreeBSD.
.Sh NOTE
It is possible to add
.Nm
module to loader config for early load. To do
this just add the following line into
.Pa /boot/loader.conf :
.Pp
.Ev gpiopwrb_load="YES"
.Pp
.Sh SEE ALSO
.Xr gpio 4 ,
.Xr gpioctl 8 ,
.Xr acpidump 8 ,
.Xr iasl 8
.Sh OBSOLETE
This
.Nm
module becomes obsolete soon as interrupts are implemented in upstream
modules
.Pa gpiobus
and
.Pa amdgpio .
.Sh AUTHORS
This manual page was written by
.An Ruslan Zalata.
```

Для того, чтобы отобразить в печатном виде созданный нами документ формата **mdoc**, можно воспользоваться утилитой **mandoc**, передав в качестве параметра имя исходного файла документа. Вот как этот документ выглядит в моей системе:

Листинг 17в. Отображение man-стриницы для драйвера gpiopwrb на экране.

```
rz@butterfly:~/gpiopwrb % mandoc gpiopwrb.4
GPIOPWRB(4)            FreeBSD Kernel Interfaces Manual            GPIOPWRB(4)

NAME
     gpiopwrb – GPIO-based Power/Sleep Button driver that polls GPIO pin
     status and raises ACPI event.

SYNOPSIS
     To compile and install this driver, use

           make
           make install

     as superuser. Then, add this module to kld_list variable in your
     /etc/rc.conf

DESCRIPTION
     On many PCs with Hardware-Reduced ACPI equipped with AMD Ryzen CPUs,
     which are mostly modern laptops, Power Button is wired to AMD GPIO
     controller.  FreeBSD has support for AMD GPIO since 2018 implemented by
     amdgpio module.  Unfortunately, it does not support interrupts on amd64
     platform, it's simply not implemented because of lack of INTRNG support
     in upstream gpiobus module. Hence, pressing Power Button will never
     trigger poweroff or sleep event.

...
     On some machines GPIO pin number and/or ACPI method name implemented in
     AML firmware can be different. In this case it's possible to configure
     gpiopwrb module to use these hint values:

     hint.gpiopwrb.0.pin_num      GPIO pin number (decimal or 0x hex).
                                  Default: 0.

     hint.gpiopwrb.0.acpi_path    ACPI pathname to call on pin state change.
                                  Default: \_SB.GPIO._EVT

     Add them to your /boot/loader.conf like:

     hint.gpiopwrb.0.pin_num=0 # <-- set GPIO pin number for Power Button
     hint.gpiopwrb.0.acpi_path="\\_SB.GPIO._EVT" # <-- call this  ACPI event

     or use kenv(1) utility to set hints temporarily before loading gpiopwrb
     module manually with kldload(8) like in the following:

...

NOTE
     It is possible to add gpiopwrb module to loader config for early load. To
     do this just add the following line into /boot/loader.conf:

     gpiopwrb_load="YES"

SEE ALSO
     gpio(4), gpioctl(8), acpidump(8), iasl(8)

OBSOLETE
     The gpiopwrb module becomes obsolete soon as interrupts are implemented
     in upstream modules gpiobus and amdgpio.

AUTHORS
     This manual page was written by Ruslan Zalata.

FreeBSD 15.1-RELEASE       October 15, 2026                       GPIOPWRB(4)
```

4.7.3. Добавление документации в проект

Как отмечалось выше, исходный текст документа необходимо сжать архиватором **gzip** и подготовить для установки в соответствующие подкаталоги системы. Для автоматизации этого процесса в составе ОС FreeBSD имеется шаблонный **.mk**-файл с именем **bsd.man.mk**, в нем содержится весь код по сборке man-странички. Проблема с этим **bsd.man.mk** состоит в том, что его подключение на прямую в **Makefile** проекта не предусматривается. Это скрипт вызывается автоматически из другого шаблонного скрипта, **bsd.prog.mk**, предназначенного для сборки системных утилит. Однако, мы собираем не утилиту, а модуль ядра, с помощью скрипта **bsd.kmod.mk**, а он, как выяснилось, про man-страницы ничего не знает и не содержит внутреннего вызова для **bsd.man.mk**.

Один из способов решения данной проблемы состоит в том, чтобы разбить проект на две части (на два подкаталога). В одном подкаталоге, назовем его **module/**, будет располагаться файл(ы) с исходным текстом нашего драйвера и **Makefile** для него. В другом подкаталоге, **man/**, разместим файл(ы) **mdoc** формата и другой **Makefile** для сборки man-страницы. Для того, чтобы утилита **make** проходила по вложенным подкаталогам и рекурсивно вызывала саму себя для сборки отдельных частей проекта, укажем ей в **Makefile** верхнего уровня переменную **SUBDIR** с перечнем подкаталогов. Тогда наш проект примет следующую структуру:

Листинг 18а. Структура подкаталогов проекта драйвера gpiopwrb с man-стриницей.

```
rz@butterfly:~/gpiopwrb % tree
.
├── Makefile
├── README.md
├── man
│   ├── Makefile
│   └── gpiopwrb.4
└── module
    ├── Makefile
    └── gpiopwrb.c
```

У нас получается целых три сборочных **Makefile**-а — один верхнего уровня и два в подкаталогах. Вот как выглядит содержимое всех трех файлов:

Листинг 18б. Три сборочных скрипта (Makefile) для проекта gpiopwrb.

```
rz@butterfly:~/gpiopwrb % cat Makefile
# Build script for GPIO Power Button driver module and man page

SUBDIR= module man

.include <bsd.subdir.mk>


rz@butterfly:~/gpiopwrb % cat module/Makefile
# Build script for GPIO Power Button driver module

KMOD=   gpiopwrb
SRCS=   gpiopwrb.c device_if.h bus_if.h gpio_if.h acpi_if.h opt_acpi.h opt_platform.h

.if defined(DEBUG)
CFLAGS+=       -DGPIOPWRB_DEBUG=1
.endif

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

```
rz@butterfly:~/gpiopwrb % cat man/Makefile
# Build script for GPIO Power Button diver manual page

MAN=    gpiopwrb.4

.include <bsd.prog.mk>
```

Из приведенного листинга видно, что для сборки модуля, как и прежде, используется шаблон **bsd.kmod.mk**, а для сборки man-странички используется **bsd.prog.mk.** Всё это вместе соединяет шаблонный скрипт **bsd.subdir.mk**. Ответа на вопрос почему разработчики системы не добавили в шаблон **bsd.kmod.mk** поддержку сборки man-страниц я не нашел, но подозреваю, что это связано со сборкой ядра ОС FreeBSD, где man-стрницы и код модулей собираются раздельно и разнесены в разные подкаталоги.

4.8. Установка драйвера в систему

Теперь когда у нас готов весь проект, код драйвера отлажен и man-стриничка подготовлена, драйвер можно установить в систему. Для этого сначала воспользуемся командой **make install**, чтобы поместить файл с модулем в соответствующий системный каталог, после чего добавим имя модуля в конфигурационную **kld_list** в файле **/etc/rc.conf** с помощью команды **sysrc**:

```
rz@butterfly:~/gpiopwrb % sudo make install
===> module (install)
install -T release -o root -g wheel -m 444   gpiopwrb.ko /boot/modules/
kldxref /boot/modules
===> man (install)
install  -o root -g wheel -m 444 gpiopwrb.4.gz  /usr/share/man/man4/

rz@butterfly:~/gpiopwrb % sudo sysrc kld_list+=gpiopwrb
kld_list: ... -> ... gpiopwrb

rz@butterfly:~/gpiopwrb % sudo reboot
```

Если после перезагрузки ОС возникли какие-то проблемы, то всегда можно прибегнуть к «однопользовательскому» режиму воспользовавшись инструкцией из главы «*2.5. Если что-то пошло не так*» и исправить ситуацию.

На этом считаю разработку драйвера для кнопки «Power» для ОС FreeBSD полностью завешенной. Репозиторий с приведенным в данной статье кодом можно полистать на сайте Github по ссылке: https://github.com/pointcheck/gpiopwrb.git

5. Заключение

В процессе исследования проблемы мне довелось пообщаться с разработчиками ОС FreeBSD в списках рассылки, сообщить о баге в багтрекере и даже самому предложить решение проблемы отсутствующей поддержки прерываний в подсистеме GPIO. О последнем расскажу немного подробней.

Разработка ядра и всех системных утилит ОС FreeBSD ведется в едином Git репозитории обновления в который добавляются коммиттерами (разработчиками имеющими соответствующие права доступа) в виде патчей — небольших изменений кода. Патчи могут предлагать не только коммиттеры, но и обычные «смертные» программисты со всей планеты, таких людей принято называть «контрибьютерами» (человек который жертвует свой код в общую копилку). Теоретически, тут нет никаких ограничений кроме профессиональных навыков, любой может стать контрибьютером кода в репозиторий FreeBSD. Но на практике это весьма не просто, так как к предлагаемому коду всегда должны прилагаться соответствующие профессиональные навыки. Каждый предложенный патч проходит анализ (проверку) от нескольких других разработчиков, этот процесс принято называть «проверка кода» или «ревью» (review). В процесс этой проверки каждая строка предложенного патча будет разобрана «по косточкам» и её автору будут заданы технические вопросы, на которые он должен ответить путем вноса соответствующих изменений в свой код (или аргументированно объяснить почему автор считает предложенное им решение верным). Для автоматизации процесса выполнения проверки предложенных изменений, проектом FreeBSD с 2014-го года используется специальный on-line сервис известный как **Phabricator**, он доступен всем по адресу https://reviews.freebsd.org/. С помощью данного сайта любой программист, даже без регистрации, может наблюдать за тем какие патчи в данный момент проходят проверку, как они изменяются в процессе проверки и какой конкретно код будет принят (или уже был когда-то принят) в основной репозиторий. Каждый патч вынесенный на обсуждения (загруженный на сайт Phabricator) принято называть «diff» (от англ. «Differential»), они имеют сплошную нумерации вида **Dxxxxxx**, где xxxxxx — серийный номер. Очень часто разработчики и пользователи ОС FreeBSD могут ссылаться на «diff-ы» в других местах — на форумах, в списках рассылки или IRC каналах.

Для меня Phabricator оказался очень полезным ресурсом, так как в нём собраны не только предложенные патчи (принятые или отвергнутые), но и вся дискуссия организовавшаяся в процессе прохождения ревью для каждого патча. В процессе поиска решения для проблемы отсутствия поддержки фреймворка INTRNG на x86 архитектуре, я случайно наткнулся на D26407 — патч предложенный еще в 2020 году разработчиком по имени Andriy Gapon (**avg**). С помощью этого патча предлагается добавить в подсистему GPIO (в код драйвера **gpiobus**) свой «виртуальный» PIC контроллер, который по замыслу должен обрабатывать аппаратные прерывания от контроллера GPIO и транслировать их своим потребителям — другим драйверам. Процесс проверки данного патча зашел в тупик, так некоторым разработчиками участвовавшим в этом ревью (если я правильно понял дискуссию) не понравился такой «костыль» в ядре ОС. Тем не менее, мне показалось это решение весьма интересным. К тому же, к дискуссии подключился еще один разработчик по имени ShengYi Hung (**aokblast**) который работал на проблемой низкой скорости поступления событий от тач-падов на некоторых ноутбуках, где выход линии прерывания тач-пада заведен на вход GPIO. То есть для решения его проблемы ему тоже требовалась поддержка прерываний в подсистеме GPIO на x86 чтобы избавиться от режима poll-а в драйвере.

Я выкачал этот патч и попытался применить его к 16.0-CURRENT ветке ядра ОС FreeBSD. Разумеется, из-за огромного объема изменений которые претерпел код ядра с 2020 года этот патчи применяться отказался. Но я был настойчив и, за пару ночных спиритических

сеансов, мне удалось адаптировать его к самой последней версии кода ядра. Патч заработал. Правда мне пришлось исправить один нюанс связанный с особенностью маскирования прерываний при выгрузке драйвера, но всё работало так как описывал его **avg**. На таком пропатченном ядре у меня загрузился драйвер **gpioaei**, он смог подцепить свой обработчик событий (вызов управляющего ACPI метода _EVT) к GPIO контроллеру и кнопка «Power» заработала, как того и следовало ожидать! Иными словами, если бы D26407 тогда прошел ревью и был бы принят, я бы сейчас не тратил свой время на решение это глупой проблемы, а этой статьи и вовсе не случилось бы! :-)

Я загрузил обновленный вариант патча на Phabricator под номером D57968 и попытался обозначить проблему в списке рассылки freebsd-current@, вкратце обрисовав ход моих исследований. Сейчас этот патч находится на ревью уже более трех месяцев (да этот процесс никогда не был быстрым), но будем надеяться что его примут. Либо кто-то другой найдет в себе силы реализовать полную поддержку фреймворка INTRNG для платформа x86, и тогда проблема прерываний от GPIO в ОС FreeBSD будет окончательно решена.

Список используемых источников

1. Advanced Configuration and Power Interface (ACPI). Wikipedia [Электронный ресурс] // URL: https://en.wikipedia.org/wiki/ACPI (дата обращения: 02.05.2026).
2. Garrett M. Why ACPI? [Электронный ресурс] // dreamwidth.org. – 2026. – URL: https://mjg59.dreamwidth.org/68350.html (дата обращения: 02.06.2026).
3. Advanced Configuration and Power Interface Specification Version 6.6 [Электронный ресурс] // UEFI Forum. – 2025. – URL: https://uefi.org/sites/default/files/resources/ACPI_Spec_6.6.pdf (дата обращения: 03.07.2026).
4. Using and Debugging FreeBSD ACPI. FreeBSD Handbook [Электронный ресурс] // docs-archive.freebsd.org. – 2011. – URL: https://docs-archive.freebsd.org/doc/7.4-RELEASE/usr/share/doc/handbook/acpi-debug.html (дата обращения: 02.06.2026).
5. ACPI _OSI and _REV methods [Электронный ресурс] // kernel.org. – . – URL: https://www.kernel.org/doc/html//v5.4/firmware-guide/acpi/osi.html (дата обращения: 18.05.2026).
6. Brückner P. Power Button Resurrection on the Surface Go 2 [Электронный ресурс] // sylence.cc. – 2022. – URL: https://sylence.cc/posts/power-button-surface-go-2/ (дата обращения: 06.05.2026).
7. Linimon M. Complicated Interrupts [Электронный ресурс] // wiki.freebsd.org. – 2018. – URL: https://wiki.freebsd.org/Complicated_Interrupts (дата обращения: 07.06.2026).
8. Brandi E. FreeBSD Device Drivers From First Steps to Kernel Mastery [Электронный ресурс] // GitHub. – 2026. – URL: https://github.com/ebrandi/FDD-book (дата обращения: 08.07.2026).

9. FreeBSD Documentation Project Primer for New Contributors (FDP) [Электронный ресурс] // docs.freebsd.org. – 2026. – URL: https://docs.freebsd.org/en/books/fdp-primer/ (дата обращения: 09.08.2026).
10. The FreeBSD Forum [Электронный ресурс] // forums.freebsd.org. – 2026. – URL: https://forums.freebsd.org/ (дата обращения: 11.09.2026).
11. FreeBSD Phabricator [Электронный ресурс] // reviews.freebsd.org. – 2026. – URL: https://reviews.freebsd.org/ (дата обращения: 12.05.2026).
12. FreeBSD Bugzilla [Электронный ресурс] // bugs.freebsd.org. – 2026. – URL: https://bugs.freebsd.org/ (дата обращения: 12.05.2026).
13. gpiopwrb [Электронный ресурс] // GitHub. – 2026. – URL: https://github.com/pointcheck/gpiopwrb.git (дата обращения: 22.10.2026).