Разработка цифровой аппаратуры нетрадиционным методом: Контроллер USB 1.0 на SpinalHDL

Автор: Залата Руслан Николаевич

Продолжая развивать свою синтезируемую систему-на-кристалле для ПЛИС, о которой я уже написал несколько статей, столкнулся с необходимостью подключать устройства ввода типа клавиатура, манипулятор мышь или джойстик. Если обратиться к тому, чем занимаются ретро-фаны, то проблем особых нет — старый добрый интерфейс PS/2 очень прост в реализации, он позволяет легко взаимодействовать с клавиатурой и мышью с минимальными ресурсами. Фактически PS/2 это последовательный синхронный порт работающий на низких скоростях, реализовать его можно программно. С ретро-джойстиками тоже проблем нет - положение джойстика это всего лишь замыкание контактов, что легко обрабатывается программно. Проблема в том, что всё это «ретро» постепенно уходит из нашей жизни, клавиатуры и мыши с интерфейсом PS/2 всё еще можно приобрести на маркетплейсах, но всё же редкость. И от джойстика хочется чего-то большего чем просто замыкания пяти контактов, а именно — градации положения стика. Такая фича доступна либо на очень старых аналоговых джойстиках, либо на современных геймпадах с USB интерфейсом. В конце концов я разрабатываю хоть и минималистичную, но современную систему с современной архитектурой (RISC-V) предназначенную для современного промышленного применения, а не для ретро-гейминга. ;-) В общем, встал вопрос как подключать простые HID устройства ввода через USB к своей синтезируемой ЭВМ.

Интерфейс шины USB настолько широко вошел в обиход, что мы даже не задумываемся, что там внутри: сколько сигнальных проводов в USB кабеле, как они подключены, как передаются по ним данные, на каких скоростях и какие могут быть ограничения. Всё что мы знаем это то, что USB бывает разных версий: 2.0 — медленный и 3.0 — очень быстрый; и что USB устройства бывают с разными видами разъемов: **USB type A** и, с недавних пор, **USB type C**. Для большинства пользователей и программистов USB это такая штука, которую «вставил и работает». А если нет, то нужно вынуть, перевернуть устройство два раза вокруг его оси и вставить в компьютер еще раз. Если и так не заработало, то искушенный пользователь возможно вспомнит команду ***lsusb*** чтобы выяснить какие сейчас устройства присутствуют в системе или даже заглянет в ***dmesg*** чтобы выяснить наличие ошибок при детектировании устройства. Но что означают эти сообщения? Еще меньшее число пользователей понимает результат вывода команды ***lsusb -v***. Немногим лучше обстоят дела с пониманием USB у разработчиков электроники. Обычно на их уровне USB это четыре провода: **GND**, **VBUS**, **D+** и **D-**, при этом каждый электронщик знает что D+ и D- это дифференциальная пара которую требуется трассировать на печатной плате соответствующим образом. Но так ли это на самом деле?

Раз уж возникла необходимость, то надо погружаться в тему если не по уши, то хотя бы по пояс и выяснить, а насколько сложно реализовать свой собственный минималистичный USB контроллер. Ведь задача то очень простая — считать пару байт с USB клавиатуры, и, как мне казалось, осилить её можно за пару-тройку ночных сейшнов. Но я и не подозревал, что выльется это почти в полугодичный ~~ежедневный~~ еженощный зависон в обнимку с осциллографом и анализатором сигналов.

Рис. 1. Кабель стандарта USB 1.0: разъем Тип А (слева) и USB mini (справа).

СОДЕРЖАНИЕ

1. Что такое USB ?

Universal Serial Bus (далее USB) — это объемный набор стандартов описывающих цифровую шину с последовательным способом организации передачи данных, который был введен в середине 1990-х в качестве замены другой шины с последовательной передачей данных — Universal Asyncronous Receiver-Transmitter (более известной как UART, или RS-232, или «COM порт»). Как и UART, шина USB является последовательной и асинхронной шиной связывающей два устройства, то есть данные по ней передаются бит за битом от одного устройства «источника» к другому «приемнику», а начало передачи данных может происходить в любой момент времени по мере их готовности на передающей стороне. Однако, в отличии от UART, шина USB является однонаправленной, то есть передача данных в один и тот же момент времени возможна только в одном направлении от «источника» к «приемнику», а для того, чтобы данные можно было передать в обратном направлении, устройствам необходимо сначала договориться о смене ролей. Такой режим работы принято называть «полу-дуплекс», который можно противопоставить «полному дуплексу» в UART где в один и тот же момент времени данные могут передаваться в оба направления независимо. Другим отличительным свойством шины USB является то, что в связке двух USB устройств одно всегда является ведущим («host»), а другое ведомым («device»). Еще одним существенным отличием USB от UART является способ кодирования данных (представления «нулей» и «единиц» электрическими сигналами). Как и у UART в USB имеется два физических проводника для передачи данных, однако способ их использования отличается радикально. В UART эти проводники принято называть TXD и RXD. Первый используется для передачи данных от устройства, второй для приема данных на устройство. «Нули» и «единицы» в UART представляются различными уровнями напряжений на этих проводниках, например, **+3,3 В** может кодировать «единицу», а **0 В** — соответственно «ноль». Существуют и другие варианты кодирования данных в UART, например стандарт на физический интерфейс EIA RS-232 описывает уровни **+3..15 В** для логического «нуля» и **-3..-12 В** для логической «единицы». В USB для передачи данных используется одновременно два проводника которые принято называть **D+** и **D-** (иногда DP и DM), каждый может принимать два уровня напряжения +3,6 В и 0 В, что позволяет представить четыре различных состояния шины. Кодирование «нуля» или «единицы» производится не уровнями напряжений, а последовательностью смены «полярностей» между D+ и D-. Такой способ кодирования принято называть Non-Return-to-Zero (NRZ). Их существует несколько вариантов, тот что используется в USB называется NRZi от «NRZ IBM» или «NRZ-inverted» (оба варианта верны, кому как больше нравится). NRZi кодирование, во-первых, более устойчиво к помехам на высоких скоростях, а во-вторых, позволяет приемной стороне постоянно пересинхронизироваться, то есть не сбиваться при приеме данных. Я сейчас говорю про USB 1.x и USB 2.0. В USB 3.x всё гораздо сложнее и сигнальных линий там больше (три пары), но об этом подробнее чуть ниже.

Так зачем и кому могла понадобиться такая странная однонаправленная шина при том, что UART к середине 90-х имел почти повсеместное проникновение и любое цифровое оборудование того времени снабжалось последовательным интерфейсом RS-232. Главным двигателем нового стандарта (USB) была софтверная компания Microsoft, которая к середине 90-х годов прошлого века чуть более чем полностью доминировала на рынке персональных компьютеров и с помощью своего системного ПО продвигала, или даже порой навязывала, пользователям новые стандарты, в том числе на оборудование и физические интерфейсы. Но зачем? Дело в том, что на 90-е годы приходится бум расцвета архитектуры IBM PC известной нам сейчас как «персональный компьютер» (или «ПК»), её массовое проникновение повлекло появление огромного количества различных периферийных устройств, таких как

сканеры, принтеры, звуковые устройства, модемы, внешние накопители информации и т. д. Эти устройства имели разные, несовместимые межу собой способы подключения к ПК - какие-то подключались через «COM-порт» (RS-232), какие-то через параллельный порт принтера «Centronics» (LPT), какие-то через внешний «скази» (SCSI), а некоторые требовали установки в ПК своего собственного адаптера. Я припоминаю, мой первый привод CD-ROM про-ва японской фирмы Mitsumi, приобретенный мной в 1995 году, имел ATAPI интерфейс и для его подключения пришлось купить специальную версию звуковой карты формата ISA на которой присутствовал разъем ATAPI (а еще пришлось заморочиться с поиском драйверов для операционной системы OS/2). В общем, полный разброд и шатание. При том, что пользователю, как правило, хотелось подключать к своему ПК и использовать все эти полезные гаджеты одновременно, а с каждым днём появлялись всё новые. Шина UART, теоретически, могла бы заменить эти разношерстные интерфейсы если бы не ряд врожденных недостатков. Во-первых ни один из стандартов серии EIA RS-232 не предоставлял возможности обеспечить целевому устройству полноценное питание от ПК. Во-вторых, используемые в стандартах EIA RS-232 типы и виды разъемных соединителей были очень громоздкими (DB-9, DB-25) и не очень пригодными для постоянного их подключения/отключения. В третьих, EIA RS-232 за 30 лет сильно «оброс мхом» и содержал избыток редко используемых сигнальных линий — помимо сигналов RXD, TXD и GND, стандарт предписывал наличие в разъеме еще более десятка дополнительных сигнальных цепей, как-то DTR, DSR, RTS, CTS, RI, DCD и т. п., что в общем-то обуславливало габариты разъемов и толщину кабеля. Ну и четвертым немаловажным фактором было то, что RS-232, в виду своей асинхронной природы и с помехо-неустойчивым способом кодирования данных, работал на относительно низких скоростях - до 115200 бод (которые позже увеличили до 1,5 и даже до 12,500 Мегабод), в то время как пользователю требовались «десятки мегабит в секунду». А к началу 2000-х и этого окажется мало! Короче, имеющий широкое хождение стандарт EIA RS-232 был хорош всем, кроме того, что требовал радикальной переработки. И такая переработка состоялась.

15-го января 1996 года консорциум из компаний специализирующихся на производстве ПК-шного железа, среди которых были известные по тем временам бренды Compaq, Hewlett-Packard, Intel, Lucent, NEC и Philips, и возглавляемый корпорацией Microsoft, предложил спецификацию нового стандарт универсальной последовательной шины получившей название Universal Serial Bus Revision 1.0. Данная спецификация предусматривала следующие фичи:

- компактный формфактор разъемного соединителя двух типов (Type A и Type B) позволяющего легко подключать и отключать устройства к ПК на горячую, без отвертки и лишних телодвижений;
- питание устройства от хоста (ПК) напряжением +5 В и до 500 мА на порт;
- два скоростных режима: 1,5 Мбит/сек (Low Speed) и 12 Мбит/сек (Full Speed);
- более устойчивый к помехам метод кодирования передаваемых данных NRZi с использованием двух сигнальных линий вместо одной, который уже успешно использовался в других видах интерфейсов и сетей (FastEthernet);
- возможность соединения устройств каскадом с помощью простого дополнительного устройства — HUB-а, также по аналогии с Ethernet;
- протокол обмена по шине на основе пакетов разных видов, с контрольными суммами для поверки целостности переданных данных, их проверка и автоматическая перепосылка (Retransmission);
- механизмы энергосбережения (Power Management) позволяющие переводить устройство в режим «сна» с низким потреблением энергии в случае если с ним не осуществляется обмен в течении определенного времени;

- возможность автоматической идентификации подключаемых устройств операционной системой ПК, автоматический выбор и загрузка соответствующего драйвера.

В то время компания Microsoft очень усердно продвигала на рынок свою парадигму беззаботного конфигурирования устройств - «Plug and Play», и, как видно из описания, новый стандарт полностью ей соответствовал. Но прошло более 10 лет прежде чем USB действительно начал работать так, как это предполагалось - «вставил и поехал», а на начальных этапах внедрения этого стандартна всё было не очень радужно. Например, подключение манипулятора типа «мышь» через USB могло запросто «подвесить» операционную систему или отправить её в «panic!» (выдать «синий экран смерти» в ОС Windows). Но не будем о грустном.

Как отмечалось выше, первая спецификация стандарта на шину USB предлагала два скоростных режима: «Low Speed» - 1,5 Мбит/сек и «Full Speed» - 12 Мбит/сек, их начали обозначать как USB 1.x «Low Speed» и USB 1.x «Full Speed» соответственно. Режим «Low Speed» предлагался для использования в устройствах ввода, таких как клавиатура, мышь, световое перо, геймпады и джойстики, а режим «Full Speed» для всего остального — модемы, принтеры и сканеры.

Через некоторое время стало понятно, что даже 12 Мбит/сек это катастрофически мало, особенно для таких устройств как звуковые карты и накопители, и в апреле 2000-го года вышла спецификация USB 2.0 описывающая обмен по последовательной шине со скоростью 480 Мбит/сек. Версия 2.0 была обратно совместима с версией 1.0 и включала её как подмножество, что позволяло без проблем подключать старые USB устройства («Low Speed» и «Full Speed») в порт USB 2.0. Но новые устройства, такие как накопители USB Flash, выполненные в соответствии с версией 2.0 не могли работать в портах USB 1.x. Чтобы облегчить пользователям «поиск неисправности там где её нет», было предложено правило — маркировать разъемы различных версий стандарта пластиковыми вставками разного цвета: разъемы портов и устройств версии USB 1.x — белого цвета, а USB 2.0 — черного. Позже к этому правилу добавили USB 3.x, разъемы этой версии маркируются пластиковой вставкой голубого цвета. Помимо этого на корпусах разъемов начали наносить специальные логотипы отличающиеся для разных версий стандарта и указывающие на соответствие изделия стандарту (рис. 2).

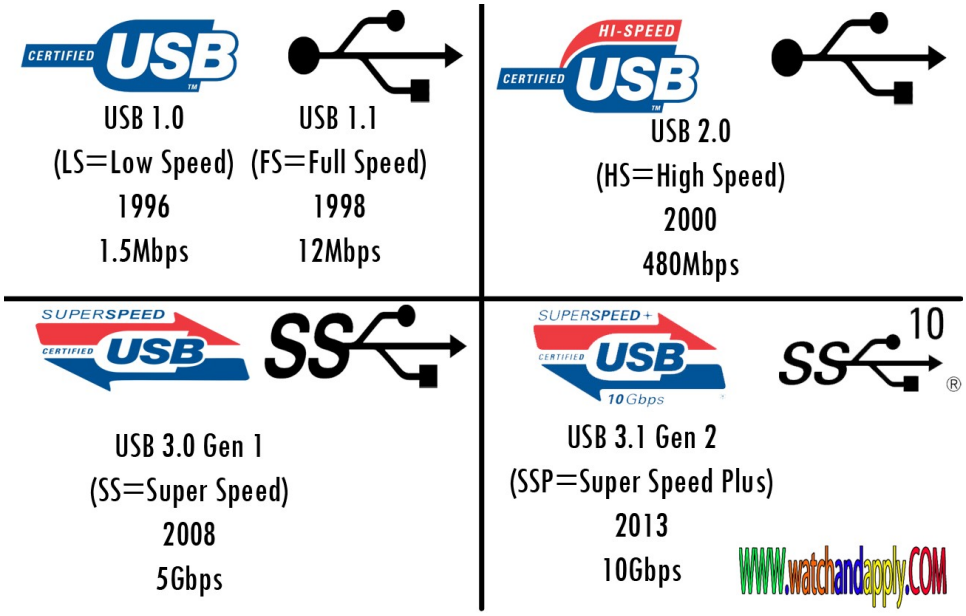

Рис. 2. Логотипы для разъемов USB различных версий.

Далее мы будем рассматривать реализацию на языке SpinalHDL контроллера USB версии 1.x («Low Speed») и немного зацепим USB 1.x («Full Speed»). Версии 2.0 и 3.x шины USB детально рассматривать не будем, хотя и упомянем их основные отличия. Причин для этого две: во-первых, спецификация на шину USB катастрофически сложная даже в самой первой версии — чтобы реализовать и описать её всю требуется существенно больше времени; во-вторых, скорости обмена в 480 Мбит/сек и выше реализовать на ПЛИС можно только с применением специальных проприетарных IP-блоков «сериализаторов» (SERDES) с поддержкой LVDS, а их использование сделает нашу имплементацию контроллера непереносимой и привязанной к конкретному производителю микросхем ПЛИС, чего хотелось бы избежать.

2. Шина USB на физическом уровне

2.1. Электрические характеристики

На физическом уровне шина USB версии 1.0 и 1.1 (и 2.0 тоже) содержит всего четыре проводника (см. рис. 3 ниже). Два из них, VBUS и GND используются для передачи питания от хоста/ПК («host») к устройству («device»). Напряжение питания передается по линии VBUS и может варьироваться от **4,75** до **5,25 В**. На практике большинство хостов выдает напряжение по верхней границе 5,25 В для компенсации падения напряжения в кабеле при большом токе. Максимальный ток отбираемый устройством из одного USB порта по-умолчанию не должен превышать **0,1 А**, однако предусматривается возможность отдавать в устройство ток до **0,5 А**. При этом устройство должно заявить в хост, на стадии инициализации, какой конкретно ток оно собирается потреблять, а хост в праве «разрешить» или «отказать». Если устройству для работы требуется ток больше 0,5 А, то предусмотрена возможность запитки от двух USB портов специальным разветвленным кабелем с двумя разъемами Type A на одном конце. Для USB 3.x и разъема типа Type A максимальный отбираемый ток повышен до **0,9 А**. Для разъема Type C максимальный допустимый ток составляет уже **1,5 А** и напряжение по линии VBUS может программно изменятся в широких пределах. На просторах сети Интернет обнаружилась сводная таблица (см. Таб. 1) с электрическими характеристиками шины USB различных версий.

Таблица 1. Электрические характеристики шины USB

Спецификация	Макс. ток	Напряже-ние	Макс. мощность
Low-power device / USB 1.x, 2.0	100 mA	5 V	0.50 W
Low-power SuperSpeed / USB 3.x device	150 mA	5 V	0.75 W
High-power device / USB 1.x, 2.0	500 mA	5 V	2.5 W
High-power SuperSpeed / USB 3.x single-lane device	900 mA	5 V	4.5 W
High-power SuperSpeed / USB 3.x dual-lane device	1.5 A	5 V	7.5 W
Battery Charging (BC 1.2)	1.5–2.4 A	5 V	7.5–12 W
USB4	1.5 A	5 V	7.5 W
Type-C current (1.5 A)	1.5 A	5 V	7.5 W
Type-C current (3 A)	3 A	5 V	15 W
Power Delivery SPR	5 A	Up to 20 V	100 W

Power Delivery EPR	5 A	Up to 48 V	240 W

Важно отметить, что потребителем электроэнергии на порту USB может быть как конечное устройство, так и концентратор (HUB), который в свою очередь может раздавать питание подключенным к нему потребителям далее в низ по цепочке. Но суммарный ток всех устройств в этой «гирлянде» не должен превышать установленный спецификацией для версии порта.

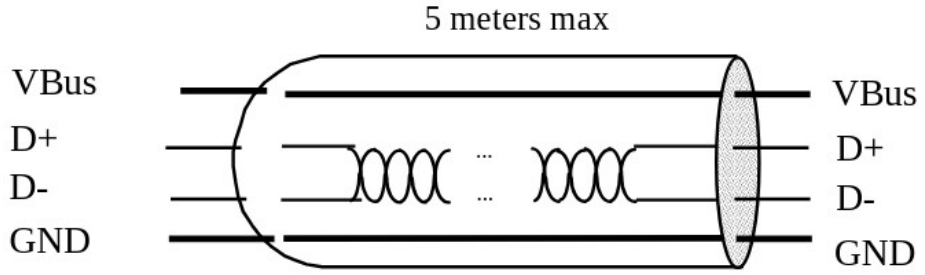

Рис. 3. Физический уровень USB: четыре проводника — два сигнальных и два для питания.

Теперь рассмотрим как в USB 1.x передаются данные. Для этого, как уже отмечалось выше, предназначены сигнальные линии D+ и D-. Каждая их этих линий может находиться в двух состояниях: либо низкого напряжения «low» (**0...0,3 В**), либо в состоянии высокого напряжения «high» (**2,8...3,6 В**). Всего у шины USB четыре поименованных состояния образованных комбинацией состояний линий D+ и D-: состояние «Idle» (или **«J»**), состояние «Inversed» (или **«K»**), состояние «Single-ended Zero» (или **«SE0**») и состояние «Single-ended One» (или «**SE1**»). При этом для USB 1.x «Low-Speed» и USB 1.x «Full-Speed» эти состояния определены по-разному. Чтобы не запутаться предлагаю сразу посмотреть в таблицу 2 позаимствованную со страницы Wikipedia посвященной протоколу обмена USB.

Таблица 2. Состояния шины USB версии 1.0.

Название сигнала	**Состояние шины**	**Описание процесса на шине**	**Low speed (подтяжка на D−)**		**Full speed (подтяжка на D+)**	
			D+	**D−**	**D+**	**D−**
«J»	Idle	Бездействие или переходное состояние при передаче данных.	low	high	high	low
«K»	Inverse of J	Переходное состояния при передачи данных	high	low	low	high
«SE0»	Single-ended zero	Обе линии D+ и D− подтянуты к нулю. Может указывать на конец пакета, на состояние сброса («reset») или отсутствия подключения («disconnect»).	low	low	low	low
«SE1»	Single-ended one	Обе линии D+ и D- подтянуты к положительному потенциалу. Запрещенное состояние, указывает на аварию.	high	high	high	high

Как видно из описания, в стандарте «Low Speed» и «High Speed» сигнальные линии D+ и D- в шине USB 1.x не являются дифференциальной парой, то есть они не строго

симметричные, а заданы относительно общей линии «земли» (GND) и обычно не требуют терминации (нагрузочного резистора).

Напротив, шина USB версии 2.0 («High Speed») работает как настоящая дифференциальная пара, уровни напряжений в ней определены как **-0,4...+0,4 В** (или дифференциальное 0,8 В), передача данных осуществляется токовым способом — одна линия является источником тока **17,7 мА**, а другая его потребителем. Шина USB 2.0 должна быть терминирована на обеих сторонах либо резистором 90 Ом между D+ и D-, либо двумя резисторами по 45 Ом между D+/D- и центральной точкой («землей»). Перед тем как перейти в дифференциальный режим («High Speed»), устройство USB версии 2.0 сначала работает в режиме «Full Speed».

На стороне хоста линии D+ и D- всегда подтянуты к низкому «low» напряжению резисторами **15 кОм**, что по-умолчанию и при отсутствии подключения переводит шину в состояние «SE0» или состояние сброса (более 2,5 мс) и далее в состояние «disconnect». Для того, чтобы определить какой тип устройства на данный момент подключен к шине, «Low Speed» или «Full Speed», на стороне подключаемого устройства одну из сигнальных линий жестко подтягивают к высокому «high» напряжению резистором сопротивлением **1,5 кОм** (см. рис. 4). Для этого обычно используется напряжение внутренней шины питания - 3,3 В. Этот резистор перетягивает линию, подтянутую на стороне хоста к «low» в сторону «high» и соответственно переводит шину в состояние «Idle» («J»). Это состояние детектируется хостом и запускается процедура инициализации. Но об этом позже, а сейчас рассмотри как кодируются данные на шине USB.

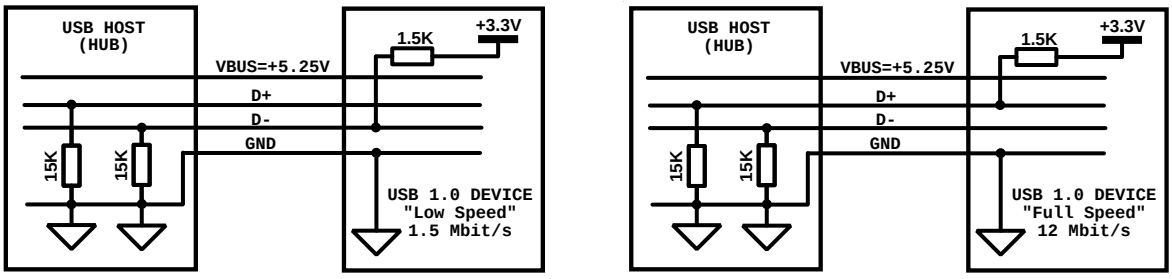

Рис. 4. Физический уровень USB: резисторы подтяжки при подключении «Low Speed» (слева) и «Full Speed» (справа) устройства.

2.2. Передача данных по шине USB

Несмотря на то, что для «Low Speed» и «Full Speed» состояния «J» и «K» кодируются по-разному, протокол оперирует именно состояниями («J» и «K») для описания процесса передачи данных. При имплементации нам надо быть внимательными и не перепутать что есть «J» и что есть «K» для конкретной версии USB. Так как мы будем реализовывать только версию USB 1.0 «Low Speed» то запомним, что пассивное состояние «J» для неё определяется как D- в состоянии «high», а D+ в состоянии «low». Состояние «K» будет строго противоположным: D- в состоянии «low» и D+ в состоянии «high».

Обмен данными на шине USB ведется пакетами нескольких типов. Инициатором обмена всегда выступает хост, он передает запрос устройству, а устройство либо отвечает одним и более пакетом, либо молчит.

Перед началом передачи шина USB всегда должна находится в состоянии Idle («J») некоторое время равное минимум двум интервалам. Каждый пакет начинается с передачи преамбулы «SYNC» — чередующаяся последовательность состояний «K» и «J» длиной шесть интервалов («KJKJKJ»), после которой следует два состояния «K» («KK»).

Длительность нахождения шины в одном из таких состояний (один интервал) определяется частотой работы шины — 1,5 или 12 МГц/сек. На передачу «K» и «J» можно смотреть как на передачу битов в UART, но важно понимать, что это не биты данных, а кратковременные состояния шины. Таким образом каждый пакет начинается с последовательности из восьми интервалов преамбулы («KJKJKJKK») которые используются для синхронизации приемника по частоте и фазе. Следом за преамбулой передаются восемь бит типа пакета («Packet ID» или «PID») за которым могут следовать дополнительные служебные данные и/или полезные данные. Пакет всегда заканчивается последовательность состояний «SE0 SE0 J», то есть два интервала SE0 (D+ и D- подтянуты к «low») и один интервал состояния «J». По завершению передачи шина всегда остается в состоянии «J» («Idle») и через один интервал готова к передаче следующего пакета.

Кодирование битов данных в USB 1.0 и 1.1 происходит следующим образом: если требуется передать бит «0», то передающая сторона инвертирует состояние шины: было «K» стало «J», и наоборот — было «J» стало «K». Чтобы передать бит «1» передающая сторона НЕ изменяет состояние шины в следующем интервале времени, т. е. было «K» и осталось в «K», было в «J» и осталось в «J». Например, для передачи двух нулей («00») сразу после преамбулы, которая заканчивается состоянием «K», на шине будут последовательно выставлены следующие состояния - «JK», а для передачи двух единиц («11») - «KK».

Важным моментом является то, что тип передаваемого состояния зависит от бита передаваемой информации и от предыдущего состояния шины!

На рис. 5 ниже приведен пример последовательности смены состояний шины USB версии 1.1 («Full Speed») при передаче одного пакета типа «NAK». В данном случае состояние «J» кодируется как D+ в «high» и D- в «low». Передача начинается из состояния «J» («Idle») с преамбулы «KJKJKJKK», за которой следует восемь битов типа пакета «JJKKKJJK» («01011010»), а за ними сразу следует признак конца пакета «00J» (или SE0, SE0, J).

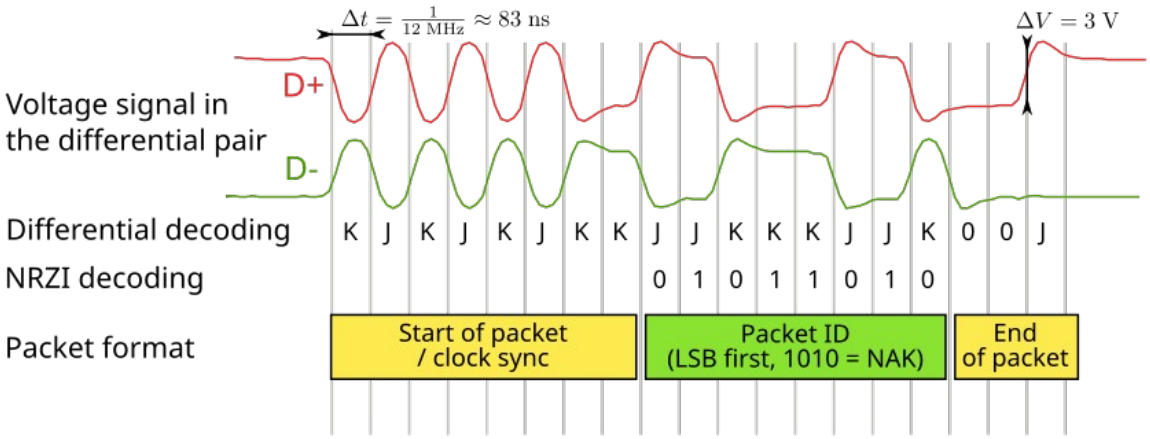

Рис. 5. Последовательность смены состояний шины USB 1.1 «Full Speed» при передаче пакета типа NAK (Wikipedia).

Еще одой особенностью кодирования данных на шине USB является использования «бит-стаффинга» — методики позволяющей избежать долгое нахождение шины в каком либо из состояний («K» или «J») при передачи пакета данных. Это необходимо для двух целей: 1) позволить принимающей стороне постоянно пересинхронизироваться от фронтов смены полярностей линий D+ и D-, и 2) удаление постоянной составляющей электрического тока для большей помехозащищенности. Бит-стаффинг в USB работает следующим образом: при передаче шести и более последовательных «единиц» (а «единицы» кодируются отсутствие

смены состояния шины), в передачу вставляется одна дополнительная смена состояний. Например, если шина находится в состоянии «K» и с этого момента требуется передать шесть «единиц» («111111»), то последовательность состояний можно описать как «KKKKKKJ» - это эквивалентно вставке в битовый поток «нулевого» бита (в сумме получится последовательность бит «1111110»). На стороне приемника этот дополнительный «нулевой» бит отслеживается и удаляется из потока. Это значит, что каждая последовательность «KKKKKKJ» или «JJJJJJK» заменяется на шесть «единиц» («111111»).

Если принимающая сторона по какой-то причине получает семь последовательных «единиц», то произошел сбой передачи!

2.3. Передача служебных сигналов

Спецификацией USB 1.0 и 1.1 предусматривается передача по шине USB служебных сигналов влияющих на работу устройств подключенных к шине. Подача сигналов производится с помощью комбинации различных состояний и интервала времени больший чем отводится при передаче одного бита данных. Что-то подобное есть и в старом UART (я имею в виду передачу сигнала BREAK), но в USB спектр служебных сигналов гораздо богаче.

В спецификации USB 1.x различают следующие служебные сигналы (или состояния) шины USB:

- «**Disconnect**» - отключение устройства. Если шина находится в состоянии «SE0» более 2 мкс, что считается что связь с устройством потеряна. Такое возможно если на шине нет физических подключений кроме 10 кОм резисторов подтяжки к «low» установленных в схеме хоста.
- «**Connect**» - подключение устройства. Если шина переходит из состояния «Disconnect» в состоянии «Idle» («J»). При этом «J» определено по разному для «Low Speed» и «High Speed», что позволяет определить какого типа устройство подключено к шине.
- «**Idle**» («J») - пассивное состояние без обмена данными, см. главу «*Передача данных по шине USB*».
- «**EOP**» - конец передачи (конец пакета). Если шина последовательно проходит состояния «SE0», «SE0» и «J» в течении трех временных интервалов.
- «**Reset**» - сброс. Инициируется хостом путем перевода шины в состояние «SE0» и удержание её в этом состоянии более 2,5 мс. На стороне устройства прием сигнала «сброс» обязан приводить к сбросу внутренней машины состояний.
- «**Bus Reset**» - полный сброс шины. Осуществляется переводом шины в состояние «SE0» на время от 10 до 20 мс. После «полного сброса» полноценное взаимодействие с устройством возможно только после поведение процедуры инициализации. Полный сброс обычно выполняется после детектирования подключения нового устройства, чтобы гарантировать его полную переинициализацию.
- «**Suspend**» - переход в режим энергосбережения («сна»). Инициируется хостом путем перевода шины в состояние «J» и удержание её в этом состоянии в течении более 3 мс. Устройство получив такой сигнал обязано перейти в режим энергосбережения. Дальнейшая коммуникация с устройством возможна только либо подачи сигнала «Resume», либо после подачи сигнала «Reset» с последующей инициализацией.
- «**Resume**» - выход из режима энергосбережения (выход из «сна»). Осуществляется подачей сигнала «K» в течении 20 мс с последующей подачей сигнала «EOP». Сигнал «Resume» может быть инициирован устройством, для этого устройство переводит шину в состояние «K» на время более 1 мс и ожидает сигнала «Resume» от хоста.

- «**Keepalive**» - не позволять устройству переходить в режим энергосбережения. Осуществляется передачей сигнала «EOP» каждую миллисекунда. Определено только для «Low Speed» устройств.
- «**SOF**» - признак начала временного фрейма (Start of Frame). Пакет данного формата рассылается хостом по шине «Full Speed» или «High Speed» с интервалом в 1 мс.

Важным моментом является то, что если устройство перешло в режим энергосбережения («сна»), то взаимодействовать с ним можно только после подачи сигнала «Resume» или «Reset», что занимает некоторое время. Если устройство перешло в режим «сна», то оно прекращает отвечать на запросы. ***Для того, чтобы устройство не переходило в режим «сна», хост должен периодически, с интервалом 1 мс, посылать по шине сигнал «Keepalive». Для режима «Full Speed» сигнал «Keepalive» не определен, вместо этого в данном режиме хост обязан высылать пакет «Start-of Frame» (SOF) с интервалом в 1 мс. И «Keepalive» и «SOF» используются в том числе для синхронизации внутренних часов на устройстве. Хаб формирует «SOF» для всех свих портов самостоятельно исходя из своего внутреннего счетчика времени.***

3. Протокольный уровень шины USB

Как было отмечено выше, в процессе обмена по шине USB данные передаются пакетами. Каждый пакет начинается с преамбулы «SYNC», содержит идентификатор типа пакета «PID», полезные данные и заканчивается сигналом «EOP» (последовательность состояний: «SE0», «SE0» и «J»). При передаче данные проходят процедуру «бит-стаффинга» - на передающей стороне к каждой последовательности из шести «единиц» добавляется один «ноль», а на приемной этот «ноль» удаляется. Данные передаются от младших битов к старшим, от младших байтов к старшим (принцип «LSB First»).

Поле «PID» содержит восемь бит из которые первые четыре бита непосредственно задают тип, а следующие четыре бита повторяют первые два в инверсном состоянии. Это сделано для того, чтобы можно было детектировать правильность приема простых пакетов не содержащих защитных циклических кодов (CRC). В стандартах USB всего возможно 16 типов пакетов, из них в версии 1.0 и 1.1 задействовано только восемь: ACK, NAK, STALL, IN, OUT, SETUP, DATA0 и DATA1. Далее мы рассмотри назначение этих пакетов.

3.1. Типы пакетов (PID) для USB 1.x

По смысловому назначению пакеты передаваемые по шине USB объединяются в три группы: Токены (Token), Управление потоком (Handshake) и пакеты передачи данных (Data). В приведенной ниже таблице 3 описаны сгруппированные по назначению используемые в USB 1.0 и 1.1 идентификаторы пакетов. Биты идентификатора «PID» указаны в таблице так, как они следуют в пакете — слева направо, от младшего бита к старшим.

Таблица 3. Перечень идентификаторов пакета (PID) для USB 1.0 и 1.1

PID	Типа пакета	Наименование	Назначение
0100 1011	Handshake	ACK	Подтверждение принятого пакета с данными.
0101 1010	Handshake	NAK	«Нет данных» для ответа.

0111 1000	Handshake	STALL	Возникла неразрешимая ошибка, требуется выполнение процедуры восстановления (error recovery).
1011 0100	Token	SETUP	Задание адреса конечной точки («endpoint») для последующего обмена служебной информацией между хостом и устройством.
1001 0110	Token	IN	Задание адреса конечной точки («endpoint») для последующей передачи данных от устройства к хосту (Device-to-Host transfer).
1000 0111	Token	OUT	Задание адреса конечной точки («endpoint») для последующей передачи данных от хоста к устройству (Host-to-Device transfer).
1010 0101	Token	SOF	Начало временного интервала (Start of Frame), используется как источник синхронизации часов на устройстве и точки перепосылки в изохронном режиме.
1100 0011	Data	DATA0	Четный пакет с данными.
1101 0010	Data	DATA1	Нечётный пакет с данными.

Все три вида пакетов имеют свой формат и свой способ защиты от ошибки при передачи (свой код CRC, который рассмотрен ниже).

3.1.1. Пакеты управления потоком (ACK, NAK, STALL)

Пакеты управления потоком содержать только PID и не защищены с помощью CRC, проверка их правильности осуществляется сверкой старшего и младшего полубайта идентификатора пакета, биты которых должны быть инвертированы друг относительно друга. К управлению потоком относятся пакеты с идентификаторами PID равным ACK, NAK и STALL. Они используются для управления передачей данных и обычно посылаются в ответ на пакет с данными DATA0 или DATA1 (вместе DATAx) с целью подтвердить правильность получения данных на приемной стороне или сигнализировать об ошибке (невозможности дальнейшей передачи). Формат управляющего пакета приведен в таблице 4.

Таблица 4. Формат пакета управления потоком (Handshake).

Поле	**SYNC**	**PID**	**EOP**
Битовые интервалы	**8**	**8**	**3**
Сигнал	KJ KJ KJ KK	XXXX XXXX	SE0 SE0 J

Передача управляющего пакета занимает 16 битовых интервалов + три битовых интервала на сигнал конца пакета (EOP), итого 19 битовых интервалов. Назначение пакетов следующее:

Пакет ACK — высылается в ответ на принятый пакет DATAx если пакет с данными был принят успешно, т. е. без ошибок и передающая сторона может продолжать передачу.

Пакет NAK — высылается если в ответ на запрос и сообщает, что у запрашиваемой стороне сейчас нет данных или она не готова. Отправка NAK означает, что передающая сторона должна повторить отправку пакета с данными (запрос) через некоторое время.

Пакет STALL — высылается в ответ на принятый пакет DATAx в том случае, если на принимающей стороне произошла невосстановимая ошибка (неверный формат данных, не поддерживаемая функция или сбой конечного автомата). Посылка этого пакета означает, что дальнейшая передача данных невозможна и требуется частичная или полная инициализация устройства.

3.1.2. Пакеты-токены (IN, OUT, SETUP и SOF)

К так называемым «токенам» относятся пакеты с идентификаторами IN, OUT и SETUP. Пакеты этой группы, помимо идентификатора PID содержат 7 битов адреса устройства («device address»), 4 бита номера конечной точки («endpoint») и пять битов контрольной суммы CRC5. Контрольной суммой защищается всё, что следует за PID. Таким образом в токенах под защиту CRC5 попадают поля «device address» и «endpoint». Общий формат токена приведен в таблице 5. Поля «адрес устройства» и «номер конечной точки» («endpoint») принято сокращенно именовать ADDR и ENDP.

Адрес устройства — это номер устройства присвоенный ему в процессе инициализации. Если устройство не было инициализировано, то оно отвечает на адрес «0». Номер конечной точки это виртуальная сущность — номер некоего приложения (канала, буфера) внутри устройства с которым в конкретный момент времени происходит взаимодействие. Нулевой («0») номер конечной точки зарезервирован для служебных нужд и активно используется при инициализации устройства. Таким образом, все USB устройства поддерживают обмен с ADDR:ENDP = 0:0.

Важно, что после завершения инициализации устройство может сменить свой номер, при этом номера конечных точек на устройстве не меняются, а комбинация 0:0 всегда остается доступной.

Таблица 5. Формат пакета-токена (Token).

Поле	**SYNC**	**PID**	**Device address (ADDR)**	**Endpoint (ENDP)**	**CRC5**	**EOP**
Битовые интервалы	**8**	**8**	**7**	**4**	**5**	**3**
Сигнал	KJ KJ KJ KK	XXXX XXXX	XXXX XXX	XXXX	XXXXX	SE0 SE0 J

Передача пакета типа «Token» занимает 32 битовых интервала + три бита на сигнал конца пакета (EOP), итого 35 битовых интервалов. Рассмотрим подробнее назначение каждого из токенов.

Токен SETUP используется для инициализации устройства или его перенастройки. За токеном SETUP всегда следует пакет DATA0 стандартизованного формата (см ниже) содержащий ровно 8 байт данных. Как правило, первоначальная процедура инициализации направляет SETUP на адрес 0:0.

Токен OUT передается от хоста к устройству адрес которого задан в поле ADDR и сообщает устройству что следом за этим пакетом будет передан один и более пакет с данными (DATAx) предназначенными для указанной в поле ENDP конечной точки (приложения). После получения пакета с данными устройство обязано подтвердить успешность приёма каждого пакета отправкой пакета ACK, NAK или STALL.

Токен IN передается от хоста к устройству адрес которого задан в поле ADDR и сообщает устройству что после этого пакета хост ожидает от устройства пакеты с данными (DATAx), то есть будет производиться передача данных от устройства к хосту. Аналогично поле ENDP указывает на номер конечной точки на устройстве от которой хост желает получать данные. Устройство получив пакет IN обязано ответить с помощью NAK или STALL или начать передачу данных с помощью пакетов DATAx. Устройство возвращает NAK если в данные момент у него нет данных для отправки. STALL возвращается если произошла внутренняя ошибка в конечном автомате устройства и дальнейшее взаимодействие невозможно. На каждый полученный пакет DATAx хост обязан ответить с помощью ACK или NAK (данные приняты без ошибки или с ошибкой), при этом NAK от хоста сигнализирует устройству о том, что требуется перепосылка (retransmission).

Токен SOF передается от хоста по шине работающей в режима «Full Speed» с интервалом 1 мс. Формат пакета SOF отличается от остальных токенов тем, что в 11-ти битном поле ADDR:ENDP вместо адреса устройства и конечной точки передается число — номер временн**о**го интервала (фрейма). Этот номер монотонно возрастает с каждым новым фреймом. Токен SOF используется для нумерации временных отрезков, что используется для синхронизации в изохронном режиме передачи - хост или устройство могут попросить друг друга возобновить передачи потока данных с какой-то конкретной точки во времени, определяемом номером в SOF.

Стандартом USB 1.0 предписывается приоритет токена SOF над всеми остальными видами пакетов. Это означает, что все данные и служебные пакеты (токены, подтверждения) должны передаваться между двумя последовательными токенами SOF ограничивающих временной отрезок. Никакой токен или пакет с данными не может быть передан вместо токена SOF.

3.1.3. Пакеты с данными (DATA0 и DATA1)

Для передачи полезных данных в USB 1.0 и 1.1 используется два идентификатора PID: DATA0 и DATA1. Пакеты с данными должны следовать только после одного из токенов (IN, OUT или SETUP) которые задают адрес устройства и номер конечной точки для обмена данными. Размер непосредственно поля для полезных данных в спецификации USB 1.0 и 1.1 определен следующим образом: **не более 8 байт** для «Low Speed» и **не более 64 байт** для «Full Speed». В USB 2.0 размер поля для полезных данных увеличен до 1024 байт. Пакет с данными всегда содержит поле циклического кода CRC16 для определения ошибки возникшей при передаче. Полный формат пакета данных приведен в таблице 6.

Таблица 6. Формат пакета с данными (DATA0 и DATA1).

Поле	SYNC	PID	Data	CRC16	EOP
Битовые интервалы	**8**	**8**	**64**	**16**	**3**
Сигнал	KJ KJ KJ KK	XXXX XXXX	XXXX … XXXX	XXXX XXXX XXXX XXXX	SE0 SE0 J

Семантика использования пакетов DATA0 и DATA1 следующая. Передающая сторона начинает передачу с пакета содержащего PID равный DATA0. Получив подтверждение (ACK), передающая сторона передает пакет с идентификатором DATA1. Получив ACK и на этот пакет, передающая сторона передает пакет с DATA0 и так по кругу. Таким образом, DATA0 и DATA1 отражают последний бит в в номере передаваемого пакета в потоке данных. Передающая сторона обязана дождаться подтверждения ACK на каждый отправленный пакет и только после получения подтверждения переходит к отправке следующего, изменяя при этом PID. Если же в ответ на пакет с данными не пришло подтверждения по истечению некоего времени (ситуация «таймаут»), то считается что пакет полностью потерян (не дошел до приемной стороны), и передающая сторона еще раз высылает предыдущий пакет не меняя его PID.

На приемной стороне на все успешно полученные пакеты (не содержащие ошибок CRC16) высылается ACK, но данные извлекаются только из пакетов с чередующимся PID, отбрасывая дубликаты (два и более пакета с одним и тем же PID). Если на приемной стороне возникает ошибка CRC16, то приемная сторона просто отбрасывает полученный пакет и ничего в ответ не высылает, что приводит к срабатыванию таймаута на передающей стороне и повторной посылке того же пакета с таким же PID. Управляющий пакет NAK отсылается только в том случае, если приемная сторона желает разорвать сеанс передачи данных с данным устройством и конечной точкой. Также NAK часто используется для сигнализации о неготовности стороны к передаче или приему данных.

Важно, что PID последнего отправленного пакета с данными учитывается отдельно для конкретной пары ADDR:ENDP, а это означает, что конечный автомат на стороне хоста должен запоминать PID для последнего отправленного пакета и последнего принятого пакета раздельно для каждой пары ADDR:ENDP. Каждый раз после пересылки IN, OUT или SETUP для заданной пары ADDR:ENDP номер пакета в потоке сбрасывается и передача начинается с PID = DATA0. Сбрасывается номер пакета также и при выполнении процедуры инициализации.

3.1.4. Алгоритмы расчета контрольных сумм CRC5 и CRC16

Вычисление циклических контрольных сумм («Cyclic Redundency Check») является традиционным способом обнаружения ошибок при передачи данных по каналам связи. Принцип строится на том, чтобы для каждого блока передаваемых данных вычислить число небольшой битности (обычно 8, 16 или 32 бита) таким образом, что при изменении любого бита в исходном пакете данных значение суммы тоже будет изменено. Т.е. случайная модификация одного или двух битов при передаче должна вызывать несовпадение на стороне приемника суммы переданной вместе с данными и вычисленной при приеме данных.

Традиционно алгоритмом для расчета CRC является вычисление остатка от деления большого числа, составленного из передаваемых данных, и какого либо известного числа-делителя заданного размера. С математической точки зрения деление больших целых чисел представляет собой деление с остатком полиномов в поле Галуа GF(2) — каждый бит делимого, делителя и остатка можно представить как коэффициент (0 или 1) у

соответствующего члена полинома, например полином 4-й степени вида $\mathbf{x^4+x^3+1}$ в коде выглядит как двоичное число **0b11001**. Полином ***P(x)*** степени ***M*** представляет собой передаваемые данные, где ***M+1*** — число передаваемых бит. Полином ***G(x)*** степени ***N*** представляет собой делитель размерностью ***N+1*** бит — порождающий полином на который делят ***P(x)***. После выполнения деления остаток ***R(x)*** будет иметь степень на единицу меньше порождающего полином (всего 2^N-1 остатков или ***N*** бит). Для улучшения качества результирующего кода на коротких последовательностях входных данных (т. е. чтобы сделать его более чувствительным к повреждению данных), ***P(x)*** умножают на x^N , то есть приписывают ***N*** нулевых битов справа (т. е. смещают влево на ***N*** бит). Тогда вычисление CRC это: $R(x) = P(x) * x^N \bmod G(x)$. В двоичной системе остаток вычисляется путем циклического вычитания делителя с помощью операции XOR и смещения разрядов делимого — отсюда и название «циклическая сумма».

Вычисленный таким способом остаток передается в качестве контрольной суммы и проверяется на принимающей стороне. Очевидно, что от делителя (формы порождающего полинома - ***G(x)***) в некоторой степени зависит то, насколько хорошо данный алгоритм позволяет обнаруживать ошибки. Также очевидно что делителем не может быть число ноль. Поэтому форму порождающего полинома подбирают под конкретный случай, используют *неприводимые* полиномы, а входные данные в алгоритме расчета CRC подвергают дополнительной обработке — добавляют биты в исходную последовательность, инвертируют биты или меняют порядок их следования, либо и то и другое одновременно. Если в передаваемых данных из которых составляется делимое содержатся лидирующие нули, то полученная сумма не будет чувствительна к «повреждениям» в этих битах. Чтобы алгоритм был чувствителен к таким случаям, в самом начале к результату прибавляется какое-то известное число (например, максимальное число данной разрядности). Более детально об алгоритмах расчета контрольных сумм можно ознакомиться в соответствующей статье на Wikipedia.

Интересно то, что если порождающий делитель содержит всего один бит равный единице, то остаток ***R(x)*** представляет собой «бит четности». Такой способ защиты данных от повреждения при передачи также широко используется, например во всё том же UART.

В USB пакетах поле CRC5 размером пять бит содержит циклическую контрольную сумму и используется для обнаружения ошибки передачи пакетов типа «Token», при этом под защиту попадают только данные полей адреса устройства (ADDR) и номера конечной точки (ENDP), то есть всего одиннадцать бит информации. Поле типа пакета (PID) защищено инвертированной четырех-битовой копией самого себя и это считается весьма надежным.

Размер контрольной суммы в пять бит во-первых позволяет обеспечить высокую вероятность обнаружения ошибки при передачи токенов, а во-вторых удачно выравнивает пакет с токеном по границе 16 бит (не учитывая преамбулу «SYNC», она не несет в себе полезной информации). В стандарте USB для расчета CRC5 используется порождающий полином: $\mathbf{x^5 + x^2 + x^0}$, в коде это представляется как **0b100101**. Код порождающего полинома всегда на один бит длиннее чем размер остатка (в данном случае он имеет длину 6 бит), но так как у него всегда старший бит равен единице, то эту единицу принято не показывать, а значит код порождающего полинома для алгоритма CRC5-USB равен **0b00101 (0x05)**.

При вычислении остатка начальное значение для расчета устанавливают в **0b11111**, а результат побитно инвертируют операцией **XOR** и записывают в обратной последовательности (**reversed**). Пример программы на языке Си для расчета контрольной суммы CRC5-USB представлен ниже в листинге 1.1. В ней для удобства вычислений код порождающего полинома, его принято называть «polynomial», также представлен в инвертированном виде без лидирующей единицы.

Листинг 1.1. Программа для расчета CRC5-USB на языке Си.

```
// https://github.com/pointcheck/code_snippets/tree/master/C/crc5usb

// Courtesy: https://electronics.stackexchange.com/questions/718294/how-is-crc5-calculated-in-detail-for-a-usb-token

#include <stdio.h>
#include <stdlib.h>

#define REV_POLYNOMIAL  0x14 // Reversed polynomial: 0b00101

unsigned char crc5usb(unsigned short input)
{
	unsigned char res = 0x1f;
	unsigned char b;
	int i;

	for (i = 0;  i < 11;  ++i) {
		b = (input ^ res) & 1;
		input >>= 1;
		if (b) {
			res = (res >> 1) ^ REV_POLYNOMIAL; /* 10100 */
		} else {
			res = (res >> 1);
		}
	}
	return res ^ 0x1f;
}

int main(int argc, char *argv[]) {
	unsigned short data = 0x0000;
	unsigned char mask = 0x01;


	if(argc > 1)
		data = strtol(argv[1], NULL, 0);

	unsigned char crc = crc5usb(data);

	printf("CRC5-USB: data = 0x%03x, crc = 0x%02x, bits to send: ",
data, crc);

	for(int i = 0; i < 5; i++)
		printf("%d ", (crc & mask) ? 1 : 0), mask <<= 1;

	printf("\n");

	return crc;
}
```

Данная программа позволяет рассчитать значение пяти бит кода CRC5 при входных 11 битах данных, так, что младшие 7 бит входного значения представляют поле ADDR, а старшие 4 бита — поле ENDP. Выходное значение выдается в виде шестнадцатеричного кода, а также в виде последовательности нулей и единиц которые необходимо отправить при передачи токена. Стоит напомнить, что биты в USB пакете передаются от младшего с старшему («LSB goes first»). Ниже в листинге 1.2 приведен прогон программы для четырех различных значений пары ADDR:ENDP.

Листинг 1.2. Пример вызова программы CRC5-USB для некоторых вариантов ADDR:ENDP.

```
$ ./crc5usb 0x000 # addr = 0, endp = 0
CRC5-USB: data = 0x000, crc = 0x02, bits to send: 0 1 0 0 0

$ ./crc5usb 0x001 # addr = 1, endp = 0
CRC5-USB: data = 0x001, crc = 0x1d, bits to send: 1 0 1 1 1

$ ./crc5usb 0x080 # addr = 0, endp = 1
CRC5-USB: data = 0x080, crc = 0x14, bits to send: 0 0 1 0 1

$ ./crc5usb 0x081 # addr = 1, endp = 1
CRC5-USB: data = 0x081, crc = 0x0b, bits to send: 1 1 0 1 0
```

Для защиты более длинных пакетов передаваемых по шине USB, таких как пакеты DATAx, применяется другой формат контрольной суммы, а именно — «CRC16-USB». Согласно документу «CYCLIC REDUNDANCY CHECKS IN USB» для расчета CRC16 используется порождающий полином 16-й степени вида: $x^{16} + x^{15} + x^2 + x^0$, что в двоичном коде без лидирующей единицы дает: **0b1000000000000101** (или **0x8005**). Параметры алгоритма CRC16-USB аналогичные: начальное значение остатка для расчета устанавливают в **0xffff** (все 16 единиц), а результат побитно инвертируют операцией **XOR** и записывают в обратной последовательности (**reversed**). Пример программы на языке Си для расчета контрольной суммы CRC16-USB представлен ниже в листинге 2.1. В данной программе код порождающего полинома также представлен в обратном порядке без лидирующей единицы и равен **0xa001**.

Листинг 2.1. Программа для расчета CRC16-USB.

```
// https://github.com/pointcheck/code_snippets/tree/master/C/crc16usb

#include <stdio.h>
#include <stdlib.h>

#define REV_POLYNOMIAL  0xa001 // Reversed polynomial: 0b1000000000000101

unsigned short crc16usb(unsigned short input, unsigned short res)
{
      unsigned short b;
      int i;

      for (i = 0;  i < 16;  ++i) {
            b = (input ^ res) & 1;
            input >>= 1;
            if (b) {
                  res = (res >> 1) ^ REV_POLYNOMIAL; /* 1010 0000 0000
0001 */
            } else {
                  res = (res >> 1);
            }
      }
      return res;
}

int main(int argc, char *argv[]) {
      unsigned short data;
      unsigned short crc = 0xffff; // initial value
```

```
	unsigned short mask = 0x001;

	if(argc < 2) {
		printf("Usage: %s <16bit_data0> <16bit_data1>
<16bit_data2> ... <16bit_dataN>\n", argv[0]);
		return -1;
	}

	argv++;

	while(argc-- > 1) {

		data = strtol(*argv++, NULL, 0) & 0xffff; // only low 16 bits
are valid

		printf("CRC16-USB data: 0x%04x\n", data);

		crc = crc16usb(data, crc);
	}

	crc = crc ^ 0xffff; // XOR all bits as per USB specification

	printf("CRC16-USB: crc = 0x%04x, bits to send: ", crc);

	for(int i = 0; i < 16; i++)
		printf("%d ", (crc & mask) ? 1 : 0), mask <<= 1;

	printf("\n");

	return crc;
}
```

При вызове программе **crc16usb** передается блок данных в виде списка 16-ти битных слов. Программа последовательно вычисляет значение CRC16-USB для каждого слова отдельно, сохраняя предыдущее состояние, и выводит результирующий код в самом конце. В листинге 2.2 приведен пример вызова программы CRC16-USB для расчета контрольной суммы для запроса «Device Description Request» и для пакета данных заполненного нулями.

Листинг 2.2. Пример вызова программы CRC16-USB для некоторых запросов.

```
# Zero-filled packet

$ ./crc16usb 0x0000 0x0000 0x0000 0x0000
CRC16-USB data: 0x0000
CRC16-USB data: 0x0000
CRC16-USB data: 0x0000
CRC16-USB data: 0x0000
CRC16-USB: crc = 0xf4bf, bits to send: 1 1 1 1 1 1 0 1 0 0 1 0 1 1 1 1

# Device Descriptor Request: 0x80 0x06 0x00 0x01 0x00 0x00 0x12 0x00

$ ./crc16usb 0x0680 0x0100 0x0000 0x0012
CRC16-USB data: 0x0680
CRC16-USB data: 0x0100
CRC16-USB data: 0x0000
CRC16-USB data: 0x0012
CRC16-USB: crc = 0xf4e0, bits to send: 0 0 0 0 0 1 1 1 0 0 1 0 1 1 1 1
```

3.2. Транзакции

Взаимодействие хоста и устройства по шине USB производится логическими единицами обмена которые принято называть «транзакция». Транзакция обычно состоит из «запроса» и «ответа», имеет начало, прохождение, успешное завершение или аварийное завершение. В процессе транзакции обмен данными осуществляется с заданной в начале транзакции парой ADDR:ENDP, которую принято называть «поток» («pipe»).

Инициатором любой транзакции в USB 1.0 и 1.1 всегда является хост. Начинается транзакция с отправки одного из токенов: SETUP, IN или OUT. За токеном может следовать один пакет с данными DATAx в ответ на которые приемная сторона высылает ACK — это успешное завершение транзакции. Транзакция также успешно завершается если приемная сторона отвечает на пакет с данными управляющим пакетом NAK.

Любая транзакция завершается аварийно если на шине возникла одна из следующих ситуаций:

- состояние «Idle» (шина в «J») в течении 10 мкс и более;
- состояние «Disconnect», «Reset» или «Bus Reset» (длительное «SE0»);
- состояние «SE1» в течении одного и более битовых интервалов;
- принимающая сторона высылает управляющий пакет STALL вместо пакета ACK или пакета данных.

После аварийного завершения транзакции, как правило, устройство требует полной или частичной инициализации.

В USB 1.0 и 1.1 выделяют четыре вида транзакций:

- «Control Transfer» - передача служебной информации;
- «Interrupt transfer» - передача рапорта (прерывания) от или к устройству;
- «Isochronous Transfers» - периодичная и регулярная (изохронная) передача;
- «Bulk transfer» - передача больших блоков данных.

Рассмотрим поподробнее каждый из этих четырех видов транзакций.

3.2.1. «Control Transfer» - передача служебной информации

Данный вид транзакций служит для передачи служебной информации, такой так управляющая команда, параметров настойки, структур описания элементов устройства или получение текущего статуса. Передача данных в рамках данного вида транзакций гарантирована (т. е. не подверженна потерям). «Control Transfer» активно используется для инициализации устройства при подключении. В процессе инициализации хост использует данный вид транзакции для считывания структуры описания устройства («Device Descriptor»), а также для считывания списка поддерживаемых интерфейсов («Interface Descriptor») и ассоциированных с ними конечными точками. Ниже приведена серия транзакций для получения структуры «Device Descriptor».

Транзакция 1. Хост инициирует запрос отправкой токена SETUP на адрес потока 0:0 (дефолтный «пайп»), за ним следует один пакета DATA0 содержащий 8 байт запроса «Device Descriptor Request» формат которого будет рассмотрен далее. Устройство приняв запрос подтверждает его отправкой управляющего пакета типа ACK. После отправки ACK устройство готово отправить хосту запрашиваемые данные, но это произойдет только после того, как хост пришлет токен IN. Устройство ожидает IN в течение последующих 12 мкс, иначе транзакция завершается аварийно.

Таблица 6.1. Запрос «Device Descriptor Request» от хоста к устройству.

Номер пакета и направление	**SYNC**	**PID = SETUP**	**Device address (ADDR)**	**Endpoint (ENDP)**	**CRC5**	**EOP**
№ 1 Host → Device	KJ KJ KJ KK	8'b0010_1101	7'b000_0000	4'b0000	5'b00010	SE0 SE0 J

Номер пакета и направление	**SYNC**	**PID = DATA0**	**8 байт данных (Device Descriptor Request)**	**CRC16**	**EOP**
№ 2 Host → Device	KJ KJ KJ KK	8’b1100_0011	8’h80, 8’h06, 8’h00, 8’h01, 8’h00, 8’h00, 8’h12, 8’h00	8’hE0 8’hF4	SE0 SE0 J

Номер пакета и направление	**SYNC**	**PID = ACK**	**EOP**
№ 3 Device → Host	KJ KJ KJ KK	8'b1101_0010	SE0 SE0 J

Транзакция 2. Хост готов принять данные ответа на свой запрос, он отправляет токен IN и тут же переходит на прием. Получив пакет с данными (DATA1) хост подтверждает прием отправкой ACK. Хост повторяет IN и принимает следующий пакет с данными (DATA0), подтверждает его с помощью ACK и так до тех пор, пока либо устройство не пришлет пакет с данными нулевой длины, либо хост посчитает, что он уже принял достаточное количество данных и готов завершить транзакцию (размер структуры «Device Descriptor» известен хосту заранее).

Таблица 6.2. Ответ от устройства содержащий «Device Descriptor».

Номер пакета и направление	**SYNC**	**PID = IN**	**Device address (ADDR)**	**Endpoint (ENDP)**	**CRC5**	**EOP**
№ 4 Host → Device	KJ KJ KJ KK	8'b0110_1001	7'b000_0000	4'b0000	5'b00010	SE0 SE0 J

Номер пакета и направление	**SYNC**	**PID = DATA1**	**8 байт данных (Device Descriptor Response)**	**CRC16**	**EOP**
№ 5 Device → Host	KJ KJ KJ KK	8’b0100_1011	8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX	8’hXX 8’hXX	SE0 SE0 J

Номер пакета и направление	**SYNC**	**PID = ACK**	**EOP**
№ 6 Host → Device	KJ KJ KJ KK	8'b1101_0010	SE0 SE0 J

Транзакция 3. Хост продолжает прием структуры «Device Descriptor», для этого он раз за разом посылает токен IN, ожидает пакет с данными DATAx и подтверждает его с помощью ACK.

Таблица 6.3. Продолжение приема данных структуры «Device Descriptor».

Номер пакета и направление	**SYNC**	**PID = IN**	**Device address (ADDR)**	**Endpoint (ENDP)**	**CRC5**	**EOP**
№ 7 Host → Device	KJ KJ KJ KK	8'b0110_1001	7'b000_0000	4'b0000	5'b00010	SE0 SE0 J

Номер пакета и направление	**SYNC**	**PID = DATA0**	**Еще 8 байта данных (Device Descriptor Response + 8)**	**CRC16**	**EOP**
№ 8 Device → Host	KJ KJ KJ KK	8’b1100_0011	8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX	8’hXX 8’hXX	SE0 SE0 J

Номер пакета и направление	**SYNC**	**PID = ACK**	**EOP**
№ 9 Host → Device	KJ KJ KJ KK	8'b1101_0010	SE0 SE0 J

Транзакция 4. Хост посчитал, что он принял достаточное количество данных и хочет завершить обмен, для этого он отправляет токен OUT за которым следует один **пустой пакет** с данными DATAx. В таком пакете поле с данным отсутствует, т. е. за полем PID сразу следует CRC16 и сигнал «EOP» конца пакета.

Таблица 6.4. Успешное завершение транзакции со стороны хоста пустым DATAx.

Номер пакета и направление	**SYNC**	**PID = OUT**	**Device address (ADDR)**	**Endpoint (ENDP)**	**CRC5**	**EOP**
№ 10 Host → Device	KJ KJ KJ KK	8'b1110_0001	7'b000_0000	4'b0000	5'b00010	SE0 SE0 J

Номер пакета и направление	**SYNC**	**PID = DATA1**	**CRC16**	**EOP**
№ 11 Host → Device	KJ KJ KJ KK	8’b0100_1011	8’hXX 8’hXX	SE0 SE0 J

Номер пакета и направление	**SYNC**	**PID = ACK**	**EOP**
№ 12 Device → Host	KJ KJ KJ KK	8'b1101_0010	SE0 SE0 J

В приведенном выше примере предполагается, что максимальный размер данных в пакете DATAx составляет 8 байт, что соответствует спецификации «Low Speed». Передающая сторона разбивает структуру «Device Descriptor» размером 18 байт на два пакета по 8 и 2 байта соответственно, но хост принимает только два блока по 8 байт и завершает обмен. Для «Full Speed» размер пакета должен быть 64 байта, за исключением последнего передающего остаток запрашиваемых данных.

Следует обратить внимание на то, что DATA0 и DATA1 чередуют друг друга на протяжении всего обмена который состоит из четырех различных транзакций! Обмен инициируется транзакцией с токеном SETUP, а завершается транзакцией с токеном IN и пустым пакетом данных DATAx.

3.2.2. «Interrupt Transfer» - передача рапорта (прерывания)

Многим программистам имевшим дело с микроконтроллерами или низкоуровневым программированием аппаратуры известно, что прерывания генерируются устройством в качестве сигнала центральному процессору о готовности данных или смене внутреннего состояния, что требует его (процессора) внимания. В USB 1.0 и 1.1 устройства не имеют возможности инициировать какой либо обмен по шине, вместо этого хост производит периодический опрос («poll») устройства на предмет наличия у него важного сообщения которое принято называть «рапорт» («Report»). Устройство буферизирует внутренние прерывания (и данные с ним ассоциированные) и дожидается пока хост не опросит его с помощью «Interrupt Transfer». Такие опросы могут быть не периодичными, но должны быть регулярными. Очевидно, что время реакции на прерывания зависит от того, как часто хост опрашивает устройство на наличие рапорта. «Interrupt Transfer» имеет следующие характеристики:

- гарантированную задержку;
- однонаправленную передачу данных — от устройства к хосту (IN) или от устройства к хосту (OUT);
- любая конченая точка может быть источником прерывания;
- обнаружение ошибок и ретрансмит при следующем опросе.

Различают две разновидности «Interrupt Transfer»: «Interrup Transfer IN» и «Interrupt Transfer OUT».

Передача «Interrupt Transfer IN» - от устройства к хосту. Хост периодически опрашивает конечную точку с интервалом времени заданном в структуре описания конечной точки («Endpoint Descriptor»). Опрос подразумевает отправку токена IN на заданную пару ADDR:ENDP. Если устройство успешно получило IN (т. е. без ошибок CRC), то при наличии у него ожидающего рапорта, оно высылает его данные в пакете DATAx. В свою очередь хост подтверждает прием данных управляющим пакетом ACK. Если же на стороне устройства нет готового рапорта (нет ждущих прерываний), то на входной токен IN оно отвечает управляющим пакетом NAK. Если же устройство приняло IN с ошибкой, то оно просто игнорирует его до следующего опроса. Если у данной конечной точки возникла внутренняя ошибка препятствующая её нормальному функционированию, то на входной токен IN устройство отвечает пакетом STALL, что указывает хосту на необходимость выполнить частичную или полную инициализацию. Пример такого запроса приведен ниже в таблице 7.1. Такие запросы могут использоваться для получения рапорта о состоянии HID устройства ввода (нажатие клавиш на клавиатуре, перемещение манипулятора «мышь» и т. д.)

Таблица 7.1. Передача рапорта от устройства через «Interrupt IN».

Номер пакета и направление	**SYNC**	**PID = IN**	**Device address (ADDR=1)**	**Endpoint (ENDP=0)**	**CRC5**	**EOP**
№ 1 Host → Device	KJ KJ KJ KK	8'b0110_1001	7'b000_0001	4'b0000	5'b11101	SE0 SE0 J

Номер пакета и направление	**SYNC**	**PID = DATAx**	**8 байта данных (Report)**	**CRC16**	**EOP**
№ 2 Device → Host	KJ KJ KJ KK	8’b1100_0011 или 8’b0100_1011	8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX	8’hXX 8’hXX	SE0 SE0 J

Номер пакета и направление	**SYNC**	**PID = ACK**	**EOP**
№ 3 Host → Device	KJ KJ KJ KK	8'b1101_0010	SE0 SE0 J

Передача «Interrupt Transfer OUT» - от хоста к устройству. Хост может передать срочные данных в конечную точку устройства в любой момент времени. Для этого он отправляет токен OUT следом за которым идет один пакет с данными DATAx. На что устройство отвечает одним из управляющих пакетов: ACK — данные рапорта успешно приняты и поставлены в обработку; NAK — устройство занято и не может сейчас принят данный рапорт, нужно повторить позже; и STALL — произошла ошибка и устройство требует частичной или полной инициализации. Практика показывает, что иногда для возобновления работоспособности бывает достаточным переинициализировать одну конечную точку в которой произошла ошибка. Примером такой передачи может служить отправка хостом рапорта для обновления статусных светодиодов на HID клавиатуре. В таблице 7.2 приведен формат такой транзакции в общем виде.

Таблица 7.2. Передача рапорта в устройство через «Interrupt OUT».

Номер пакета и направление	**SYNC**	**PID = OUT**	**Device address (ADDR=1)**	**Endpoint (ENDP=0)**	**CRC5**	**EOP**
№ 1 Host → Device	KJ KJ KJ KK	8'b1110_0001	7'b000_0001	4'b0000	5'b11101	SE0 SE0 J

Номер пакета и направление	**SYNC**	**PID = DATAx**	**8 байта данных (Report)**	**CRC16**	**EOP**
№ 2 Host → Device	KJ KJ KJ KK	8’b1100_0011 или 8’b0100_1011	8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX	8’hXX 8’hXX	SE0 SE0 J

Номер пакета и направление	**SYNC**	**PID = ACK**	**EOP**

№ 3 Device → Host	KJ KJ KJ KK	8'b1101_0010	SE0 SE0 J

3.2.3. «Isochronous Transfer» - постоянный периодичный обмен

Многие устройства, такие как аудио и видео кодеки, требуют непрерывного потока данных без временных задержек. Любая задержка вызванная перепосылкой пакета может оказаться критической для работы устройства и приводить к потере синхронизации потоков аудио и видео. С другой стороны, случайная и редкая потеря пакета может пройти незамеченной для пользователя. Для работы таких потоковых устройств, не требующих гарантии доставки данных, и предназначен режим изохронной передачи данных («Isochronous Transfer»), который характеризуется следующими свойствами:

- Гарантированная полоса пропускания на шине USB;
- Предсказуемая задержка;
- Однонаправленность потока;
- Обнаружение ошибок передачи с помощью CRC, но без возможности перепосылки;
- Поддерживается только для «Full Speed» и «High Speed» устройств (USB 1.1 и 2.0).

Максимальный размер блока полезных данных для данного режима определяется параметром в структуре «Endpoint Descriptor» для конечных точек поддерживающих изохронный режим (установлен соответствующий флажок). Для «Full Speed» максимальный размер блока данных не должен превышать 1023 байта, для «High Speed» - не более 1024.

Изохронные передачи также могут быть двух видов: «Isochronous Transfer IN» - от устройства к хосту, и «Isochronous Transfer OUT» - от хоста к устройству. В таблицах 7.3а и 7.3б ниже приведены примеры двух разнонаправленных изохронных передач.

Таблица 7.3а. Передача данных от устройства через «Isochronous IN».

Номер пакета и направление	**SYNC**	**PID = IN**	**Device address (ADDR=1)**	**Endpoint (ENDP=0)**	**CRC5**	**EOP**
№ 1 Host → Device	KJ KJ KJ KK	8'b0110_1001	7'b000_0001	4'b0000	5'b11101	SE0 SE0 J

Номер пакета и направление	**SYNC**	**PID = DATAx**	**0 - 1024 байта данных**	**CRC16**	**EOP**
№ 2 Device → Host	KJ KJ KJ KK	8’b1100_0011 или 8’b0100_1011	8’hXX, 8’hXX, 8’hXX, 8’hXX, ... 8’hXX, 8’hXX, 8’hXX, 8’hXX	8’hXX 8’hXX	SE0 SE0 J

Таблица 7.3б. Передача данных к устройству через «Isochronous OUT».

Номер пакета и направление	**SYNC**	**PID = OUT**	**Device address (ADDR=1)**	**Endpoint (ENDP=0)**	**CRC5**	**EOP**
№ 1 Host → Device	KJ KJ KJ KK	8'b1110_0001	7'b000_00001	4'b0000	5'b11101	SE0 SE0 J

Номер пакета и направление	**SYNC**	**PID = DATAx**	**0-1024 байта данных**	**CRC16**	**EOP**
№ 2 Host → Device	KJ KJ KJ KK	8’b1100_0011 или 8’b0100_1011	8’hXX, 8’hXX, 8’hXX, 8’hXX, ... 8’hXX, 8’hXX, 8’hXX, 8’hXX	8’hXX 8’hXX	SE0 SE0 J

Как видно, пакеты данных DATAx в данном виде транзакций не имеют подтверждающих управляющих пакетов ACK/NAK/STALL, а поток данных не может быть прерван или остановлен в рамках данного вида транзакций. Для того, чтобы выяснить состояние принимающей стороны, передающая сторона нуждается в отдельно запросе статуса (рапорта), например через режим «Interrupt Transfer».

Для синхронизации потока изохронный режим передачи полагается на периодические токены типа SOF, которые высылаются хостом каждые 1 мс. Каждый токен SOF содержит 11-ти битный номер текущего временного интервала (текущего фрейма). Устройство или хост осуществляющие передачу в изохронном режиме могут потребовать у другой стороны возобновить поток данных с какой-то временной точки в прошлом, например, если обнаружится ошибка передачи. Эта точка во времени определяется номером фрейма. Очевидно, что пересинхронизация и перепосылка возможна в пределах 2-х секунд (2048 мс).

3.2.4. «Bulk Transfer» - передача больших блоков данных

Некоторые устройства, такие как принтер или сканер, нуждаются в способе передачи больших объемов данных с минимальными «накладными расходами» и с гарантией доставки (с подтверждением). «Bulk Transfer» и есть такой способ. При использовании «Bulk Transfer» шину допускается занимать только тогда, когда она свободна от других видов передачи («Control Tansfer», «Interrupt Transfer» или «Isochronous Transfer»), то есть задержка при передачи таким способом может быть любой, а следовательно его нельзя использовать для чувствительных ко времени передачи приложений. Данный вид передачи характеризуется следующими свойствами:

- Используется для передачи больших объемов данных;
- Обнаружение ошибок передачи с помощью CRC и гарантия доставки через повторную посылку (retransmit);
- Ни пропускная полоса, ни задержка не гарантируется;
- Однонаправленность передачи;
- Поддерживается только для «Full Speed» и «High Speed» устройств (USB 1.1 и 2.0).

Размер полезного блока данных для «Bulk Transfer» определен следующим образом: для «Full Speed» - 8, 16, 32 или 64 байта; для «High Speed» - 512 байт. Если пакет не может быть заполнен данными на полную, то допускается отправка пакета DATAx с частичным заполнением, при этом выравнивание («padding» нулями) не производится. Транзакция

данного типа считается завершенной если передающая сторона передала заданное (определенное в запросе) количество байт данных, передает не полностью заполненный пакет с данными или пакет с данными нулевой длины.

На каждый успешно принятый пакет DATAx приемная сторона обязана ответить управляющим пакетом ACK. Если пакет принят с ошибкой, то принимающая сторона никак не реагирует и ждет пока передающая сторона повторно пришлет этот же пакет (на передающей стороне сработает таймаут). Принимающая сторона может ответить пакетом NAK если она не готова к приему данных и требует повторить посылку этих же данных позже. В случае ошибки на приемной стороне она индицирует своё аварийное состояние отправкой управляющего пакета STALL.

Как и для всех остальных режимов здесь тоже определены два вида: «Bulk Transfer IN» - передача от устройства к хосту, и «Bulk Transfer OUT» - для передачи данных от хоста к устройству.

«Bulk Transfer IN». Когда хост готов принять данные от устройства, он передает токен IN с указанием адреса ADDR устройства и номера конечной точки ENDP. Если устройство принимает IN с ошибкой, то просто игнорирует его. Если же IN принят на стороне устройства без ошибки, то устройство начинает посылку с первого пакета DATAx, или высылает NAK если данные не готовы, или STALL если произошла внутренняя ошибка. Хост получив пакет с данными обязан подтвердить его с помощью ACK. Чтобы получить следующий блок данных хост опять отправляет IN на ту же пару ADDR:ENDP и процесс повторяется. Отсутствие подтверждения в течении ~10 мкс приводит к тому, что устройство при следующем запросе IN вышлет повторно этот же блок данных. Если у устройства больше нет данных для отправки, то оно высылает DATAx нулевой длины для индикации окончания обмена или NAK если требуется подождать готовности новых данных. Пример данного вида обмена приведен в таблице 7.4а.

Таблица 7.4а. Передача данных к хосту через «Bulk Transfer IN».

Номер пакета и направление	**SYNC**	**PID = IN**	**Device address (ADDR=1)**	**Endpoint (ENDP=0)**	**CRC5**	**EOP**
№ 1 Host → Device	KJ KJ KJ KK	8'b0110_1001	7'b000_0001	4'b0000	5'b11101	SE0 SE0 J

Номер пакета и направление	**SYNC**	**PID = DATAx**	**8, 32, или 64 байта данных для «Full Speed»**	**CRC16**	**EOP**
№ 2 Device → Host	KJ KJ KJ KK	8’b1100_0011 или 8’b0100_1011	8’hXX, 8’hXX, 8’hXX, 8’hXX, ... 8’hXX, 8’hXX, 8’hXX, 8’hXX	8’hXX 8’hXX	SE0 SE0 J

Номер пакета и направление	**SYNC**	**PID = ACK**	**EOP**
№ 3 Host → Device	KJ KJ KJ KK	8'b1101_0010	SE0 SE0 J

«Bulk Transfer OUT». Когда хост готов отправить большой блок данных в устройство он передает токен OUT с указанием адреса устройства ADDR и номера конечной точки для ENDP для которой предназначаются данные, а следом за OUT отправляется первый пакет DATAx. Устройство отвечает ACK если данные приняты корректно или игнорирует сбойный пакет, что вызывает повторную посылку этих же данных с стороны хоста. Если устройство не готово к приему данных, то оно может ответить с помощью NAK или индицировать ошибку с помощью STALL. Хост дожидается ACK и повторяет транзакцию для следующего блока данных. Последним высылается пакет DATAx с нулевой длиной блока полезных данных, что указывает на конец обмена.

Таблица 7.4б. Передача данных к устройству через «Bulk Transfer OUT».

Номер пакета и направление	**SYNC**	**PID = OUT**	**Device address (ADDR=1)**	**Endpoint (ENDP=0)**	**CRC5**	**EOP**
№ 1 Host → Device	KJ KJ KJ KK	8'b1110_0001	7'b000_0001	4'b0000	5'b11101	SE0 SE0 J

Номер пакета и направление	**SYNC**	**PID = DATAx**	**8,32,64 байта данных для «Full Speed»**	**CRC16**	**EOP**
№ 2 Host → Device	KJ KJ KJ KK	8’b1100_0011 или 8’b0100_1011	8’hXX, 8’hXX, 8’hXX, 8’hXX, ... 8’hXX, 8’hXX, 8’hXX, 8’hXX	8’hXX 8’hXX	SE0 SE0 J

Номер пакета и направление	**SYNC**	**PID = ACK**	**EOP**
№ 3 Device → Host	KJ KJ KJ KK	8'b1101_0010	SE0 SE0 J

С технической точки зрения «Bulk Transfer» мало чем отличается от «Interrupt Transfer». Основное отличие состоит в том, что циклические транзакции «Bulk Transfer» передают разные данные смещаясь по большому блоку, в то время как «Interrupt Transfer» каждый раз передает один и тот же блок данных (рапорт).

3.2.5. Процедура завершения обмена

В спецификации USB четко сказано, что любой обмен должен завершаться отсылкой пакета DATAx содержащий блок данных размером меньше чем параметр **wMaxPacketSize** заданный для данной конечной точки (менее 8 байт для «Low Speed»). Если данные оканчиваются ровно на границе этого значения, то обязательно отсылается «пустой» пакет DATAx. Также в спецификации указывается, что данная задача возлагается на драйвер конечного устройства. Далее, на осциллограммах снятых с анализатора, мы увидим как это происходит, а пока небольшая цитата из спецификации:

Delimiting USB data transfers with packets smaller than wMaxPacketSize

Compliant USB 2.0 and USB 1.1 drivers must transmit packets of maximum size (wMaxPacketSize) and then end the transmission with a packet of less than maximum size, or delimit the end of the transmission with a zero-length packet. The transmission isn't complete until the driver sends a packet smaller than wMaxPacketSize. If the transfer size is an exact multiple of the maximum, the driver must send a zero-length delimiting packet to explicitly terminate the transfer

The device driver is responsible for delimiting the data transmission with zero-length packets, as required by the USB specification. The system USB stack doesn't generate these packets automatically.

3.2.6. Менеджмент пропускной способности

Согласно спецификации вся работа по обеспечению правильной загрузки шины USB ложится на хост. В процессе инициализации устройства хост выясняет у конечных точек режимы их работы считывая структуры «Endpoint Descriptor», после чего настраивает их исходя из имеющихся ресурсов. В процессе работы с шиной хост контролирует нагрузку исходя и того, что нагрузка создаваемая периодическим трафиком («Isochronous Transfer» и «Interrupt Transsfer») не должна превышать 90% от всего ресурса шины для «Full Speed» и 80% для «High Speed». Оставшиеся 10% резервируются для управляющих транзакций «Control Transfer», а из того что останется после этого может быть отдано для «Bulk Transfer».

3.3. Дескрипторы USB

Каждое USB устройство предоставляет в хост информацию о себе, о производителе устройства, о поддерживаемых протоколах, количестве интерфейсов и конечных точек, их типе и способах взаимодействия. Эта информация содержится в виде иерархических структур данных называемых «дескрипторами» («Descriptors»). Наиболее часто употребляемы следующие типы дескрипторов:

- «Device Descriptor» - структура описывающая общие характеристики устройства;
- «Configuration Descriptor» - наборы конфигураций поддерживаемые устройством;
- «Interface Descriptor» - структура описывающая интерфейс и его свойства;
- «Endpoint Descriptor» - структура описывающая конечную точку ассоциированную с каким-либо интерфейсом;
- «Additional Descriptor» - произвольная структура данных не определяемая стандартом.
- «String Descriptors» - пронумерованные строки символов;

Устройство может иметь только одну структуру типа «Device Descriptor». Данная структура содержит общие сведения об устройстве: версию спецификации USB, двухбайтовые идентификаторы производителя и продукта («Vendor ID» - VID и «Product ID» - PID) которые используются операционной системой для поиска и загрузки драйвера. Также «Device Descriptor» содержит число конфигураций («Configuration Descriptors») поддерживаемых устройством. «Configuration Descriptor» содержит информацию о потребляемой устройством мощности, способе электропитания (от хоста или от внешнего источника) и некоторые другие параметры. Каждая конфигурация это древовидная структура в состав которой входя структуры типа «Interface Descriptor», содержащие в свою очередь структуры типа «Endpoint Descriptor». Ниже на рис. 6 изображена древовидная структура данных USB устройства.

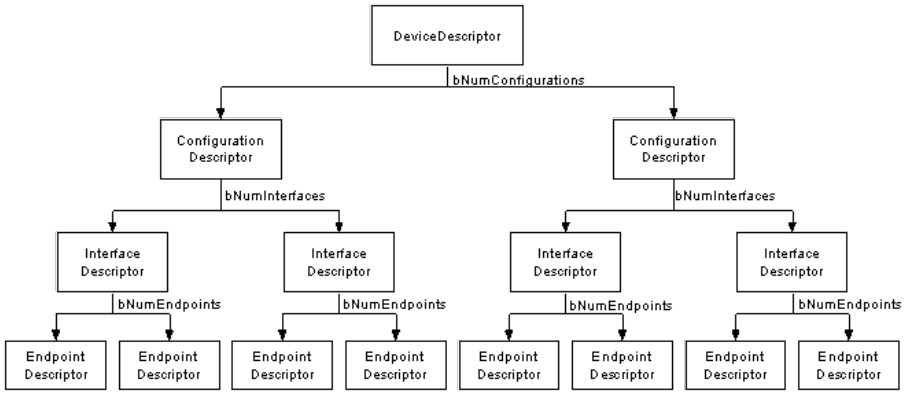

Рис. 6. Древовидная структура данных USB (дескрипторы).

В процессе инициализации хост считывает структуру типа «Device Descriptor», по ней определяет какие варианты конфигураций поддерживаются устройством и активирует одну из них. Например, устройство может поддерживать два вида конфигурации в зависимости от источника питания: «от хоста» или «от внешнего блока», при этом каждая конфигурация может потреблять различную мощность. Выше говорилось о том, что устройство может отбирать от одного порта хоста ограниченную мощность. Если этой мощности недостаточно, то устройство может переходить в специальный режим ограниченного энергопотребления и работать «в пол силы». Или же устройство может быть подключено к своему независимому источнику питания обеспечивающего полный функционал («self powered»). Хост может выбрать один из этих двух способов питания путем активации одной из конфигураций

(только одна конфигурация может быть активной). Помимо этого, разные конфигурации могут содержать различные наборы интерфейсов и конечных точек. Несмотря на такую гибкость, многие устройства поддерживают всего одну конфигурацию.

Структура «Interface Descriptor» может рассматриваться как описательный заголовок для нескольких конечных точек сгруппированных по функциональному признаку и с общими параметрами. Например, устройство офисного MFU («Multi-Functional Unit» - принтер, сканер, факс) может иметь три интерфейса: один для взаимодействия с устройством как с принтером, второй — для работы со сканером и третий для отправки факсов или электронных писем. Хост получает список интерфейсов в процессе инициализации и активирует нужные ему в данный момент времени интерфейсы. Устройство может иметь один и более активный интерфейс.

Интересной особенностью является то, что у структуры типа «Interface Descriptor» имеется два поля (**bInterfaceNumber** и **bAlternateSetting**) идентифицирующие заданный интерфейс (набор конечных точек). Первое поле определяет номер интерфейса, а второе — вариант режима передачи и некоторых специфических настроек. Например, один и тот же интерфейс с номером **bInterfaceNumber** равным **1** может иметь два режима **bAlternateSetting = 0** для передачи прерываний (Interrupt Transfer) и **bAlternateSetting = 1** для передачи больших блоков данных (Bulk Transfer). Хост может менять настройки интерфейса на ходу и переключаться между двумя вариантами одного и того же интерфейса с помощью команды **SetInterface**. Структура такой конфигурации показана на рис. 7.

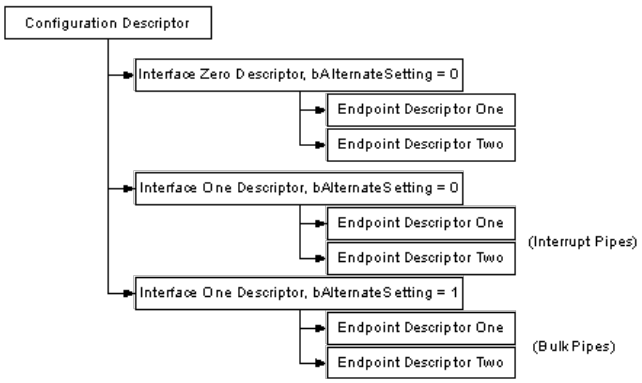

Рис. 7. Пример конфигурации интерфейсас двумя альтернативными настройками.

Структура «Endpoint Descriptor» содержит информацию о способе передачи, направлении обмена, интервале опроса и максимальному размеру пакета для обмена с данной конечной точкой. Конечная точка с номером ноль («default endpoint») всегда рассматривается как управляющая конечная точка и поэтому никогда не содержит дескриптора.

Все дескрипторы имеют общий двухбайтовый заголовок состоящий их двух параметров: **bLength** — указывающий на длину структуры в байтах, и **bDescriptionType** — собственно тип структуры. За этими двумя байтами следуют специфические для каждого их типов дескрипторов данные. Далее мы детально разберем структуру некоторых часто употребляемых дескрипторов.

3.3.1. «Device Descriptor»

Как отмечалось выше, данная структура используется для описания всего устройства в целом и содержит важную для работы устройства информацию. Каждое USB устройство имеет только одну структуру такого типа, формат которой приведен в таблице 8.1.

Таблица 8.1. Описание полей структуры параметров устройства «Device Descriptor»

Смещение	**Поле**	**Размер, байт**	**Тип или значение**	**Назначение**
0	bLength	1	Number	Длина дескриптора в байтах (**18**).
1	bDescriptorType	1	Constant	Фиксированное число (**0x01**).
2	bcdUSB	2	BCD	Номер версии спецификации USB с которой совместимо устройство. Задается в формате BCD. Пример: 0x0100 — версия USB 1.0, 0x0110 — версия USB 1.1. 0x0200 — версия USB 2.0.
4	bDeviceClass	1	Class	Класс устройства согласно утвержденному организацией USB-IF списку. Если равен **0x00**, то каждый интерфейс имеет свой собственный класс. Если равен **0xFF**, то класс определяется производителем (нестандартное устройство).
5	bDeviceSubClass	1	SubClass	Подкласс устройства.
6	bDeviceProtocol	1	Protocol	Код протокола.
7	bMaxPacketSize	1	Number	Максимальный размер пакета для «нулевой» (дефолтной) конечной точки. Возможные варианты: 8, 16, 32, 64.
8	idVendor	2	ID	Идентификатор производителя (Vendor ID), назначается организацией USB-IF.
10	idProduct	2	ID	Идентификатор продукта (Product ID), назначается производителем.
12	bcdDevice	2	BCD	Номер версии устройства в формате BCD, аналогично полю bcdUSB.
14	iManufacturer	1	Index	Индексный номер строки описывающей производителя.
15	iProduct	1	Index	Индексный номер строки описывающей продукт.
16	iSerialNumber	1	Index	Индексный номер строки содержащей

				серийный номер устройства.
17	bNumConfigurations	1	Integer	Число поддерживаемых конфигураций.

Поля **bDeviceClass**, **bDeviceSubClass** and **bDeviceProtocol** используются для поиска и назначения драйвера для обслуживания данного устройства, но не всегда. Часто устройства предпочитают идентифицировать себя через интерфейс, то есть поля **bDeviceClass**, **bDeviceSubClass** в структуре «Device Descriptor» содержат 0x00, а класс и подкласс определяются аналогичными полями в структуре «Interface Descriptor».

3.3.2. «Configuration Descriptor»

USB устройство может иметь несколько различных конфигураций, хотя большинство устройств поддерживают только одну. Конфигурация устройства описывается структурой «Configuration Descriptor» которая содержит информацию о способе электропитания устройства - «от хоста» или «от внешнего источника» («bus powered» или «mains powered», оно же «self powered») и максимальной потребляемой мощности (ток в миллиамперах). Также данный дескриптор содержит число интерфейсов поддерживаемых устройством. Описание полей структуры «Configuration Descriptor» приведено в таблице 8.2.

После того, как хост получил от устройства и проанализировал данные всех конфигурационных дескрипторов, он обязан активировать один из дескрипторов (одну из конфигураций) путем отправки команды **SetConfiguration** с указанием ненулевого значения в поле **bConfigurationValue** совпадающего с номером одного из доступных в устройстве конфигурационных дескрипторов. Выбранная таким образом конфигурация становится активной.

Таблица 8.2. Описание полей конфигурационной структуры «Configuration Descriptor»

Смещение	Поле	Размер, байт	Тип или значение	Назначение
0	bLength	1	Number	Длина дескриптора в байтах (**0x09**).
1	bDescriptorType	1	Constant	Фиксированное число (**0x02**).
2	wTotalLength	2	Number	Общее число байтов возвращаемое в ответе.
4	bNumInterfaces	1	Number	Количество интерфейсов в данной конфигурации.
5	bConfigurationValue	1	Number	Порядковый номер данной конфигурационной структуры. Используется для активации в качестве параметра для команды **SetConfiguration**.
6	iConfiguration	1	Index	Индексный номер текстовой строки описывающий данную конфигурацию.
7	bmAttributes	1	Bitmap	Бит 7 для USB 1.0 указывает на поддержку режима питания «Bus Powered». Для остальных всегда равен 1. Бит 6 указывает, что устройство может работать в режиме «Self Powered». Если

				устройство потребляет ток от хоста (шины), то в параметра **bMaxPower** указывается какой именно. Бит 5 указывает, что устройством может инициировать процедуру удаленного пробуждения хоста («Remote Wakeup»). Биты 4..0 зарезервированы и установлены в 0.
8	bMaxPower	1	mA	Максимальный потребляемый ток в 2мА единицах. Данное значение не может превышать 250 (что равно 500мА). Если устройство теряет питание от внешнего источника, то оно не должно потреблять с шины более заявленного в **bMaxPower** значения.

Особенность работы с данной структурой состоит в том, что когда хост считывает «Configuration Descriptor», устройство возвращает сразу всю ветку дерева включая соответствующие интерфейсы и конечные точки. Параметр **wTotalLength** указывает хосту на количество байт данных в возвращаемой иерархии. Определить начало вложенных структур данных можно по параметрам **bLength** и **bDescriptionType** — они всегда идут первыми и однозначно идентифицируют тип и длину соответствующей структуры данных. Пример такой иерархии приведен на рис. 8.

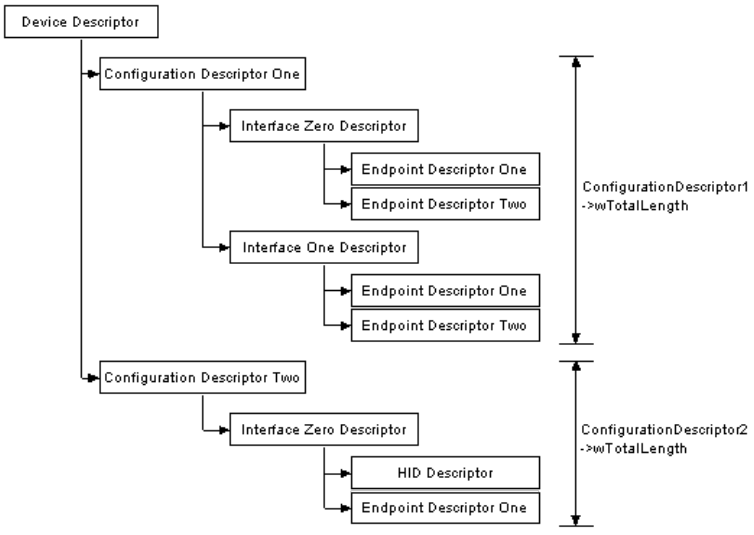

Рис. 8. Пример конфигурационной иерархии.

3.3.3. «Interface Descriptor»

Дескриптор интерфейса представляет собой групповой заголовок для списка конечных точек объединенных функционально. Формат этой структуры представлен в таблице 8.3.

Таблица 8.3. Описание полей структуры «Interface Descriptor».

Смещение	Поле	Размер, байт	Тип или значение	Назначение
0	bLength	1	Number	Длина дескриптора в байтах (**0x09**).
1	bDescriptorType	1	Constant	Фиксированное число (**0x04**).
2	bInterfaceNumber	1	Number	Порядковый номер интерфейса начиная с нуля.
3	bAlternateSetting	1	Number	Значение используемое для выбора альтернативного набора настроек (см. выше).
4	bNumEndpoints	1	Number	Количество конечных точек в данном интерфейсе за исключением нулевой.
5	bInterfaceClass	1	Class	Код класса устройства предопределяемый организацией USB-IF.
6	bInterfaceSubClass	1	SubClass	Код под-класса устройства, также определяемый организацией USB-ID.
7	bInterfaceProtocol	1	Protocol	Код протокола поддерживаемого устройством, тоже определяемый организацией USB-ID.
8	iInterface	1	Index	Индексный номер строки текста с описанием данного интерфейса.

Параметры **bInterfaceClass**, **bInterfaceSubClass** и **bInterfaceProtocol** могут быть использованы для определения драйвера для обслуживания данного интерфейса согласно классу устройства (например, «HID», «communications», «mass storage» и т. д.). Это позволяет использовать один и тот же драйвер для совершенно разных устройств если их интерфейсы поддерживают соответствующий класс и протокол.

3.3.4. «Endpoint Descriptor»

Дескриптор конечной точки описывает характеристики точки обмена, в том числе тип обмена («Transfer type»), частоту опроса от хоста и используемую максимальную пропускную способность. Хост обязан запросить эти данные перед тем как начать обмен с данной точкой. Конечная точка с номером «ноль» всегда готова к обмену, не требует предварительной настройки и используется для служебных («Control Transfer») нужд. Структура параметров этого дескриптора приведена в таблице 8.4.

Таблица 8.4. Описание полей структуры «Endpoint Descriptor».

Смещение	Поле	Размер, байт	Тип или значение	Назначение
0	bLength	1	Number	Длина дескриптора в байтах (**0x07**).
1	bDescriptorType	1	Constant	Фиксированное число (**0x05**).
2	bEndpointAddress	1	Endpoint	Адрес конечной точки — битовое поле: Биты 0..3 задают номер конечной точки. Биты 4..6 зарезервированы и установлены в «ноль». Бит 7 определяет направления передачи: 0 = OUT, 1 = IN. Для служебной контрольной точки (с номером 0), данный бит игнорируется.
3	bmAttributes	1	Bitmap	Битовое поле определяющее характеристики обмена: Биты 0..1 определяют тип обмена: 00 = «Control Transfer» 01 = «Isochronous Transfer» 10 = «Bulk Transfer» 11 = «Interrupt Transfer» Биты 2..7 зарезервированы. Биты 3..2 задают тип синхронизации («Synchronisation Type») для изохронного режима: 00 = Без синхронизации 01 = Асинхронный 10 = Адаптивный 11 = Синхронный Биты 5..4 определяют цель с которой используется данная конечная точка в изохронном режиме («Usage Type»): 00 = Data Endpoint 01 = Feedback Endpoint 10 = Explicit Feedback Data Endpoint 11 = зарезервировано.
4	wMaxPacketSize	2	Number	Максимальный размер пакета который данная конечная точка способна принимать или передавать.
6	bInterval	1	Number	Для режима «Interrupt Transfer» задает интервал времени через который данная точка должна опрашиваться для передачи данных. Для USB 1.x параметр задает время в 1 мс квантах. Для USB 2.x — в 125 мкс. Может принимать значения от 1 до 255. Для «Isochronous Transfer» должен быть установлен в 1 - минимальный интервал

				времени. Для «Bulk Transfer» и «Controls Transfer» данный параметры игнорируется.

3.3.5. «Additional Descriptor»

Дескриптор данного типа позволяет передавать нестандартные структуры данных специфичные для данного устройства или структуры данных определяемые в спецификациях, более высокого уровня. Формат этого дескриптора заранее не определен, но первые два байта так же как и у остальных дескрипторов определяют размер структуры (**bLength**) и её тип (**bDescriptorType**).

Если **bDescriptorType = 0x24**, то данный дескриптор представляет собой «Class Specific Descriptor» формат и назначение которого определяется соответствующим документом описывающим данный класс устройств. Обычно третьим байтов для «Class Specifc» идет поле **bDescriptorSubType**.

Если **bDescriptorType = 0x21**, то это дескриптор «HID» устройства. Третьим байтом в данном случае следует параметр **bcdHID** определяющий версию «HID» спецификации.

И так далее.

3.3.6. «String Descriptor»

Данный дескриптор предназначен для хранения текстовых строк символов. Сложность с этим дескриптором состоит в том, что строки текста могут быть представлены на различных языках, а сам текст строки представляется в формате **Unicode UTF-16LE** (благодарности отправляйте в компанию Microsoft). Строка с индексом «ноль» представляет собой пустую строку и используется в служебных целях, например в том случае если описание дескриптора (или иного элемента данных) отсутствует.

Перед тем как запросить у устройства строку по её индексному номеру, сначала необходимо запросить список поддерживаемых языков. Делается это запросом структуры «String Descriptor» для строки **с индексом «ноль»**. Формат структуры в данном случает представляет собой два параметры **bLength** и **bDescriptorType**, за которым следует массив двухбайтовых элементов «LANGID» идентификаторов языка. Значения этих кодов по чистой случайности совпадают с идентификаторами языков «Microsoft for Windows» описанных в документе «*Developing International Software for Windows 95 and Windows NT, Nadine Kano, Microsoft Press, Redmond, Washington*». Пример возвращаемой структуры с кодами поддерживаемых языков приведен в таблице 8.5.

Таблица 8.5. Описание структуры «String Descriptor» для нулевого индекса содержит список поддерживаемых языков.

Смеще-ние	Поле	Размер, байт	Тип или значение	Назначение
0	bLength	1	Number	Длина дескриптора в байтах (**X**).
1	bDescriptorType	1	Constant	Фиксированное число (**0x03**).
2	wLANGID[0]	2	Number	Нулевой элемент с кодом языка, например:

				0x0409 English - United States
4	WLANGID[1]	2	Number	Первый элемент с кодом языка, например: 0x0c09 English - Australian
…	…	…	…	...
N	wLANGID[x]	2	Number	N-й элемент с кодом языка, например: 0x0407 German — Standard

По сложившейся традиции нулевым элементом этого массива всегда следует «**0x0409 English - United States**», а это означает что можно смело игнорировать запрос списка языков и всегда указывать в качестве кода **0x0409** в поле **wIndex** при запросе требуемой строки.

Структура данных «String Descriptor» для строки с индексом отличным от нулевого будет иметь следующий вид:

Таблица 8.6. Описание структуры «String Descriptor» для ненулевого индекса содержит текст на указанном в wIndex языке.

Смещение	Поле	Размер, байт	Тип или значение	Назначение
0	bLength	1	Number	Длина дескриптора в байтах (**X**).
1	bDescriptorType	1	Constant	Фиксированное число (**0x03**).
2	bString	N	Unicode	Текст строки в формате UTF-16LE.

3.3.7. Классы устройств

В дополнение к основной спецификации, в декабре 1997 года был введен дополнительный документ описывающий разные виды устройств сгруппированные по общим свойствам. Этот документ получил название «Universal Serial Bus Common Class Specification». Документ вводит классификацию устройств по трем параметрам: Class, SubClass и Protocol, то есть появляется еще один уровень абстракции. Операционные системы вольны использовать три этих параметра для того, чтобы найти и назначить подходящий драйвер к вновь подключенному устройству. Такой подход очень сильно упростил разработку устройств и драйверов к ним — у производителей устройств отпала необходимость в разработке собственных драйверов если его устройство соответствует спецификации для какого-то из уже существующих классов. Ниже позволю себе привести небольшую выдержку из главы «*3.2 Why Have Classes?*» данного документа объясняющую зачем нужны классы:

3.2 Why Have Classes?

Grouping devices or interfaces together in classes and then specifying the characteristics in a Class Specification allows the development of host software which can manage multiple implementations based on that class. Such host software adapts its operation to a specific device or interface using descriptive information presented by the device. A class specification serves as a framework defining the minimum operation of all devices or interfaces which identify themselves as members of the class.

By developing in compliance with a Class Specification, entities other than the device manufacturer are able to develop software which can interact with the device. This relieves the device manufacturer from having to develop software for every combination of host platform and operating system that potentially could support the device. It also makes it easier for a device to fit into a platform/operating system's system management schemes without requiring additional support from the manufacturer. Thus, the device can be more compatible in areas such as power and connection management.

In addition, operating system vendors desiring to support a number of USB devices need to develop only a few class-specific drivers in order to make a wide-range of USB devices available for their environment. In this way, end-users have the ability to attach the latest USB devices to their system and device manufacturers get another market for their devices without requiring the development effort and distribution problems related to the use of device-specific drivers.

Напомню, что на рубеже веков у пользователей ПК существовала серьезная проблема с поиском и установкой «правильного» драйвера для приобретенного устройства. Драйверы от производителей «железа» не отличались высоким качеством кода и стабильностью работы. Создание универсальных открытых драйверов в конечном счете решило эту проблему, но уже ближе к 2010-у году.

К спецификации «Universal Serial Bus Common Class Specification» прилагается список с перечнем классов устройств и их кратким описанием, а к каждому классу позже был разработан и выпущен отдельный документ с названием в виде «Universal Serial Bus Class Definitions for XXX», где XXX — название класса, подробно регламентирующий взаимодействие с устройством данного класса. Например, для класса «CDC» в январе 1999-го года был выпущен документ «Universal Serial Bus Class Definitions for Communication Devices» описывающий работу модемов и устройств преобразования телекоммуникационных интерфейсов. Количество классов и подклассов устройств постоянно расширяется, ниже в таблице 9 приведена подборка классов устройств действующих на 2023 год.

Таблица 9. Перечень некоторых классов и подклассов USB устройств.

Класс (Class)	**Подкласс (SubClass)**	**Протокол (Protocol)**	**Описание / назначение**
0x00	0x00	0x00	Данный класс-код используемый в «Device Descriptor» указывает на то, что классификация производится согласно данным из «Interface Descriptor».
0x01	0xXX	0xXX	Audio device

0x02	0xXX	0xXX	Communications and CDC Control
0x03	0xXX	0xXX	HID – Human Interface Device
0x05	0xXX	0xXX	Physical device class
0x06	0x01	0x01	Still Imaging
0x07	0xXX	0xXX	Printer device
0x08	0xXX	0xXX	Mass Storage
0x09	0x00	0x00	Hub - Full speed Hub
0x09	0x00	0x01	Hub - Hi-speed hub with single TT
0x09	0x00	0x02	Hi-speed hub with multiple Tts
0x0A	0xXX	0xXX	CDC-Data
0x0B	0xXX	0xXX	Smart Card
0x0D	0x00	0x00	Content Security
0x0E	0xXX	0xXX	Video
0x0F	0xXX	0xXX	Personal Healthcare
0x10	0x01	0x00	Audio/Video Device – AVControl Interface
0x10	0x02	0x00	Audio/Video Device – AVData Video Streaming Interface
0x10	0x03	0x00	Audio/Video Device – AVData Audio Streaming Interface
0x11	0x00	0x00	Billboard Device
0x12	0x00	0x00	USB Type-C Bridge Device
0x13	0x00	0x00	USB Bulk Display Protocol Device Class
0x14	0x00	0x01, 0x02	MCTP Management-controller and Managed-Device endpoints
0x14	0x01	0x01, 0x02	MCTP Host Interface endpoint
0x3C	0x00	0x00	I3C Device
0xDC	0xXX	0xXX	Diagnostic Device
0xE0	0xXX	0xXX	Wireless Controller
0xEF	0xXX	0xXX	Miscellaneous

0xFE	0x01	0x01	Device Firmware Upgrade
0xFE	0x02	0x00	IRDA Bridge device
0xFE	0x03	0x00	USB Test and Measurement Device
0xFE	0x03	0x01	USB Test and Measurement Device conforming to the USBTMC USB488
0xFF	0xXX	0xXX	Vendor specific

Очевидно, что список действующих классов давно не подчищался, а некоторые из классов устройств (например «Billboard Device») вообще никогда не выходили в свет. ;-)

3.3.8. Пример иерархической структуры дескрипторов для реального устройства

Ниже в листинге 3 приведен фрагмент вывод системной утилиты **usbconfig -vv** на ОС FreeBSD, отображающий всю иерархическую структуру данных устройства «Logitech USB Receiver» подключенного к USB порту с номером **0** и присвоенным ему адресом устройства **2**. Аналогичный результат можно получить в ОС Linux по команде **lsusb -vv**.

Из этого фрагмента видно, что самый первый блок информации представляет собой структуру типа «Device Descriptor» так как её **bDescriptorType = 0x0001**. За ним следует структура «Configuration Descriptor» с порядковым номером **0**, и таких конфигурационных структур всего одна. Из конфигурационного дескриптора следует, что устройство получает питание от хоста («bus powered») и потребляет ток не более 98мА (**bMaxPower = 0x0031**).

Внутри конфигурационной структуры присутствует два дескриптора типа «Interface Descriptor». Интерфейс с номером **0** отнесен к классу **3/1/1** что соответствует «HID keyboard», а интерфейс **1** к классу **3/1/2** - «HID pointing device» (то есть «мышь»). Оба интерфейса содержат по одной конечной точке типа «Interrupt Transfer IN», каждая из которых предназначена для того, чтобы периодически получать от устройства рапорт содержащий изменение состояния устройства (либо состояние нажатых клавиш, либо перемещения указателя мыши).

Адреса конечных точек отличаются на единицу: у одной **bEndpointAddress = 0x0081** что соответствует ENDP = 1, а у другой **bEndpointAddress = 0x0082** — то есть ENDP = 2. Также каждый из интерфейсов содержит по одной структуре типа «Additional Descriptor» где **bLength = 0x9** и **bDescriptorType = 0x21**, что соответствует «HID Descriptor». Назначение полей этой структуры читателю придется загуглить самостоятельно, но скажу сразу — никакой полезной информации там не содержится. ;-)

Фактически, подключенное устройство «Logitech USB Receiver» представляет собой два стандартных логических устройства — «HID клавиатуру» и «HID мышь», каждому из которых операционной системой назначается свой HID-драйвер. Эти драйверы периодически опрашивают и получают рапорт от своей конечной точки, достают из рапорта данные о состоянии/нажатии клавиш (перемещения указателя) и формируют в системе события через устройства ввода (**/dev/input/eventXX**).

*Листинг 3. Фрагмент вывода команды **usbconfig -vv** с информацией об устройстве «Logitech USB Receiver».*

```
ugen0.2: <Logitech USB Receiver> at usbus0, cfg=0 md=HOST spd=FULL (12Mbps) pwr=ON (98mA)
ugen0.2.0: usbhid0: <Logitech USB Receiver, class 0/0, rev 2.00/29.01, addr 1>
ugen0.2.1: usbhid2: <Logitech USB Receiver, class 0/0, rev 2.00/29.01, addr 1>

  bLength = 0x0012
  bDescriptorType = 0x0001
  bcdUSB = 0x0200
  bDeviceClass = 0x0000  <Probed by interface class>
  bDeviceSubClass = 0x0000
  bDeviceProtocol = 0x0000
  bMaxPacketSize0 = 0x0008
  idVendor = 0x046d
  idProduct = 0xc534
  bcdDevice = 0x2901
  iManufacturer = 0x0001  <Logitech>
  iProduct = 0x0002  <USB Receiver>
  iSerialNumber = 0x0000  <no string>
  bNumConfigurations = 0x0001


 Configuration index 0

    bLength = 0x0009
    bDescriptorType = 0x0002
    wTotalLength = 0x003b
    bNumInterfaces = 0x0002
    bConfigurationValue = 0x0001
    iConfiguration = 0x0004  <RQR29.01_B0016>
    bmAttributes = 0x00a0
    bMaxPower = 0x0031

    Interface 0
      bLength = 0x0009
      bDescriptorType = 0x0004
      bInterfaceNumber = 0x0000
      bAlternateSetting = 0x0000
      bNumEndpoints = 0x0001
      bInterfaceClass = 0x0003  <HID device>
      bInterfaceSubClass = 0x0001
      bInterfaceProtocol = 0x0001
      iInterface = 0x0000  <no string>

      Additional Descriptor

      bLength = 0x09
      bDescriptorType = 0x21
      bDescriptorSubType = 0x11
       RAW dump:
       0x00 | 0x09, 0x21, 0x11, 0x01, 0x00, 0x01, 0x22, 0x3b,
       0x08 | 0x00

     Endpoint 0
        bLength = 0x0007
        bDescriptorType = 0x0005
        bEndpointAddress = 0x0081  <IN>
        bmAttributes = 0x0003  <INTERRUPT>
        wMaxPacketSize = 0x0008
        bInterval = 0x0008
        bRefresh = 0x0000
        bSynchAddress = 0x0000


    Interface 1
      bLength = 0x0009
      bDescriptorType = 0x0004
      bInterfaceNumber = 0x0001
      bAlternateSetting = 0x0000
      bNumEndpoints = 0x0001
      bInterfaceClass = 0x0003  <HID device>
      bInterfaceSubClass = 0x0001
      bInterfaceProtocol = 0x0002
      iInterface = 0x0000  <no string>

      Additional Descriptor
```

```
 bLength = 0x09
 bDescriptorType = 0x21
 bDescriptorSubType = 0x11
  RAW dump:
  0x00 | 0x09, 0x21, 0x11, 0x01, 0x00, 0x01, 0x22, 0xb1,
  0x08 | 0x00

Endpoint 0
   bLength = 0x0007
   bDescriptorType = 0x0005
   bEndpointAddress = 0x0082  <IN>
   bmAttributes = 0x0003  <INTERRUPT>
   wMaxPacketSize = 0x0014
   bInterval = 0x0002
   bRefresh = 0x0000
   bSynchAddress = 0x0000
```

3.4. Стандартные запросы к устройству по шине USB

Согласно спецификации хост может в любой момент отправить устройству управляющий пакет с токеном SETUP на конечную точку ENDP с номером **0** («default pipe»), при этом в поле адреса ADDR может содержаться либо номер устройства ранее присвоенный ему хостом, либо число **0** если устройству еще не присваивалось адреса или же предварительно был выполнен полный сброс («Bus reset»). Следом за токеном SETUP должен следовать пакет с данными DATA**0** длиной ровно 8 байт. Такая последовательность пакетов называется USB запрос («USB request»). На каждый запрос, если он добрался до устройства без ошибок, устройство обязано сформировать и выслать USB ответ («USB response»). Общий формат такой транзакции детально описан в главе «*3.2.1. «Control Transfer» - передача служебной информации*», а пример запроса приведен в таблице 6.1. USB запросы главным образом служат для обнаружения устройств, выполнения их настройки и изменения некоторых параметров в процессе работы.

На обработку запроса и формирование ответа устройству отводится ограниченное количество времени. Спецификацией предписываются следующие временные ограничения:

- Пакет DATA0 следующий за токеном SETUP быть подтвержден служебным пакетом ACK в течении **18-ти битовых интервалов** (что для «Low Speed» составляет **12 мкс)** после сигнала признака конца пакета («EOP»). По истечению этого времени хост должен повторить попытку отправить запрос или завершить транзакцию аварийно.
- Для транзакции запроса рапорта и установки статуса максимальное время исполнения составляет 50 мс.
- Для транзакций установки нового адреса устройства (SET_ADDRESS) максимально время ответа также составляет 50 мс, но у устройства после этого будет 2 мс на смену адреса.
- Если ответ на запрос предполагает передачу блока данных в ответе, то максимальное время на ответ составляет 500 мс.
- Любая транзакция (с самым длинным блоком данных в ответе) обязана закончиться в течении 5 сек.
- Аналогичные таймауты действуют и в обратном направлении — если передачей данных занимается устройство, то хост обязан ответить в соответствующий интервал времени.

Если хотя бы одно из эти условий нарушается со стороны устройства, то устройство признается неработоспособным и дальнейшее поведение шины USB зависит от драйвера. Драйвер, например, может выполнить «Bus reset» и повторить попытку выполнить запрос, а может просто перейти в режим «сна» предварительно оповестив систему о возникшей ошибке. В любом случае дальнейшее взаимодействие по шине возможно только после выполнения «Bus reset».

Если нарушения происходят со стороны хоста (хост не вовремя отправил ACK, например, или не прислал IN), то устройство может перейти в аварийное состояние и дальнейшее взаимодействие с ним возможно также после выполнения «Bus reset».

Важным моментом в этом деле является то, что таймауты очень короткие и если для отладки драйвера используется низкоскоростной канал выдачи отладочной информации (используется функция *printf()* для вывода в UART), то задержка на вывод скорее всего превысит установленные интервалы времени, что будет приводить к аварийному завершению транзакции, «зависанию» устройства и неожиданным результатам в процессе отладки!

Как было сказано выше, запрос всегда содержит 8 байт данных в пакете DATA**0** следующим за токеном SETUP. Ниже в таблице 10 приведен формат пакета «USB request» состоящим из пяти полей.

Таблица 10. Формат пакета «USB request».

Смещение	**Поле**	**Размер, байт**	**Тип или значение**	**Назначение**
0	bmRequestType	1	Bitmap	Бит 7 — Направление передачи данных: 0 = Host to Device 1 = Device to Host Биты 6..5 — Тип запроса: 0 = Standard 1 = Class 2 = Vendor 3 = Reserved Биты 4..0 — указывают на сущность, которая является получателем запроса: 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	bRequest	1	Value	Код запроса.
2	wValue	2	Value	Значение устанавливаемое данным запросом. Наример, для SetAddress здесь будет новый адрес устройства.
4	wIndex	2	Index/ Offset	Если ответ на запрос предполагает передачу в хост блока данных, то в этом поле указывается с какой позиции (смещение) будут выбираться данные для ответа.
6	wLength	2	Count	Количество байт данных в ответе ожидаемых хостом.

Запрос идентифицируется устройством по его коду содержащемуся в поле **bRequest**. Для того, чтобы передать с запросом расширенные параметры, могут использоваться поля **wValue** и **wIndex**.

Запросы могу быть нескольких типов: стандартные («Standard Request») - обязательные для реализации любым USB совместимым устройством, классовые («Class Request») - зависящие от класса устройства (или интерфейса) и определенные в соответствующей спецификации класса, и вендоро-зависимые («Vendor-specific Request»).

В спецификации USB 1.0 для каждой из сущностей «Device», «Interface» или «Endpoint» определено некоторое количество стандартных запросов. Рассмотрим каждый из этих видов запросов подробнее.

3.4.1. «Standard Device Requests»

Спецификацией USB 1.x предусматривается всего восемь стандартных запросов к устройству. В таблице 10.1 приведены значения полей для каждого из этих запросов.

Таблица 10.1 Значения полей при формирования «Standard Device Request».

bmRequestType	**bRequest**	**wValue**	**wIndex**	**wLength**	**Возвращае-мые данные**
1000 0000b	GET_STATUS (0x00)	0	0	2	«Device Status»
0000 0000b	CLEAR_FEATURE (0x01)	Feature Selector	0	0	Нет
0000 0000b	SET_FEATURE (0x03)	Feature Selector	0	0	Нет
0000 0000b	SET_ADDRESS (0x05)	Device Address	0	0	Нет
1000 0000b	GET_DESCRIPTOR (0x06)	Descriptor Type & Index	0 или LANGID	Размер дескриптора	Запрашиваемый дескриптор
0000 0000b	SET_DESCRIPTOR (0x07)	Descriptor Type & Index	0 или LANGID	Размер дескриптора	Запрашиваемый дескриптор
1000 0000b	GET_CONFIGURATION (0x08)	0	0	1	Конфигурацион-ный дескриптор
0000 0000b	SET_CONFIGURATION (0x09)	Configuration Value	0	0	Нет

В ответ на запрос GET_STATUS устройство возвратит блок данных DATAx размером два байта в которых значение имеют только младшие два бита: **бит 0** - указывает на то, что устройство имеет внешнее питание («Self Powered»), а **бит 1** — говорит о том, что устройство может пробуждать хост (выполнять процедуру «Remote Wakeup»). **Бит 1** может быть изменен запросом SET_FEATURE или CLEAR_FEATURE с указанием номер фичи DEVICE_REMOTE_WAKEUP (0x01).

Запросы SET_FEATURE и CLEAR_FEATURE используются для изменения значения одно-битовой настройки (фичи). Для запроса типа «Standard Device Request» предусматривается только две фичи: DEVICE_REMOTE_WAKEUP и TEST_MODE.

Запрос SET_ADDRESS используется в процессе инициализации устройства для назначения ему уникального номера (ADDR) который может принимать значения от 1 до 127. Адрес меняется только по завершению транзакции и на изменение адреса устройству дается 2 мс.

Запросы GET_DESCRIPTOR и SET_DESCRIPTOR возвращают содержимое соответствующей структуры (дескриптора). При запросе дескриптора его тип задается в старших 8-ми битах поля **wValue**, а порядковый номер дескриптора задается в младших битах этого же поля. Таким образом поле **wValue** может принимать следующие значения:

0x01XX — «Device Descriptor»;
0x02XX — «Configuration Descriptor»;
0x03XX — «String Descriptor»;
0x04XX — «Interface Descriptor»;
0x05XX — «Endpoint Descriptor».

где XX — номер запрашиваемого дескриптора. В поле **wIndex** указывается либо ноль, либо идентификатор языка в формате UTF-16LE. Если запрашивается конфигурационный дескриптор, то может быть возвращена вся иерархия данных в одном ответе: «Device Descriptor», «Configuration Descriptor» и всё множество входящих в него «Interface Descriptors» и «Endpoint Descriptors». Для этого в поле **wLength** необходимо указать число возвращаемых байтов в ответе составляющее сумму длин всех всех дескрипторов (можно указать число 255).

Важно! Запроса GET_DESCRIPTOR поддерживается только для «Device Descriptor», «Configuration Descriptor» и «String Descriptor». Дескрипторы типа «Interface Descriptor» и «Endpoint Descirptor» возвращаются в составе «Configuration Descriptor».

Запросы GET_CONFIGURATION и SET_CONFIGURATION возвращают или устанавливают номер текущей конфигурации, для этого используется поле **wValue**. SET_CONFIGURATION также применяется для активации устройства. Возвращаемый пакет данных на запрос GET_CONFIGURATION содержит только один байт. Если этот байт равен нулю, то устройство не было активировано. Иначе возвращается номер текущей (активной) конфигурации.

3.4.2. «Standard Interface Requests»

Спецификацией USB 1.x определяется пять запросов типа «Standard Interface Request» (запросов к интерфейсу), поле **wIndex** длиной два байта содержит номер интерфейса в младшем байте, а старший байт всегда равен нулю. В таблице 10.2 приведены значения полей для этих запросов.

Таблица 10.2 Значения полей при формирования «Standard Interface Request».

bmRequestType	**bRequest**	**wValue**	**wIndex**	**wLength**	**Возвращаемые данные**
1000 0001b	GET_STATUS (0x00)	0	Interface	2	Два байта статуса - всегда нули!
0000 0001b	CLEAR_FEATURE (0x01)	Feature Selector	Interface	0	Нет
0000 0001b	SET_FEATURE (0x03)	Feature Selector	Interface	0	Нет
1000 0001b	GET_INTERFACE (0x0A)	0	Interface	1	Альтернативный интрефейс.
0000 0000b	SET_INTERFACE (0x0B)	Alt. Setting	Interface	0	Нет

В ответ на запрос GET_STATUS всегда возвращает пакет DATAx содержащий два нуля (0x00, 0x00).

Запросы SET_FEATURE и CLEAR_FEATURE используются для изменения значения одно-битовой настройки (фичи). В спецификация USB 1.x и 2.0 не определено ни одной из таких настроек.

Запросы SET_INTERFACE и GET_INTERFACE используются для установки альтернативной функции на данном интерфейсе (см. описание «Interface Descriptor» выше).

3.4.3. «Standard Endpoint Requests»

Спецификацией USB 1.x определяется четыре запроса типа «Standard Endpoint Request» (запросов к конечным точкам). Аналогичным образом поле **wIndex** длиной два байта содержит номер конечной точки в младших четырех битах, **Бит 7** этого же поля указывает направление запроса - «к конечной точке» или «от неё» (OUT или IN), остальные биты равны нулю. В таблице 10.3 приведены значения полей для четырех доступных запросов к конечным точкам.

Таблица 10.3 Значения полей при формирования «Standard Interface Request».

bmRequestType	bRequest	wValue	wIndex	wLength	Возвращаемые данные
1000 0010b	GET_STATUS (0x00)	0	Endpoint	2	Два байта: бит 0 — Halt.
0000 0010b	CLEAR_FEATURE (0x01)	Feature Selector	Endpoint	0	Нет
0000 0010b	SET_FEATURE (0x03)	Feature Selector	Endpoint	0	Нет
1000 0010b	SYNCH_FRAME (0x0C)	0	Endpoint	Two	Номер кадра

В ответ на запрос GET_STATUS всегда возвращает пакет DATAx содержащий два байта, в которых значащим является только один: **Бит 0 — Halt**. Если бит Halt установлен в «1», то данная конечная точка находится в остановленном режиме. Изменить Halt («запустить» или «остановить» поток данных) можно запросами типа CLEAR_FEATURE или SET_FEATURE.

Запросы SET_FEATURE и CLEAR_FEATURE используются для изменения значения одно-битовой настройки (фичи). Для конечных точек доступна только одна настройка — ENDPOINT_HALT (0x00), позволяющая хосту временно приостанавливать поток данных. Данная фича определена только для конечных точек отличных от нулевой («default pipe»).

Запрос SYNCH_FRAME используется для синхронизации кадров данных («frame») и применяется в изохронном режиме (см описание «Isochronouse Transfer»). Предполагается, что в таком режиме данные передаются кадрами с последовательной нумерацией. Хост может попросить устройство повторить (или начать) передачу данных с определенного кадра, например, если на стороне хоста была зафиксирована потеря данных.

3.5. Несколько слов о хабах (USB Hub)

Спецификацией USB 1.0 шина USB представляется как древовидная сеть устройств строгой подчиненности. В узлах сети могут располагаться как конечные устройство («device»), так и устройство разветвления шины — «USB Hub». Хабы с одной стороны подключаются к хосту и ведут себя как устройство определенного класса (Hub Class), с другой — выступают в качестве «хоста» для одного и более портов через которые могут быть подключены как конечные устройства, так и другие хабы. Всего в сети может располагаться не более $\mathbf{2^7 = 127}$ устройств (7 бит отведено на номер устройства в поле ADDR) включая сами хабы, а значит глубина такого дерева ограничена **7-ю уровнями** (сбалансированное дерево). Порт хаба используемый для связи с хостом называется «root port», а порт для связи с подчиненным устройством - «downstream port». В промежуточных хабах порт для связи с хабом «вверх» называется «upstream port».

С точки зрения хоста, любой хаб это такое же устройство как и остальные, но этому устройству присвоен класс Hub Class (0x09) и оно реализует набор команд позволяющий

управлять своими «downstream» портами, так, как это делает обычный хост — то есть выполнять сброс шины на порту, детектировать наличие подключения/отключения устройств, отправлять и получать данные через него, проводить инициализацию шины подключенной к порту (выполнять «Device Enumeration»). Структура хаба приведена на рис. 9.1.

В составе USB хаба имеется контроллер (Hub Controller) который работает по следующему принципу. Каждый хаб в сети функционирует как простейшее ретранслирующее устройство. В отличии о того же Etherenet коммутатора, USB хаб не запоминает номера (адреса) устройств назначаемые хостом. Когда хост отправляет пакет, он поступает на «upstream» порт хаба. Обнаружив преамбулу SYNC (или «Start of Packet») на своём «upstream» порту, хаб производит коммутацию своего «upstream» порта со всеми активными (не заблокированными) «downstream» портами и ретранслирует принимаемый «сверху» пакет «вниз» во все порты до окончания пакета (EOP). Это позволяет хосту отправить пакет к любому из устройств в сети. Устройство обязано отвечать только на пакеты адресованные ему или на адрес 0. Когда хаб детектирует наличие преамбулы SYNC на одном из своих «donwstream» портов, то он производит коммутацию этого порта с «upstream» портом и ретранслирует принимаемый «снизу» пакет «вверх» по дереву, блокируя при этом остальные порты. Это в свою очередь позволяет хосту получать ответы от адресуемого устройства. Хаб удерживает коммутацию до тех пор, пока не получит признак конца пакета (EOP) или не произойдет таймаут на длительность передачи одного пакета. На рис. 9.2. приведены три варианта коммутации портов внутри USB хаба.

У USB хаба есть и другие режимы работы и задачи, такие как например управление питанием или сбросом. Поскольку разработка USB хаба нашей задачей не ставится, то я не буду вдаваться в эти подробности и надеюсь, что общий принцип функционирования USB хаба понятен. Основной момент тут состоит в том, что USB хаб с точки зрения хоста это такое же USB устройство как и все остальные.

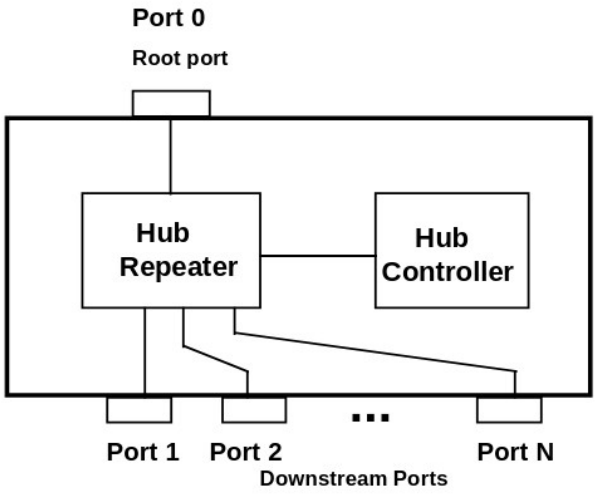

Рис. 9.1. Структура типового USB Hub-а.

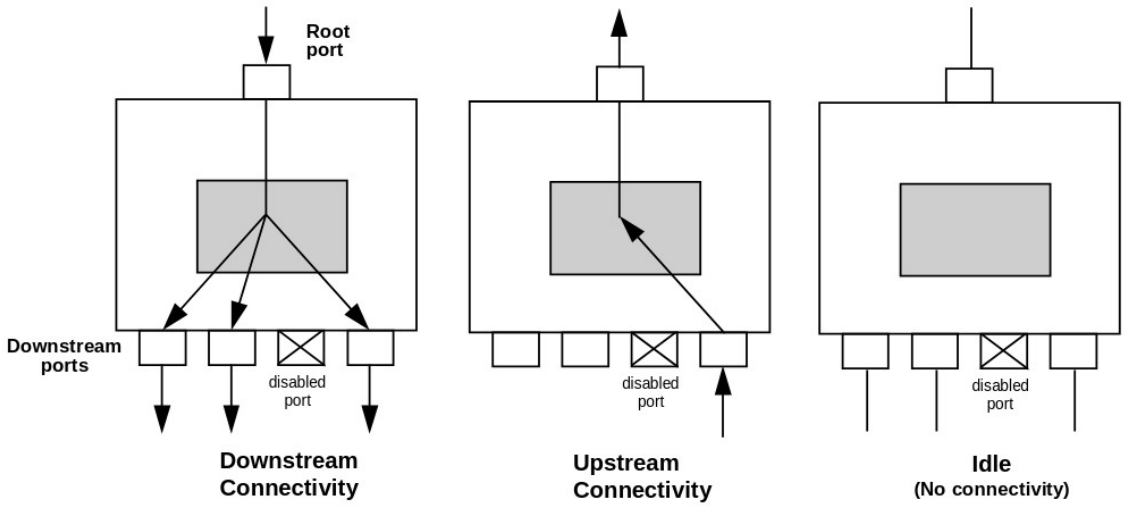

Рис. 9.2. Варианты коммутации портов внутри USB хаба.

3.6. Процедура инициализация USB устройства

Теперь когда у нас есть понимание того, как передаются данне по шине USB, какие форматы данных в ней используются и как формируются запросы, мы можем поговорить о том, как происходит процесс инициализации вновь подключенного устройства к шине. В спецификации этот процесс называется «Device Enumeration and Configuration» (перечисление и конфигурирование устройств), одной из стадией этого процесса является выяснение числа устройств на шине и присвоение каждому из них отдельного уникального номера (адреса). Сразу отмечу, что процедура инициализации весьма запутана, зависит от класса подключенного устройства и наличия цепочки хабов. Для полноты картины добавлю, что полная инициализация шины с одним устройством на ней содержит порядка 60-ти транзакций, каждая из которых содержит два или три пакета, не учитывая возникающие перепосылки. Инициализация повторяется рекурсивно для всех портов каждого из хабов и может составлять несколько тысяч транзакций (см. «*Appendix B*» в документе «*AN57294 USB 101: An Introduction to Universal Serial Bus 2.0*» от компании CYPRESS). Мы же ограничимся минималистичной процедурой инициализации без участия хабов.

Процедура инициализации всегда начинается с выполнения сброса шины (или порта хаба) сразу после обнаружения на ней устройства. После сброса взаимодействие с подключенным устройством начинается с обмена через нулевую конечную точку (через «default pipe», ADDR:ENDP = 0:0). Когда USB устройство подключается к шине или определяет сигнал сброса, оно присваивает себе нулевой адрес, а значит пакеты отправляемые хостом на **0:0** будут достигать этого устройства. Одной из первостепенных задач со стороны хоста при выполнении процедуры инициализации вновь подключенного устройства является выяснение максимального размера пакета для нулевой конечной точки, для этого запрашиваются первые 8 байт структуры «Device Descriptor» и из ответа извлекается поле **bMaxPacketSize0** — оно показывает предельный размер блока данных отправляемого в нулевую конечную точку на устройстве. Напомню, что для «Low Speed» устройств максимальный размер блока полезных данных в пакете DATAx не может превышать 8 байт, а для устройств «Full Speed» и «High Speed» размер блока данных уже существенно больше — 1023 и 1024 байт соответственно. Отсылка большого блока данных

может создавать проблемы простым устройствам не имеющим такого объема оперативной памяти для входного буфера и этот момент требуется принимать во внимание.

Затем устройству присваивается уникальный номер который в дальнейшем используется как адрес. Когда устройству присвоен номер отличный от нуля, то оно принимает, обрабатывает и отвечает только на пакеты/запросы адресованные этому номеру и более не реагирует на пакеты адресованные нулевому устройству до очередного сброса шины. Это значит, что после присвоения номера, хост продолжает процедуру инициализации устройства отправляя пакеты уже по конкретному адресу. После присвоения номера, производится полное считывания дескриптора устройства, конфигурационных структур, активация требуемой конфигурации, настройка интерфейсов и конечных точек.

Согласно выше сказанному упрощенная процедура инициализации может выглядеть следующим образом:

1. Выполнить сброс шины (отправить по шине «Bus reset»).
2. Запросить структуру «Device Descriptor» отправив «Device Descriptor Request» на адрес **0:0**, из ответа получить только первый блок данных размером 8 байт и выяснить поддерживаемую устройством версию USB, а также значение **bMaxPacketSize0**.
3. Сделать повторный сброс шины («Bus reset»). Это, как оказалось, не обязательное требование предписывается спецификация USB 1.x для того, чтобы «разглючить» устройства ранних реализаций которые были склонны зависать.
4. Повторно запросить структуру «Device Descriptor» отправив «Device Descriptor Request» на адрес **0:0**, полностью принять и сохранить всю структуру.
5. Присвоить и установить уникальный номер устройству (отправить «Device SET_ADDRESS Request» на адрес **0:0**).
6. Запросить дерево конфигурации (отправить «Configuration Request» на адрес **X:0**, где **X** — вновь присвоенный адрес), получить и сохранить структуру с деревом конфигурации.
7. Выбрать подходящую конфигурацию и активировать её (отправить «Device SET_CONFIGURATION Request» на адрес **X:0**).

Данная процедура далеко не полная. Обычно после описанных выше действий выполняют конфигурирование интерфейсов и конечных точек. Однако, её вполне достаточно для многих простых устройств, таких как устройства класса HID.

Ниже в таблице 11 почастям представлена последовательность транзакций между хостом и устройством в процессе инициализации. Данная процедура упрощена, в ней отсутствует двойной запрос структуры «Device Descriptor» для выяснения параметра **bMaxPacketSize0 -** это валидно только для шины работающей в режиме «Low Speed» где размер блока данных не может превышать 8 байт.

Таблица 11. Процедура инициализации USB (упрощенный вариант для «Low Speed»).

Транзакция 1. Сброс и отправка запроса «Device GET_DESCRIPTOR Request». Поле **bRequest = 0x06**, поле w**Length = 0x12**. Запрашиваем сразу всю структуру размером 18 байт.

Номер пакета и направление	**SYNC**	**PID**	**Device address (ADDR)**	**Endpoint (ENDP)**	**CRC5**	**EOP**
№ 1 Host → Bus	SE0 длительностью не менее 20мс (Reset)					J

№ 2 Host → Device	KJ KJ KJ KK	8'b0010_1101 (SETUP)	7'b000_0000 (ADDR=0)	4'b0000 (ENDP=0)	5'b00010	SE0 SE0 J
№ 3 Host → Device	KJ KJ KJ KK	8’b1100_0011 (DATA0)	8’h80, 8’h06, 8’h00, 8’h01, 8’h00, 8’h00, 8’h12, 8’h00 (Device Descriptor Request)		8’hE0 8’hF4	SE0 SE0 J
№ 4 Device → Host	KJ KJ KJ KK	8'b1101_0010 (ACK)				SE0 SE0 J

Транзакция 2. Получение первого блока данных (8 байт) ответа содержащего «Device Descriptor Response». Всего в ответе ожидаем 18 байт.

№ 5 Host → Device	KJ KJ KJ KK	8'b0110_1001 (IN)	7'b000_0000 (ADDR=0)	4'b0000 (ENDP=0)	5'b00010	SE0 SE0 J
№ 6 Device → Host	KJ KJ KJ KK	8’b0100_1011 (DATA1)	8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX (Device Descriptor Response)		8’hXX 8’hXX	SE0 SE0 J
№ 7 Host → Device	KJ KJ KJ KK	8'b1101_0010 (ACK)				SE0 SE0 J

Транзакция 3. Получение следующего блока данных (8 байт) ответа содержащего «Device Descriptor Response».

№ 8 Host → Device	KJ KJ KJ KK	8'b0110_1001 (IN)	7'b000_0000 (ADDR=0)	4'b0000 (ENDP=0)	5'b00010	SE0 SE0 J
№ 9 Device → Host	KJ KJ KJ KK	8’b1100_0011 (DATA0)	8’hXX, 8’hXX, 8’hXX, 8’hXX 8’hXX, 8’hXX, 8’hXX, 8’hXX (Device Descriptor Response + 8)		8’hXX 8’hXX	SE0 SE0 J
№ 10 Host → Device	KJ KJ KJ KK	8'b1101_0010 (ACK)				SE0 SE0 J

Транзакция 4. Получение последнего блока данных (2 байта) ответа содержащего «Device Descriptor Response».

№ 11 Host → Device	KJ KJ KJ KK	8'b0110_1001 (IN)	7'b000_0000 (ADDR=0)	4'b0000 (ENDP=0)	5'b00010	SE0 SE0 J
№ 12	KJ KJ KJ KK	8’b0100_1011	8’hXX, 8’hXX		8’hXX	SE0

Device → Host		(DATA1)	(Device Descriptor Response + 16)	8’hXX	SE0 J
№ 13 Host → Device	KJ KJ KJ KK	8'b1101_0010 (ACK)			SE0 SE0 J

Транзакция 5. Завершение обмена для запроса «Device Descriptor Request». Хост получил все 18 байт запрашиваемой структуры и отправляет токен OUT с «пустым» пакетом DATAx, что является признаком конца обмена.

Номер пакета и направление	**SYNC**	**PID**	**Device address (ADDR)**	**Endpoint (ENDP)**	**CRC5**	**EOP**
№ 14 Host →Bus	SE0 длительностью не менее 20мс (Reset)					J
№ 15 Host → Device	KJ KJ KJ KK	8'b1110_0001 (OUT)	7'b000_0000 (ADDR=0)	4'b0000 (ENDP=0)	5'b00010	SE0 SE0 J
№ 16 Host → Device	KJ KJ KJ KK	8’b0100_1011 (DATA1)	(данные отсутствуют)		8’h00 8’h00	SE0 SE0 J
№ 17 Device → Host	KJ KJ KJ KK	8'b1101_0010 (ACK)				SE0 SE0 J

К этому моменту у хоста имеется структура «Device Descriptor» содержащая общую информацию об устройстве, в том числе номер поддерживаемой версии USB, идентификаторы производителя VID и продукции PID, а также Class/SubClass устройства.

Хост должен проверить версию USB, а также выяснить имеется ли у него драйвер для данного VID/PID и Class/SubClass если последние не равны нулю. Напомню, что если эти параметры равны нулю, то класс устройства определяется в структуре «Interface Descriptor». Если версия USB не поддерживается или хост не имеет совместимого драйвера, то процедура инициализации заканчивается и шина переводится в состояние «сна».

Транзакция 6. Присвоение устройству нового номера (адреса) путем отправки запроса «Device SET_ADDRESS Request».

№ 18 Host → Device	KJ KJ KJ KK	8'b0010_1101 (SETUP)	7'b000_0000 (ADDR=0)	4'b0000 (ENDP=0)	5'b00010	SE0 SE0 J
№ 19 Host → Device	KJ KJ KJ KK	8’b1100_0011 (DATA0)	8’h00, 8’h05, **8’h01**, 8’h00, 8’h00, 8’h00, 8’h00, 8’h00 (Device SET_ADDRESS = **1** Request)		8’hEB 8’h25	SE0 SE0 J
№ 20 Device → Host	KJ KJ KJ KK	8'b1101_0010 (ACK)				SE0 SE0

			J

С этого момента устройство будет отвечать только на запросы отправленные на ADDR=1. Дальнейшая инициализация продолжается с использованием нового адреса. Хост обычно ведет учет использованных адресов путем монотонного увеличения номера.

Транзакция 7. Отправка запроса «Configuration Descriptor Request» на **ADDR=1**. Поле **wIndex = 0** указывает на нулевой конфигурационный профиль. Поле **wLength = 0xff** указывает на то, чтобы устройство выдало всю ветку дерева включая нулевой «Configuration Descriptor», входящие в него нулевой «Interface Descriptor» и нулевой «Endpoint Descriptor». **wLength** в данном превышает сумму длин всех трех структур. Так как запрос отправляется на новый адрес, то он начинается с пакета **DATA0**.

№ 21 Host → Device	KJ KJ KJ KK	8'b0010_1101 (SETUP)	7'b000_0001 (ADDR=**1**)	4'b0000 (ENDP=0)	5'b11101	SE0 SE0 J
№ 22 Host → Device	KJ KJ KJ KK	8’b1100_0011 (**DATA0**)	8’h80, 8’h06, **8’h00**, 8’h02, 8’h00, 8’h00, **8’hff**, 8’h00 (Configuration Descriptor Request)		8’hE9 8’hA4	SE0 SE0 J
№ 23 Device → Host	KJ KJ KJ KK	8'b1101_0010 (ACK)				SE0 SE0 J

Транзакция 8. Получение первого блока данных (8 байт) ответа содержащего «Configuration Descriptor Response». Всего в ответе ожидаем 25 байт.

№ 24 Host → Device	KJ KJ KJ KK	8'b0110_1001 (IN)	7'b000_0001 (ADDR=1)	4'b0000 (ENDP=0)	5'b11101	SE0 SE0 J
№ 25 Device → Host	KJ KJ KJ KK	8’b0100_1011 (DATA1)	8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX, 8’hXX (Configuration Descriptor Response)		8’hXX 8’hXX	SE0 SE0 J
№ 26 Host → Device	KJ KJ KJ KK	8'b1101_0010 (ACK)				SE0 SE0 J

Транзакция 9. Получение следующего блока данных (8 байт) ответа содержащего «Configuration Descriptor Response».

№ 27 Host → Device	KJ KJ KJ KK	8'b0110_1001 (IN)	7'b000_0001 (ADDR=1)	4'b0000 (ENDP=0)	5'b11101	SE0 SE0 J
№ 28	KJ KJ KJ KK	8’b1100_0011	8’hXX, 8’hXX, 8’hXX, 8’hXX		8’hXX	SE0

Device → Host		(DATA0)	8’hXX, 8’hXX, 8’hXX, 8’hXX (Configuration Descriptor Response + 8)	8’hXX	SE0 J
№ 29 Host → Device	KJ KJ KJ KK	8'b1101_0010 (ACK)			SE0 SE0 J

Транзакция 10. Получение еще одного блока данных (8 байт) ответа содержащего «Configuration Descriptor Response».

№ 30 Host → Device	KJ KJ KJ KK	8'b0110_1001 (IN)	7'b000_0001 (ADDR=1)	4'b0000 (ENDP=0)	5'b11101	SE0 SE0 J
№ 31 Device → Host	KJ KJ KJ KK	8’b0100_1011 (DATA1)	8’hXX, 8’hXX, 8’hXX, 8’hXX 8’hXX, 8’hXX, 8’hXX, 8’hXX (Configuration Descriptor Response + 16)		8’hXX 8’hXX	SE0 SE0 J
№ 32 Host → Device	KJ KJ KJ KK	8'b1101_0010 (ACK)				SE0 SE0 J

Транзакция 11. Получение последнего блока данных содержащего 1 байт) ответа на запрос «Configuration Descriptor Response». Устройство высылает неполный пакет, это означает что у него больше нет данных, то есть все структуры были переданы.

№ 33 Host → Device	KJ KJ KJ KK	8'b0110_1001 (IN)	7'b000_0000 (ADDR=0)	4'b0000 (ENDP=0)	5'b11101	SE0 SE0 J
№ 34 Device → Host	KJ KJ KJ KK	8’b1100_0011 (DATA0)	8’hXX (Configuration Descriptor Response + 24)		8’hXX 8’hXX	SE0 SE0 J
№ 35 Host → Device	KJ KJ KJ KK	8'b1101_0010 (ACK)				SE0 SE0 J

Транзакция 12. Завершение обмена для запроса «Configuration Descriptor Request». Хост получил 25 байт запрашиваемой структуры и признак того, что данных для него больше нет, отправляет токен OUT с «пустым» пакетом DATAx для завершения обмена.

№ 36 Host → Device	KJ KJ KJ KK	8'b1110_0001 (OUT)	7'b000_0001 (ADDR=1)	4'b0000 (ENDP=0)	5'b11101	SE0 SE0 J

№ 37 Host → Device	KJ KJ KJ KK	8’b0100_1011 (DATA1)	(данные отсутствуют)	8’h00 8’h00	SE0 SE0 J
№ 38 Device → Host	KJ KJ KJ KK	8'b1101_0010 (ACK)			SE0 SE0 J

К этому моменту у хоста имеется четыре информационные структуры описывающие устройство: «Device Descriptor» - содержащая общую информацию об устройстве, «Configuration Descriptor» - содержащая информацию о дефолтной конфигурации, нулевой «Interface Descriptor» в данной конфигурации, и нулевой «Endpoint Descriptor» в данном интерфейсе.

Хост может еще раз проверить Class/SubClass полученные уже из «Interface Descriptor» чтобы выбрать драйвер основываясь на классе устройства.

Транзакция 13. Активация (выбор) конфигурации путем отправки запроса «Device SET_CONFIGURATION Request». Как отмечалось выше, подавляющее большинство USB устройств имеют только одну конфигурацию. Перед активацией, хост должен выяснить её индекс из полученной структуры «Configuration Descriptor». Обычно нумерация конфигурационных структур начинается с единицы, поэтому поле **wValue = 1**. Если в **wValue** указать ноль, то это приведет к **деактивации** устройства! **wLength = 0** так как никакого ответа от устройства не ожидается.

№ 39 Host → Device	KJ KJ KJ KK	8'b0010_1101 (SETUP)	7'b000_0001 (ADDR=1)	4'b0000 (ENDP=0)	5'b11101	SE0 SE0 J
№ 40 Host → Device	KJ KJ KJ KK	8’b1100_0011 (DATA0)	8’h00, 8’h09, **8’h01**, 8’h00, 8’h00, 8’h00, 8’h00, 8’h00 (Device SET_CONFIGURATION = **1** Request)		8’h25 8’h27	SE0 SE0 J
№ 41 Device → Host	KJ KJ KJ KK	8'b1101_0010 (ACK)				SE0 SE0 J

Транзакция 14. Завершение обмена путем отправки пакета типа IN с «пустым» DATAx.

№ 42 Host → Device	KJ KJ KJ KK	8'b0110_1001 (IN)	7'b000_0001 (ADDR=1)	4'b0000 (ENDP=0)	5'b11101	SE0 SE0 J
№ 43 Host → Device	KJ KJ KJ KK	8’b0100_1011 (DATA1)	(данные отсутствуют)		8’h00 8’h00	SE0 SE0 J
№ 44 Device → Host	KJ KJ KJ KK	8'b1101_0010 (ACK)				SE0 SE0 J

С этого момента устройство готово к работе! Программное обеспечение на хосте может продолжить выяснять подробности об устройстве путем запрашивания индексных строк с описанием производителя и т. д., а драйверы более высокого уровня могут провести дополнительную настройку интерфейсов и конечных точек (например, драйвер клавиатуры может установить светодиодную индикацию в соответствующее состояние).

Определенно, приведенную выше процедуру инициализации можно сократить еще более, например не запрашивать структуру «Interface Descriptor» и содержащийся в ней «Endpoint Descriptor», но тогда мы не сможем правильно определить класс устройства и отделить «мышь» от «клавиатуры». Именно такую процедуру мы будем выполнять в нашей реализации USB контроллера и программного драйвера.

3.7. Инициализация USB устройства на экране анализатора сигналов

С целью более полного понимания картины я воспользовался анализатором сигналов Saleae Logic 8, работающим под управлением открытого ПО ***Sigrok/PulseView,*** и записал весь обмен по шине USB (в режиме «Low Speed») в процессе исполнения описанной выше процедуры инициализации USB HID устройства типа «клавиатура». Ниже представлены участки «осциллограмм» записанного обмена в том виде и в той последовательности как они реально передавались. Ко всем осциллограммам я представил свои комментарии.

Но перед этим позволю себе несколько слов про незаменимую утилиту PulseView. Данная утилита представляет собой графический интерфейс для Sigrok - подсистемы захвата и декодирования цифровых сигналов. У Sigrok также имеется и текстовый интерфейс предоставляемый утилитой **sigrok-cli**. Sigrok в связке с PulseView предоставляет удобный интерфейс захвата сигналов через поддерживаемое устройство, визуализации его и декодирования протокола. Список устройств немалый. Список поддерживаемых для декодирования протоколов тоже достаточно обширный, среди них имеется USB «Low Speed» и «Full Speed».

Для работы с PulseView необходимо установить в систему следующие пакеты с помощью утилиты **apt-get** (Linux) или **pkg** (FreeBSD):

```
libsigrok-0.5.2_5          Framework for hardware logic analyzers, core library
libsigrokdecode-0.5.3      Framework for hardware logic analyzers, protocol decoders library
sigrok-cli-0.7.2_2         Framework for hardware logic analyzers, CLI client
pulseview-0.4.2_7          GUI client that supports various hardware logic analyzers
```

После чего следует подключить анализатор сигналов к USB порту вашего ПК и убедиться в том, что анализатор детектируется операционной системой. Команда **usbconfig -v** или **lsusb -v** должна отображать устройство анализатора в списке обнаруженных. В моём случае анализатор Saleae Logic 8 определяется следующим образом:

```
ugen0.4: <vendor 0x0925 product 0x3881> at usbus0, cfg=0 md=HOST spd=HIGH (480Mbps) pwr=ON
(100mA)

  bLength = 0x0012
  bDescriptorType = 0x0001
  bcdUSB = 0x0200
  bDeviceClass = 0x00ff  <Vendor specific>
  bDeviceSubClass = 0x00ff
  bDeviceProtocol = 0x00ff
  bMaxPacketSize0 = 0x0040
```

```
idVendor = 0x0925
idProduct = 0x3881
bcdDevice = 0x0001
iManufacturer = 0x0000  <no string>
iProduct = 0x0000  <no string>
iSerialNumber = 0x0000  <no string>
bNumConfigurations = 0x0001
```

Запустить утилиту **pulseview** можно либо из командной строки, либо из меню **Applications → Development** графической оболочки. В утилите необходимо настроить каналы источники для двух анализируемых сигналов: **D0** (**черный** провод) на анализируемый сигнал USB_DM с целевого устройства, и **D1** (**коричневый** провод) на сигнал USB_DP этого же устройства. Подключить щупы сигнальных линий D0 и D1 можно параллельно к разъему USB на целевом устройстве не забыв при этом также соединить линию GND (**серый** провод). В качестве целевого устройства в моём случае выступает плата «Карно» с микросхемой ПЛИС и с загруженным битстримом содержащим уже синтезированную систему-на-кристелле в составе которой присутствует разработанный мной контроллер USB 1.0 и драйвер к нему.

После подключения источников сигнала необходимо установить частоту сэмплирования в значение на один порядок больше чем частота анализируемого сигнала. Для USB «Low Speed» я выбираю частоту сэмплирования 24 МГц (с запасом), объем выборки для записи - 50M сэмплов вполне достаточно. Запись сигналов начинается нажатием кнопки «Run». Чтобы включить декодирование, необходим кликнуть на значок «Add protocol decoder» и выбрать в списке «USB Request» указав на каких линиях находятся сигналы **D-** (USB_DM) и **D+** (USB_DP).

Итак, после включения записи вставляем штекер от USB клавиатуры в разъем на целевом устройстве, дожидаемся окончания записи и начинаем анализировать записанный сигнал. Общий вид всей процедуры инициализации устройства (USB HID Keyboard Device), которая заняла около 80 мс, представлен на осциллограмме рис. 10.1. В ней видно два «сброса» (розовые «ромбы») после каждого из которых, спустя паузу в 14 мс, имеется обмен (серые «овалы»), при этом обмен после второго сброса существенно более длинный по числу пересланных пакетов и затраченного времени, это видно по размеру «овала».

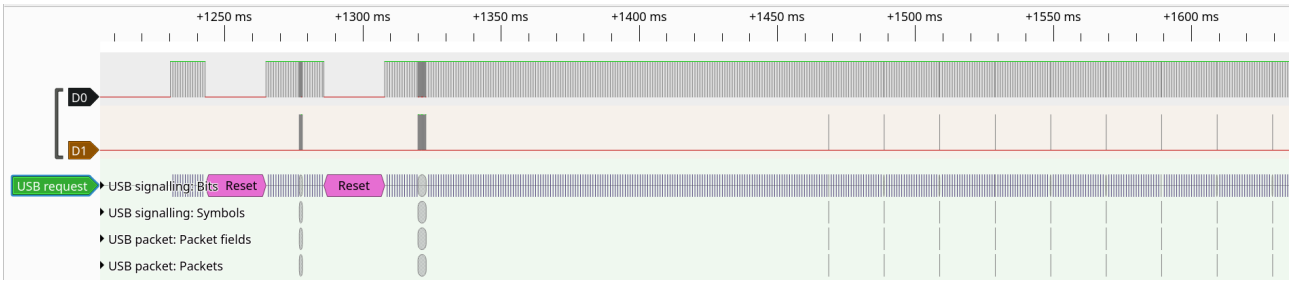

Рис. 10.1. Общий вид процедуры инициализации USB устройства.

На осциллограмме видно, что спустя примерно 150 мс после инициализации, начинается регулярный, с интервалом в 20 мс, обмен с устройством. Также на этой осциллограмме виден «частокол» из сигналов «Keepalive» следующих с интервалом в 1 мс, что характерно для USB «Low Speed».

3.7.1. Первый сброс и запрос структуры «Device Descriptor»

Приблизимся и посмотрим как выглядит обмен после первого сброса шины. Напомню, что в нашей минималистичной процедуре после первого сброса выполняется

полное считывание структуры «Device Descriptor». Если увеличить первый серый овал, то мы увидим пакет типа SETUP (рис. 10.2а) отправленный хостом на нулевую конечную точку нулевого устройства. Следом за ним идет пакет типа DATA0 (рис. 10.2б) содержащий запрос вида «Device Descriptor Request». В теле запроса 6-й байт равен 0x12 — это поле **wLength** задающее размер ожидаемого ответа в 18 байт, что соответствует размеру структуры «Device Descriptor».

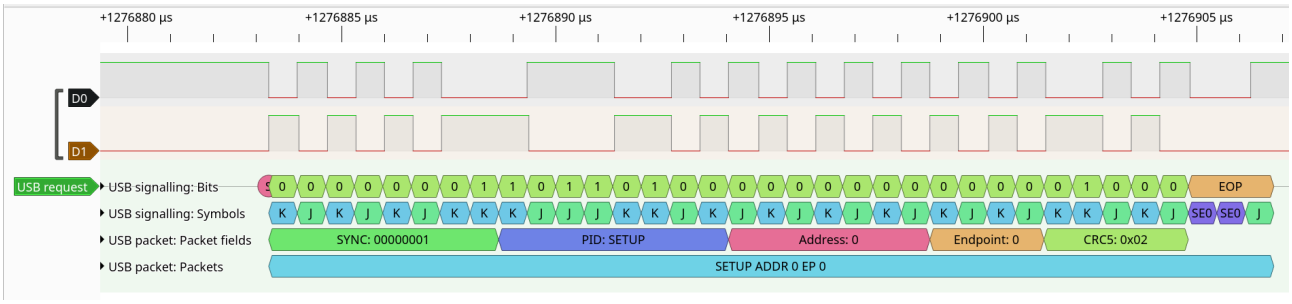

Рис. 10.2а. Пакет SETUP адресованный на 0:0.

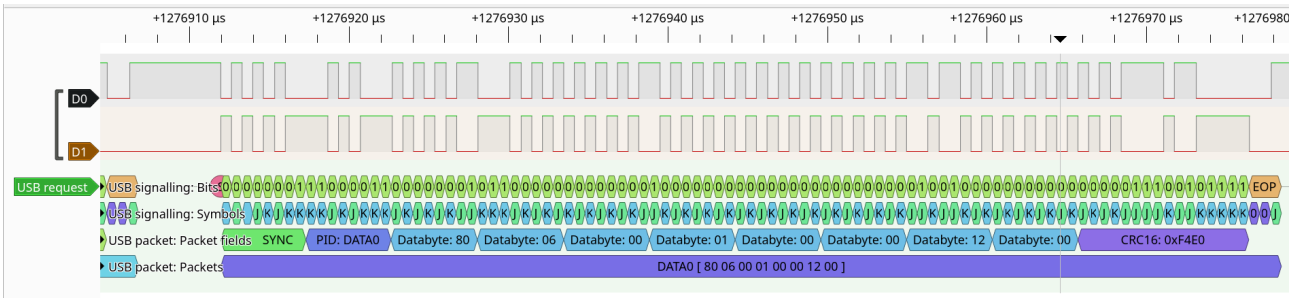

Рис. 10.2б. Пакет DATA0 содержащий запрос «Device Descriptor Request».

Следом за DATA0 мы видим пакет типа ACK (рис. 10.2в) — это подтверждение от устройства сигнализирующее о том, что пакет с запросом принят без ошибок. Для наглядности, на рис. 10.2г представлены все три этих же пакета на одной осциллограмме.

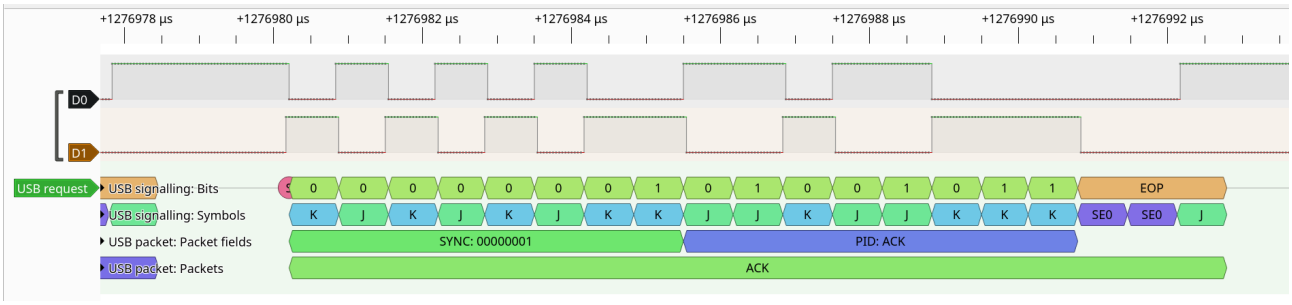

Рис. 10.2в. Пакет ACK от устройства подтверждающий успешный прием данных.

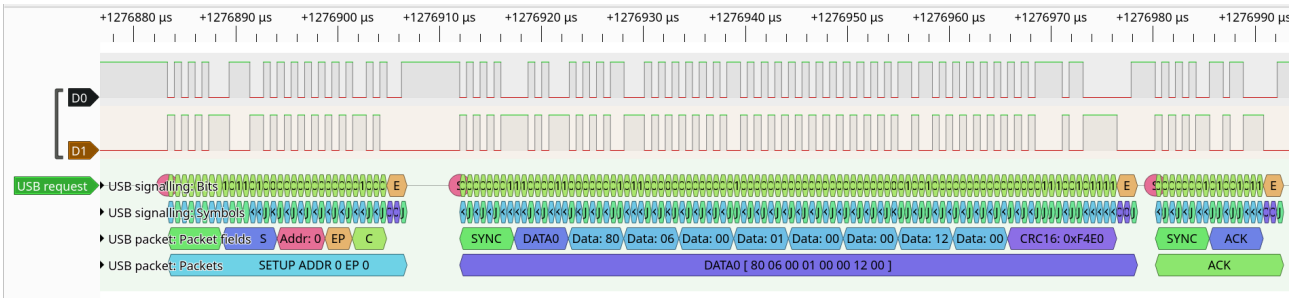

Рис. 10.2г. Пакеты SETUP, DATA0 и ACK на одной осциллограмме.

Далее, хост производит небольшую задержку и начинает опрашивать устройство на предмет наличия у того данных составляющих ответ на запрос. Для этого хост посылает пакеты с токеном IN (рис. 10.3а) на всё ту же пару ADDR:END = 0:0. На этой же осциллограмме мы видим, что хост получает от устройства отрицательный ответ готовности данных - токен NAK. Хост повторяет опрос некоторое время, пока не получит пакет с данными. На рис. 10.3б видно, что хосту пришлось пять раза отравить IN перед тем, как устройство выдало пакет с данными DATA1. Всё это время потребовалось устройству чтобы подготовить ответ на присланный ранее запрос.

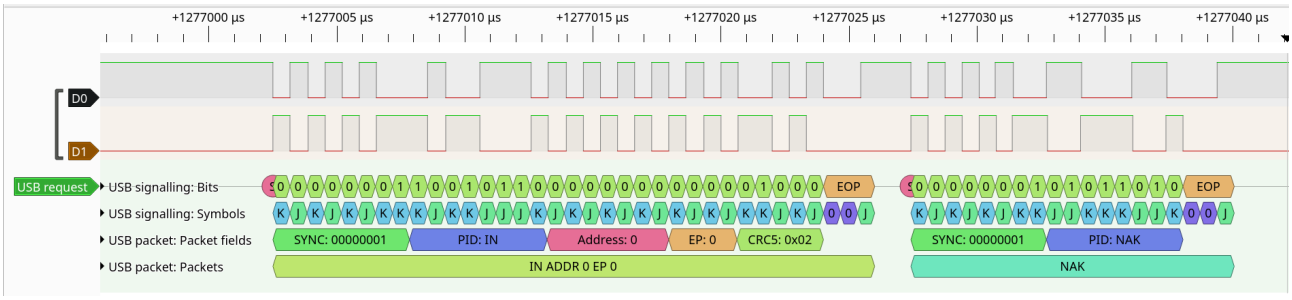

Рис. 10.3а. Хост опрашивает устройство на предмет наличие ответа отсылая токен IN. Устройство отрицает готовность данных токеном NAK.

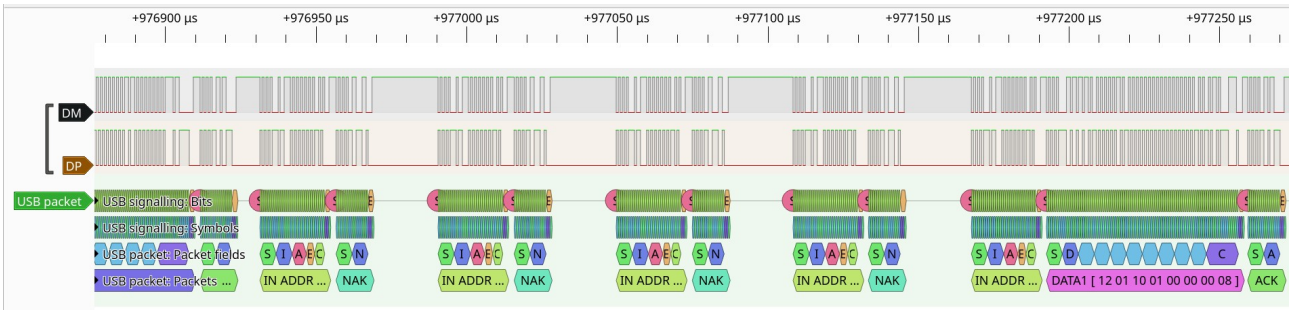

Рис. 10.3б. Хост отправляет пять раз IN перед тем, как получит пакет с данным DATA1.

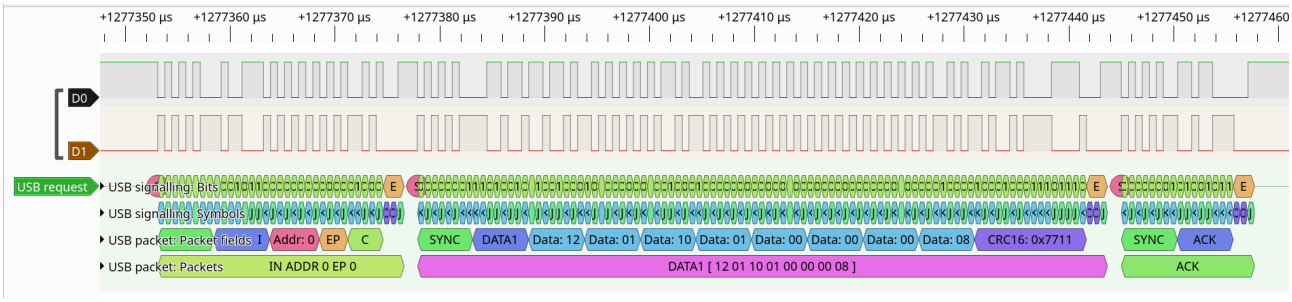

Рис. 10.3в. Хост отправляет токен IN, получает пакет с данным DATA1 и подтверждает его приём токеном ACK.

На рис. 10.3в приведена увеличенная осциллограмма токена IN и последовавшего за ним пакета с данными DATA1.

Получив первые 8 байт ответа хост продолжает опрашивать устройство отправкой еще одного токена IN в надежде получить следующий блок данных ответа. В этот раз устройство высылает пакет с данными DATA0 незамедлительно (рис. 10.3г). Хост также подтверждает успешность приёма токеном ACK.

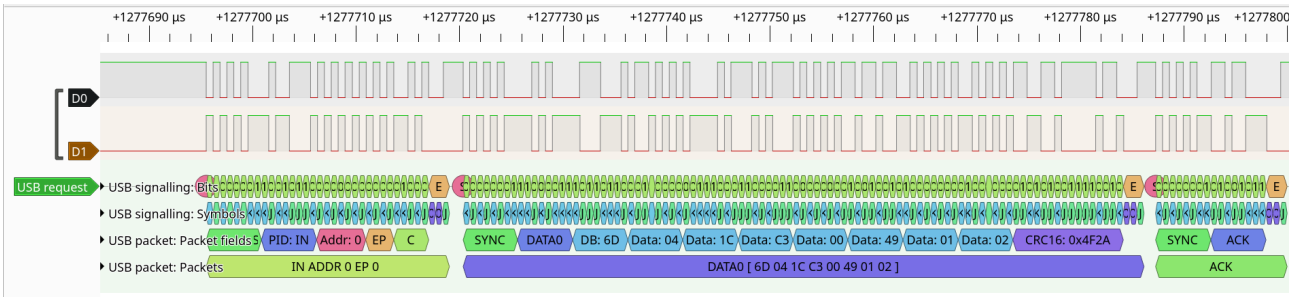

Рис. 10.3г. Хост повторяет токен IN, незамедлительно получает пакет с данным DATA0 и подтверждает его приём токеном ACK.

Хост продолжает опрашивать устройство отсылкой токена IN до тех пор, пока не получит требуемое количество байт данных (в данном случае 18 байт). На рис. 10.3д представлена осциллограмма еще одной транзакции осуществляемой хостом для получения данных от устройства. В данном случае устройство возвращает пакет DATA1 с двумя байтами полезной нагрузки — это всё, что осталось у устройства от сформированного им ответа на запрос «Device Descriptor Request».

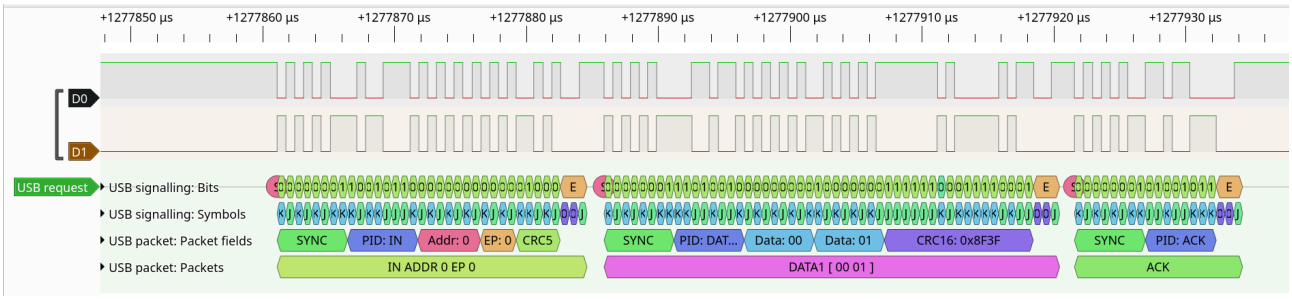

Рис. 10.3д. Хост в очередной раз отправляет токен IN, на что устройство выдает последний блок данных размером 2 байта в пакете DATA1.

Возврат неполного или пустого пакета DATAx сигнализирует хосту от том, что у устройства больше нет данных для пересылки. Хост, посчитав что он принял необходимое количество байт данных, завершает обмен по данному «пайпу» отправкой транзакции OUT с пустым DATAx. Эта транзакция представлена на рис. 10.3е. Устройство обязательно подтверждает успешность приема данных токеном ACK. Если подтверждение не придет в течение заданного времени, то хост будет вынужден перепослать токен OUT и пакет DATA1 еще раз (таймаут).

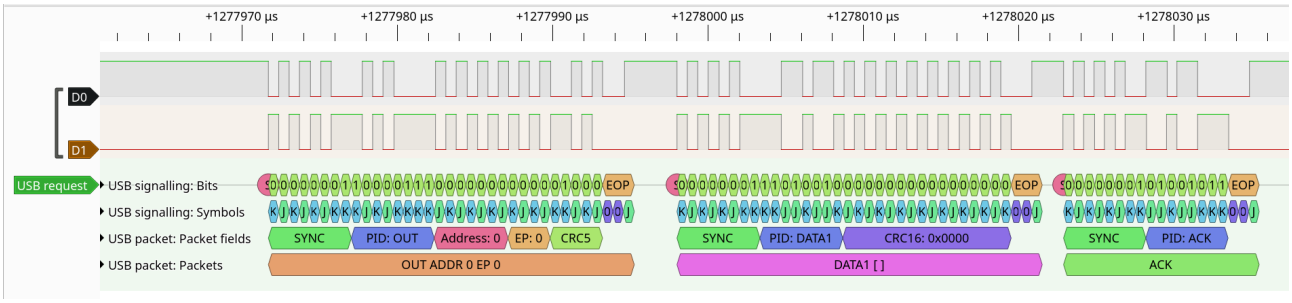

Рис. 10.3е. Хост отправляет токен OUT и следом пустой пакет DATA1 сигнализируя о конце обмена. Устройство подтверждает прием токеном ACK. Конец обмена!

К этому моменту хост полностью получил запрашиваемую им структуру «Device Descriptor». Мы можем попытаться декодировать принятые хостом данные и выяснить VID/PID устройства и его класс. Принятый поток данных складывается из трех пакетов:

первый пакет: 0x12, 0x01, 0x10, 0x01, 0x00, 0x00, 0x00, 0x08

второй пакет: 0x6D, 0x04, 0x1C, 0xC3, 0x00, 0x49, 0x01, 0x02

третий пакет: 0x00, 0x01.

Наложим эти данные на структуру описание которой дано в таблице 8.1. «*Описание полей структуры параметров устройства «Device Descriptor»*»:

```
typedef struct {
        uint8_t bLength;              // 1 Length of this descriptor = 18 bytes
        uint8_t bDescriptorType;      // 1 Descriptor type = DEVICE (01h)
        uint16_t bcdUSB;              // 2 USB specification version (BCD)
        uint8_t bDeviceClass;         // 1 Device class
        uint8_t bDeviceSubClass;      // 1 Device subclass
        uint8_t bDeviceProtocol;      // 1 Device Protocol
        uint8_t bMaxPacketSize0;      // 1 Max Packet size for endpoint 0
        uint16_t idVendor;            // 2 Vendor ID (or VID, assigned by USB-IF)
        uint16_t idProduct;           // 2 Product ID (or PID, assigned by the manufacturer)
        uint16_t bcdDevice;           // 2 Device release number (BCD)
        uint8_t iManufacturer;        // 1 Index of manufacturer string
        uint8_t iProduct;             // 1 Index of product string
        uint8_t iSerialNumber;        // 1 Index of serial number string
        uint8_t bNumConfigurations;   // 1 Number of configurations supported
} USB10_Description;
```

Результат декодиования:

```
bLength = 0x12        – 18 байт;
bDescriptorType = 0x01– тип структуры: «Device»;
bcdUSB = 0x0110 – USB версии 1.1;
bDeviceClass = 0x00   – класс устройства будет определен в структуре «Interface Descriptor»;
bDeviceSubClass = 0x00;
bDeviceProtocol = 0x00;
bMaxPacketSize0 = 0x08– максимальный размер пакета для пайта 0:0 составляет 8 байт;
idVendor = 0x046D     – соответствует «Logitech Inc.»;
idProduct = 0xC31C    – соответствует «USB Keyboard»;
bcdDevice = 0x4900    – ревизия (версия) устройства;
iManufacturer = 0x01  – индексный номер строки с описанием производителя;
iProduct = 0x02       – индексный номер строки с описанием продукта;
iSerialNumber = 0x00  – строка с серийным номером отсутствует;
bNumConfigurations = 0x01 – число поддерживаемых конфигураций.
```

3.7.2. Второй сброс и запрос присвоения адреса «Device SET_ADDRESS Request»

Продолжая анализировать записанный с шины USB сигнал, мы увидим второй сигнал сброса такой же продолжительностью 14 мс. После этого сигнала сброса и паузы в 12 мс мы увидим еще один токен SETUP так же отправленный на адрес 0:0. Следом за ним хост передает пакет DATA0 с новым запросом - «Device SET_ADDRESS Request» в котором поле **wValue** установлено в **1** (см. рис. 10.4а). Этим запросом хост пытается назначить устройству новый адрес — **1**. Устройство подтверждает успешный прием данного пакета с запросом.

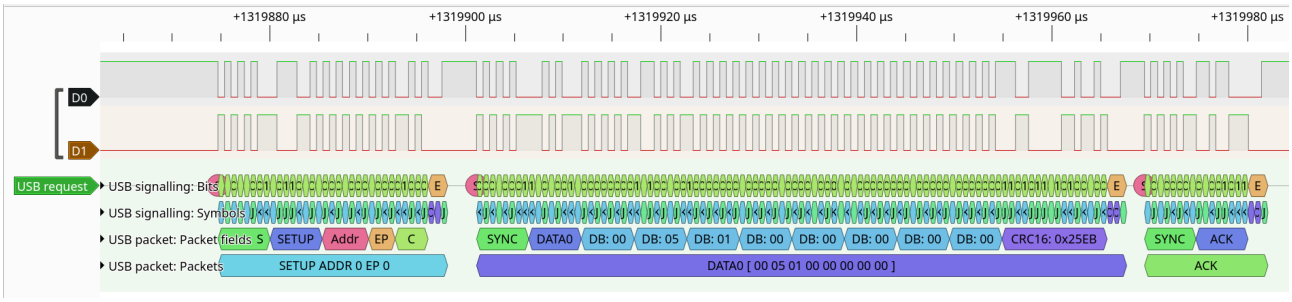

Рис. 10.4а. Хост отправляет токен SETUP и пакет с данными запроса «Device SET_ADDRESS Request». Устройство подтверждает успешность приема токеном ACK.

Теперь хост должен выяснить, был ли запрос обработан успешно или же возникла какая-то внутренняя ошибка на устройстве. Для этого он отправляет токен IN (см. рис. 10.4а) сигнализируя устройству о том, что хост желает получить от него блок данных. Устройство незамедлительно высылает пустой пакет DATA1. Это означает, что 1) у устройства нет данных для хоста или возврат данных не предусматривается запросом, и 2) устройство функционирует нормально. Если бы в процессе обработки запроса возникла ошибка, то устройство ответило бы токеном STALL и хосту потребовалось бы начинать процедуру инициализации с самого начала. Хост получив пустой пакет с данными подтверждает его прием отправкой токена ACK и на этом завершает обмен в рамках данного запроса.

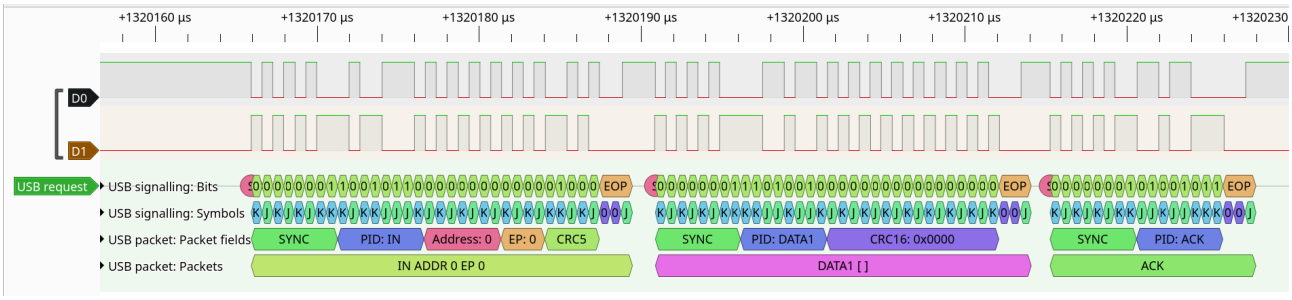

Рис. 10.4б. Хост отправляет токен IN чтобы узнать состояние устройства. Устройство отвечает пустым пакетом DATA1 — признак того, что всё в порядке, запрос обработан.

Тем временем устройство производит смену адреса и всё дальнейшее взаимодействие с ним должно будет производиться по новому адресу ADDR:ENDP = **1**:0.

3.7.3. Запрос конфигурационной структуры «Configuration Descriptor Request»

После присвоения нового адреса, далее по плану минималистичной процедуры инициализации, хост должен запросить дефолтную структуру с конфигурацией. Напомню, что вместе с конфигурационным дескриптором устройство может выдать подчиненную часть дерева, в том числе «Interface Descriptor» и «Endpoint Descriptor» входящий в состав интерфейсного дескриптора. Хост может воспользоваться этой возможностью чтобы сократить число запросов. Для этого в запросе «Configuration Descriptor Request» в поле **wLength** необходимо указать размер (число байт) равный или превышающий сумму размеров всех этих подчиненных структур вместе с размером структуры «Configuration Descriptor». Если число структур и их размер заранее неизвестен, то хост может указать какое-то произвольное большое число, например **0xff**. Грязный хак, скажет читатель, но он работает! Более правильное решение состоит в том, чтобы запросить сначала первые 8 байт структуры

«Configuration Descriptor», добыть из них значение поля **wTotalLength**, оно указывает на максимальное число байт для всех структур, а потом еще раз перезапросить конфигурационную структуру указав это значение в поле **wLength**

Если в запросе установить поле **wIndex = 0**, то устройство в ответе выдаст нулевой конфигурационный профиль и всё что к нему относится. Пример такого запроса приведен на рис. 10.5а. Запрос, как мы видим, отправляется на новый адрес ADDR:ENDP = 1:0. В подавляющем большинстве устройств присутствует только один конфигурационный профиль, поэтому данного запроса достаточно чтобы получить исчерпывающую информацию об устройстве.

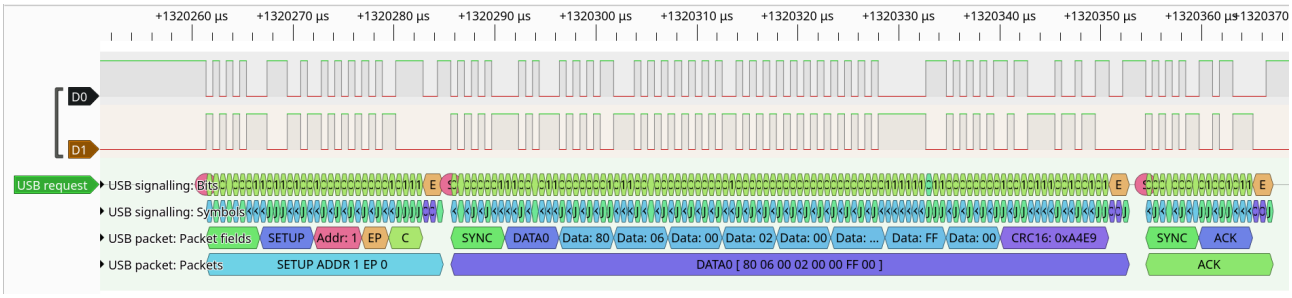

Рис. 10.5а. Хост отправляет токен SETUP и пакет DATA0 содержащий запрос «Configuration Descriptor Request». Устройство подтверждает прием запроса.

Чтобы получить ответ, хост опрашивает устройство отсылкой токена IN. Обычно устройству требуется некоторое время чтобы подготовить ответ, и если ответ еще не готов, то устройство отвечает токеном NAK (что означает «приходите позже»). Получив NAK хост продолжает посылать токены IN и в какой-то момент устройство отвечает пакетом с данными содержащим ответ на запрос. Первый такой пакет DATA1 приведен на рис. 10.5б.

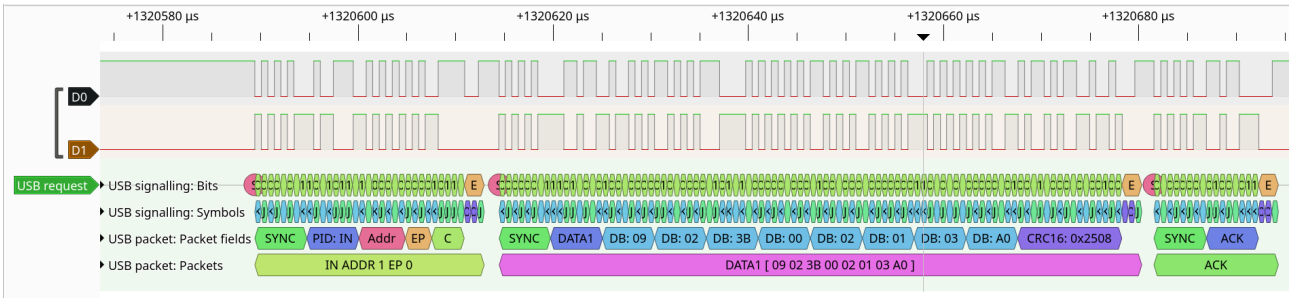

Рис. 10.5б. Хост отправляет токен IN, на что устройство отвечает пакетом DATA1 содержащий часть ответ на запрос «Configuration Descriptor Request». Хост подтверждает прием токеном ACK.

Хост продолжает методично опрашивать устройство посылая токен IN и раз за разом получает порцию данных ответа на запрос в следующих пакетах DATA0, DATA1,…, DATA0. На рис. 10.5в ниже приведены осциллограммы всей серии пакетов при получении хостом ответа на запрос конфигурационного дескриптора.

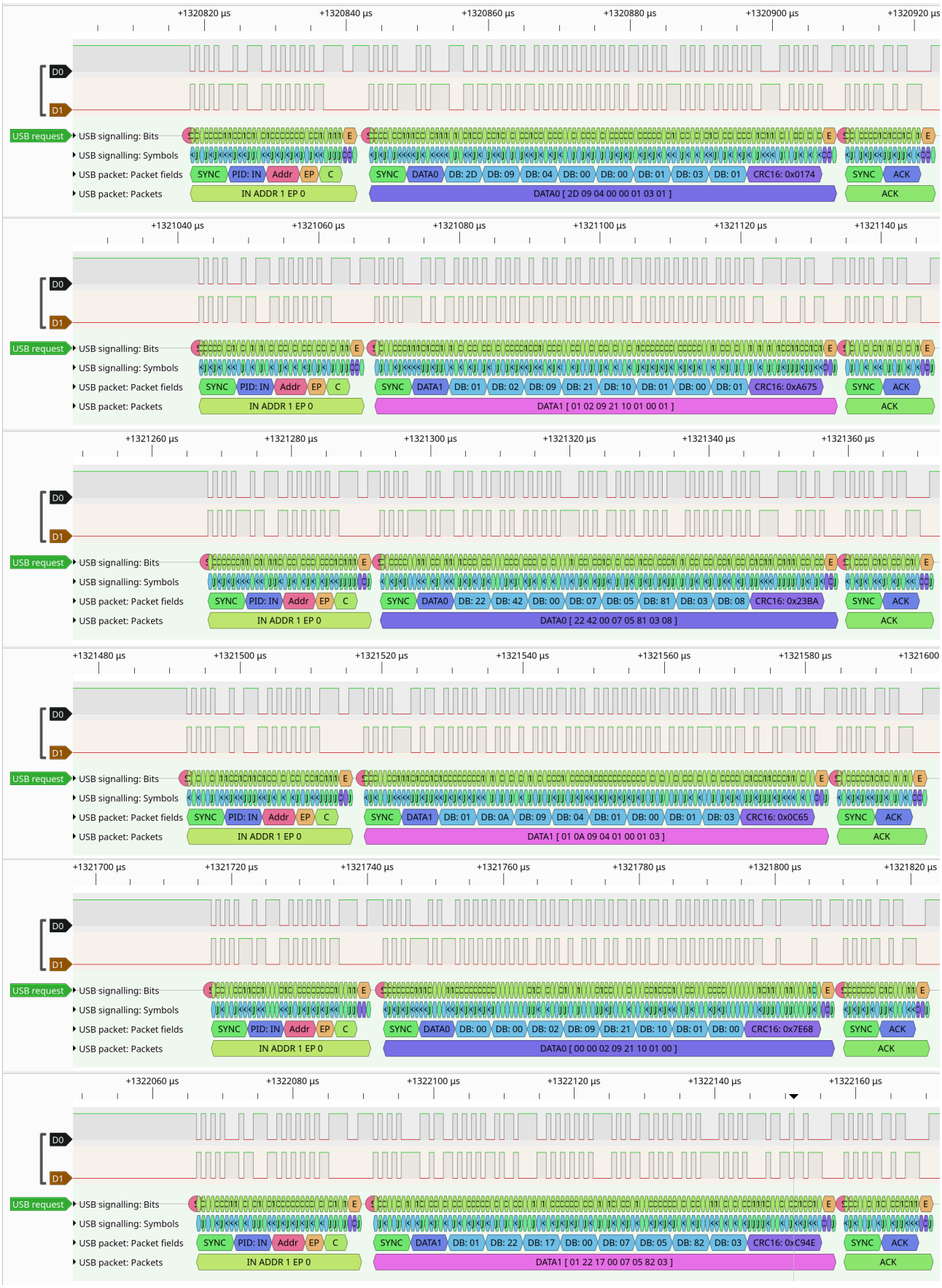

Рис. 10.5в. Хост многократно отправляет токены IN и получает от устройства следующую порцию данных в пакетах DATAx подтверждая прием отсылкой токена ACK.

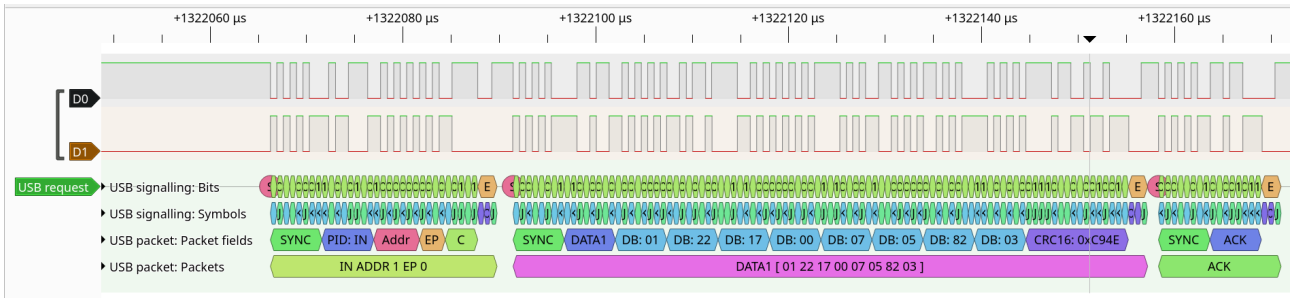

Рис. 10.5в. (продолжение)

В какой-то момент получив в очередной раз токен IN устройство отвечает неполным пакетом DATA0 (см. рис. 10.5г). Это означает, что у устройства более нет данных для отправки хосту, то есть весь ответ на запрос был передан.

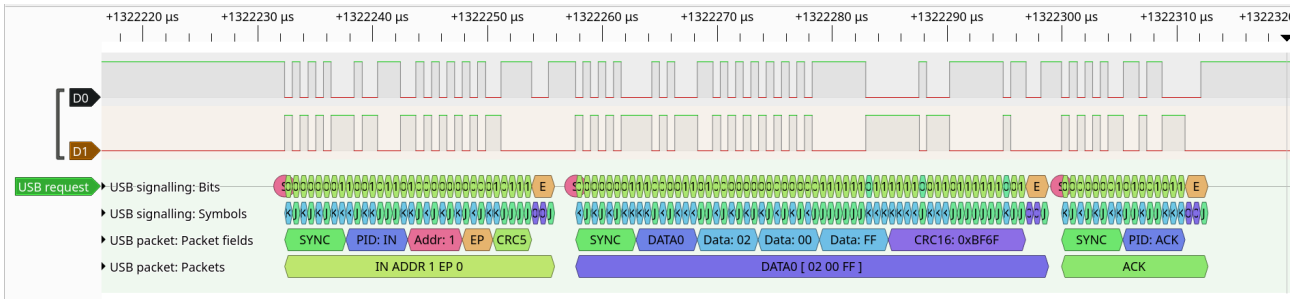

Рис. 10.5г. На очередной токен IN устройство отвечает пакетом DATA0 с размером блока данных менее максимального, что является признаком конца передаваемого ответа.

Получив такой укороченный пакет хост, как обычно, отправляет токен OUT с пустым блоком данных в пакете DATAx чтобы завершить обмен.

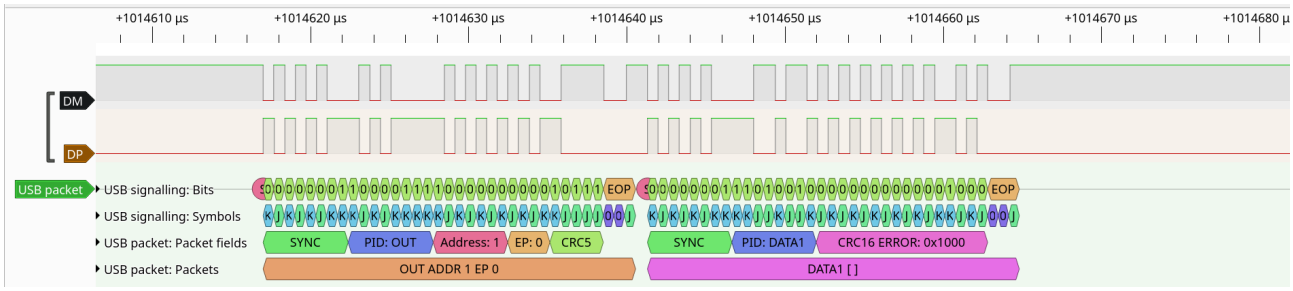

Рис. 10.5д. Зафиксирован сбой при передачи пакета DATA1 — неверный CRC16 (должно быть 0x0000, а принято 0x1000).

На рис. 10.5д. показан момент когда при передачи хостом пустого пакета DATA1, для завершения обмена, возникла ошибка CRC16. Факт ошибки был зафиксирован анализатором сигнала о чем он сообщает пользователю текстом «CRC16 ERROR» в поле контрольной суммы. Устройство получив «сбойный» пакет с данными просто игнорирует его.

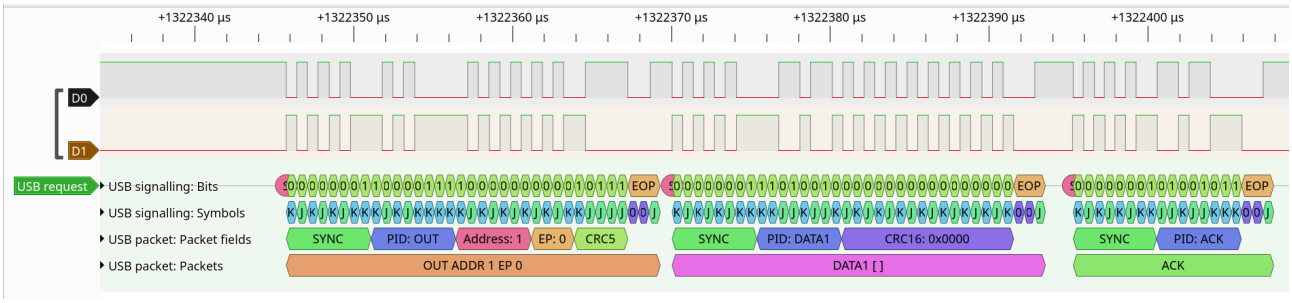

Рис. 10.5е. Перепосылка завершающего OUT и пустого DATA1.

Хост недождавшись от устройства подтверждающего токена ACK в течении 14 мкс принимает решение перепослать данные. Хост повторяет посылку токена OUT и следом за ним отправляет тот же пустой пакет DATA1. На этот раз пакет доходит успешно (рис. 10.5е), устройство подтверждает его прием токеном ACK и обмен окончательно завершается.

Устройство передало хосту восемь пакета с данными (исключая пустой пакет), семь из них длиной 8 байт и один, последний, пакет длиной 3 байта. Попробуем декодировать полученные хостом данные также как мы это делали со структурой «Device Descriptor». Но перед этим нам потребуется выделить известные нам типы структур опираясь на номер типа и длину (поля **bLength** и **bDescriptorType** присутствуют у всех структур в самом начале). Хостом получены следующие данные:

первый пакет: 0x09, 0x02, 0x3B, 0x00, 0x02, 0x01, 0x03, 0xA0;
второй пакет: 0x2D, 0x09, 0x04, 0x00, 0x00, 0x01, 0x03, 0x01;
третий пакет: 0x01, 0x02, 0x09, 0x21, 0x10, 0x01, 0x00, 0x01;
четвертый пакет: 0x22, 0x42, 0x00, 0x07, 0x05, 0x81, 0x03, 0x08;
пятый пакет: 0x01, 0x0A, 0x09, 0x04, 0x01, 0x00, 0x01, 0x03;
шестой пакет: 0x00, 0x00, 0x02, 0x09, 0x21, 0x10, 0x01, 0x00;
седьмой пакет: 0x01, 0x22, 0x17, 0x00, 0x07, 0x05, 0x82, 0x03;
восьмой пакет: 0x02, 0x00, 0xff;

Разным цветом обозначены данные относящиеся к трем различным структурам возвращенным в ответе: «Configuration Descriptor» (код 0x02) - зеленым, «Interface Descriptor» (код 0x04) - красным и «Endpoint Descriotor» (код 0x21) - синим. В процессе раскрашивания выявилось, что у нас имеется одна конфигурационная структура в которой содержится два интерфейса (два зеленых блока), в каждом по одной конечной точке (два синих блока). Также у нас имеются два блок данных структуры с кодом 0x21 выделенных желтым цветом — это структура типа «Additional Descriptor» о назначении которой может быть известно только драйверу от производителя устройства. Эту структуру декодировать не будем.

Распределим данные и наложим их на форматы соответствующих структур.

Декодируем структуру «Configuration Descriptor»:

Данные: 0x09, 0x02, 0x3B, 0x00, 0x02, 0x01, 0x03, 0xA0, 0x2D

Формат структуры:

```
typedef struct {
        uint8_t bLength;                // 1 Length of this descriptor = 9 bytes
        uint8_t bDescriptorType;        // 1 Descriptor type = CONFIGURATION (02h)
```

```
        uint16_t wTotalLength;          // 2 Total length including interface and endpoint
descriptors
        uint8_t bNumInterfaces;         // 1 Number of interfaces in this configuration
        uint8_t bConfigurationValue;    // 1 Configuration value used by SET_CONFIGURATION
to select this configuration
        uint8_t iConfiguration;         // 1 Index of string that describes this
configuration
        uint8_t bmAttributes;           // 1 Bit 7: Reserved (set to 1), Bit 6: Self-
powered, Bit 5: Remote wakeup
        uint8_t bMaxPower;              // 1 Maximum power required for this configuration
(in 2 mA units)
} USB10_Configuration;
```

Результат декодирования:

```
bLength = 0x09                — размер самой структуры составляет 9 байт;
bDescriptorType = 0x02        — тип структуры «Configuration Descriptor»;
wTotalLength = 0x003B         — количество байт данных включая все подчиненные интерфейсы
                                и конечные точки составляет 59 байт;
bNumInterfaces = 0x02         — количество интерфейсов входящих в эту конфигурацию: 2;
bConfigurationValue = 0x01    — идентификационный номер данной конфигурации: 1;
iConfiguration = 0x03         — идентификационный номер строки описывающей
                                данную конфигурацию: 3;
bmAttributes = 0xA0           — битовые поля с атрибутами:
                                      self-powered = off, remote wakeup = on;
bMaxPower = 0x2D              — потребляемый ток 45 * 2 = 90 мА;
```

Декодируем структуру «Interface Descriptor»:

Данные: 0x09, 0x04, 0x00, 0x00, 0x01, 0x03, 0x01, 0x01, 0x02

Формат структуры:

```
typedef struct {
        uint8_t bLength;                // 1 Length of this descriptor = 9 bytes
        uint8_t bDescriptorType;        // 1 Descriptor type = INTERFACE (04h)
        uint8_t bInterfaceNumber;       // 1 Zero based index of this interface
        uint8_t bAlternateSetting;      // 1 Alternate setting value
        uint8_t bNumEndpoints;          // 1 Number of endpoints used by this interface (not
including EP0)
        uint8_t bInterfaceClass;        // 1 Interface class
        uint8_t bInterfaceSubclass;     // 1 Interface subclass
        uint8_t bInterfaceProtocol;     // 1 Interface protocol
        uint8_t iInterface;             // 1 Index to string describing this interface
} USB10_Interface;
```

Результат декодирования:

```
bLength = 0x09                — размер структуры составляет 9 байт;
bDescriptorType = 0x04        — тип структуры «Interface Descriptor»;
bInterfaceNumber = 0x00       — индексный номер интерфейса (начиная с нуля) равен 0;
bAlternateSetting = 0x00      — альтернативная функция интерфейса: 0 (отсутствует);
bNumEndpoints = 0x01          — количество конечных точек в данном интерфейса: 1;
bInterfaceClass = 0x03        — класс 3 (HID устройство);
bInterfaceSubclass = 0x01     — подкласс 1 (1 — поддерживает Boot Protocol);
bInterfaceProtocol = 0x01     — тип устройства 1 (1 - «клавиатура», 2 - «мышь»);
iInterface = 0x02             — индексный номер строки содержащей описание интерфейса: 2;
```

В результате декодирования структуры «Interface Descriptor» мы видим, что подключенное устройство представляет собой устройство класса HID («Human Interface Device»), а по подклассу определяемое как «клавиатура». Напомню, что в полученной ранее структуре «Device Descriptor» также присутствовали параметры class и subClass, но они были установлены в ноль, это указывает на то, что класс устройства определяется в структуре интерфейса.

Декодируем структуру «Endpoint Descriptor»:

Данные: 0x07, 0x05, 0x81, 0x03, 0x08, 0x01, 0x0A;

Формат структуры:

```
typedef struct {
        uint8_t bLength;                // 1 Length of this descriptor = 7 bytes
        uint8_t bDescriptorType;        // 1 Descriptor type = ENDPOINT (05h)
        uint8_t bEndpointAddress;       // 1
              // Bit 3...0: The endpoint number
              // Bit 6...4: Reserved, reset to zero
              // Bit 7: Direction. Ignored for Control
              //        0 = OUT endpoint
              //        1 = IN endpoint
        uint8_t bmAttributes;           // 1
              // Bits 1..0: Transfer Type
              //        00 = Control
              //        01 = Isochronous
              //        10 = Bulk
              //        11 = Interrupt
              // If not an isochronous endpoint, bits 5...2 are reserved and must be
              // set to zero. If isochronous, they are defined as follows:
              // Bits 3..2: Synchronization Type
              //        00 = No Synchronization
              //        01 = Asynchronous
              //        10 = Adaptive
              //        11 = Synchronous
              // Bits 5..4: Usage Type
              //        00 = Data endpoint
              //        01 = Feedback endpoint
              //        10 = Implicit feedback Data endpoint
              //        11 = Reserved
        uint16_t wMaxPacketSize;        // 2 Maximum packet size for this endpoint
        uint8_t bInterval;              // 1 Polling interval in milliseconds for interrupt
endpoints
                // (1 for isochronous endpoints, ignored for control or bulk)
} USB10_Endpoint
```

Результат декодирования:

```
bLength = 0x09               — размер структуры составляет 9 байт
bDescriptorType = 0x05       — тип структуры «Endpoint Descriptor»
bEndpointAddress = 0x81      — ENDP=1, направление — к хосту (IN)
bmAttributes = 0x03          — для данных, тип передачи «Interrupt Transfer»
wMaxPacketSize = 0x0108      — максимальный размер блока данных при обмене: 8 байт
bInterval = 0x0A             — интервал опроса хостом: 10 мс.
```

3.8. «Device Class Definition for Human Interface Devices» (HID)

Так как основной причиной и стимулом к изучению темы USB для меня была необходимость подключать и использовать современные устройства ввода к разрабатываемой мной синтезируемой СнК для микросхем ПЛИС, то предлагаю поверхностно рассмотреть один из самых популярных классов USB устройств — «Human Interface Device — HID». Выше мы уже видели как выглядят основные дескрипторы для HID устройства типа «клавиатура», но как происходит обмен данными с таким устройством и в каком формате устройство выдает в хост информацию о нажатых клавишах? Как отличить данные «клавиатуры» от данных поступающих от «мыши»? Чтобы понять это, нам опять потребуется погрузиться в спецификацию.

Класс «HID» является одним из самых простых видов устройств подключаемых к шине USB. Его спецификация «Device Class Definition for Human Interface Devices Revision 1.11» была принята в Мае 2001 года, составляет всего 97 страниц. Позже были приняты различные дополнения, но в целом этот документ является актуальным по сей день. Однако это только часть айсберга. Спецификацией USB HID вводится такое понятие как «UsageID» - это некое событие поступающее от устройства ввода создаваемое человеком информирующее о текущем положении или состоянии элемента управления (клавиши клавиатуры, положения рычажка «джойстика»). Все возможные события пронумерованы - им присвоены числовые идентификаторы, и поименованы - присвоены имена. События собранны в различные группы - «страницы» («Pages»). Полное описание всех событий приводится в отдельном документе который озаглавлен «HID Usage Tables FOR Universal Serial Bus (USB)» или просто «Usage Tables». Этот документ постоянно обновляется, его актуальная версия на дату написания этой статьи имеет номер 1.6 от 2025 года. В данной версии «Usage Tables» описывается более 30 отдельных «страниц», в каждой из которых присутствует от несколько десятков до нескольких сотен кодов «UsageID». В документе приводятся подробные описания каждого события - в какой момент оно формируется устройством и как оно должно интерпретироваться драйвером устройства. Например, на странице **0x05 «Game Controls Page»** определено событие **0x21 «Turn Right/Left»** - это событие от аналогово геймпада при перемещении стика «влево/вправо». Более подробно некоторые из «страниц» документа «Usage Table» мы рассмотрим чуть ниже, а пока вернемся к общему описанию HID устройств.

Спецификацией HID покрываются следующие виды устройств ввода:

- Клавиатуры и устройства указания - «мышь», «трекболл», «джойстик»;
- элементы управления на панелях управления - поворотные регуляторы уровней (громкость), переключатели и ползунковые регуляторы («слайдеры»);
- элементы управления на различных устройства, таких как мобильные телефоны, пульты дистанционного управления видеомагнитофонами, пульты игровых консолей, элементы управления симуляторами («перчатка», педаль «газа», рулевое управление и педали, РУД).
- Также определены устройства, которые не предназначены для взаимодействия с человеком, но могу выдавать данные в аналогичном виде — считыватели BAR и QR кодов, термометры, вольтметры и т. д.

3.8.1. Дополнительные дескрипторы для HID

К стандартному набору дескрипторов (структур конфигурационных данных) спецификацией HID добавляется еще три типа дескрипторов: «HID Descriptor» (**0x21**), «Report Descriptor» (**0x22**) и «Physical Descriptor» (**0x23**), все они являются подчиненным подмножеством «Interface Descriptor», т. е. при формировании запроса GET_DESCRIPTOR необходимо указывать номер интерфейса в поле **wIndex**, а в поле **wValue** в старшем байте — тип, в младшем — номер дескриптора.

Структура «HID Descriptor» описывает какие еще другие HID дескрипторы присутствуют в устройстве, такие как «Report Descriptor» и «Physical Descriptor». Структура «Report Descriptor» описывает форматы данных генерируемых устройством — события или измерения. Фактически, данная структура содержит перечень элементов управления и размеры данных которые они формируют в рапорте при опросе устройства. Опциональная структура «Physical Descriptor» описывает какие части человеческого тела используются для активации элементов управления данного устройства и применяется очень редко. Кстати, в спецификации приводится таблица в которой перечислены и пронумерованы все части тела пригодные для ввода данных и формирования управляющего воздействия, их там аж 39 штук, включая левую и правую бровь. :-)

Общий вид дерева дескрипторов для HID устройства приведена на рис. 11.1. Форматы структуры «HID Descriptor» приведен в таблице 11.1.

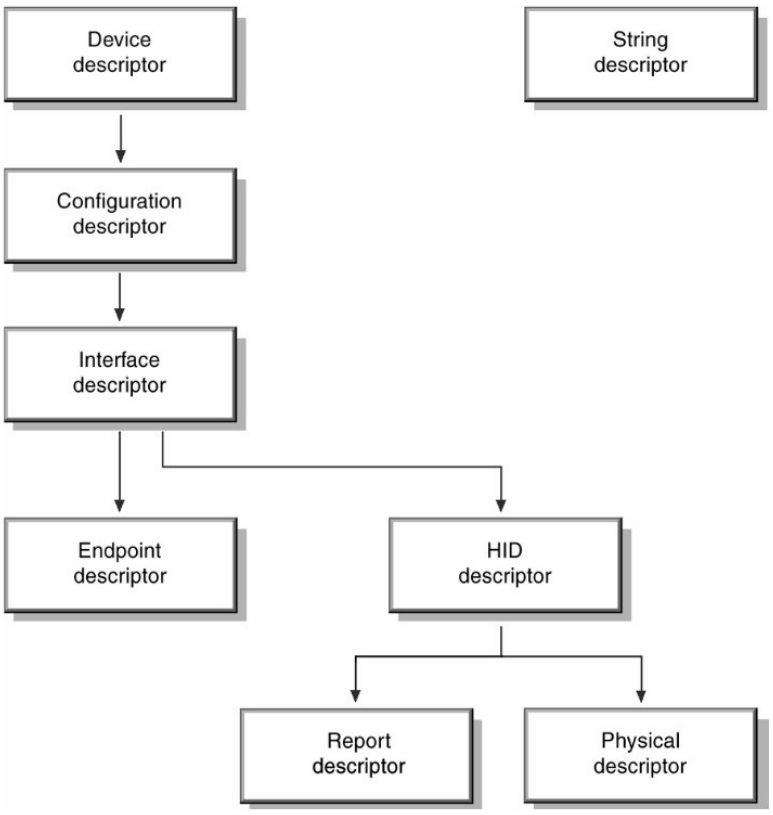

Рис. 11. Дерево дескрипторов для HID устройства.

Таблица 11.1. Описание полей структуры HID устройства «HID Descriptor»

Смеще-ние	Поле	Размер, байт	Тип или значение	Назначение
0	bLength	1	Number	Длина дескриптора в байтах (**9+N*3**), где N — число дополнительных дескрипторов за вычетом первого («Report Descriptor» всегда присутствует).
1	bDescriptorType	1	Constant	Фиксированное число (**0x21**).
2	bcdHID	2	BCD	Номер версии спецификации USB HID 0x0100 — версия HID 1.0, 0x0110 — версия HID 1.1. 0x0111 — версия HID 1.11.
4	bCountryCode	1	Number	Код страны для локализации устройства: 0x00 — устройство не локализовано. 0x08 — France. 0x09 — Germany 0x14 — Italy. 0x23 — Russia 0x32 — UK 0x33 — US Полный список стран приведен в спецификации.
5	bNumDescriptors	1	Number	Число подчиненных дескрипторов. Как минимум один - «Report Descriptor».
6	bDescriptorType	1	Constant	Идентификатор первого подчиненного дескриптора («Report Descriptor»)
7	wDescriptorLength	2	Number	Размер первого подчиненного дескриптора (размер «Report Descriotor»)
8	[bDescriptorType]	1	Constant	Идентификатор следующего (опционального) подчиненного дескриптора
10	[wDescriptorLength]	2	Number	Размер следующего (опционального) подчиненного дескриптора
	...	...	...	...

3.8.2. Структура описания форматов данных «Report Descriptor»

Формат структуры «Report Descriptor» у HID устройств несколько более сложен. Длина и содержимое данного дескриптора может меняться в зависимости о числа элементов данных требуемых для описания формируемого устройством потока данных в рапорте.

«Report Descriptor» состоит из массива элементов переменной длины. Элементы бывают двух видов «short item» (короткие) длиной от 1 до 5 байт, и «long item» (длинные) — от 3 до 258 байт. Оба вида описательных элементов состоят из двух частей: первая часть

элемента содержит байт с битовыми полями **bSize[1:0]**, **bType[3:2]** и **bTab[7:4]** — описывает тип данных, вторая часть сопровождает элемент дополнительными данными. Форматы элементов обоих видов приведены в таблицах 11.2 и 11.3.

Таблица 11.2. Формат описательного элемента вида «short item».

Байт 4	Байт 3	Байт 2	Байт 1	Байт 0 (первая часть)		
[data4]	[data2]	[data1]	[data0]	bTag [7:4]	bType [3:2]	bSize [1:0]

Назначение битов в первой части для короткого элемента следующее:

- Поле **bSize** кодирует количество дополнительных байт в элементе: 0 — отсутствует, 1 — один байт, 2 — два байта, 3 — четыре байта;
- Поле **bType** задает тип элемента: 0 — **Main**, 1 — **Global**, 2 — **Local**, 3 — **Reserved**;
  - Элемент типа **Main** непосредственно описывает формат данных передаваемых устройством в рапорте и может быть пяти разновидностей: **Input**, **Output**, **Feature**, **Collection** и **End Collection**.
  - Элемент типа **Global** задает характеристики элемента (например, минимальное и максимальное значение);
  - Элемент типа **Local** также задает характеристики элемента, но позволяет сделать это локально, для одного элемента.
- Поле **bTag** указывает на функциональное назначение элемента, оно зависит от типа элемента задаваемого полем **bType** (для подвидов **Main**, **Local** и **Global** определены разные значения **bTag**).
- Поля **dataX** содержат дополнительную информацию; для короткого описательного элемента это поле содержит размер данных ассоциируемых с этим элементом.

В таблице 11.4 приведен перечен всех функциональных элементов используемых для типов **Main, Global** и **Local**.

Таблица 11.4. Функциональные назначения элементов для типа Main, Global и Local.

Тип элемента (**bType**)	Значение первого байта (**bTag, bType, bSize**)	Биты в блоке данных (**dataX**)	Назначение битов
bType = Main			
Input	8’b1000_00_nn	Бит 0	{Data (0) | Constant (1)}
		Бит 1	{Array (0) | Variable (1)}
		Бит 2	{Absolute (0) | Relative (1)}
		Бит 3	{No Wrap (0) | Wrap (1)}
		Бит 4	{Linear (0) | Non Linear (1)}
		Бит 5	{Preferred State (0) | No Preferred (1)}
		Бит 6	No Null position (0) | Null state(1)}
		Бит 7	Reserved (0)
		Бит 8	{Bit Field (0) | Buffered Bytes (1)}
		Биты 31-9	Reserved (0)
Output	8’b1001_00_nn	Бит 0	{Data (0) | Constant (1)}
		Бит 1	{Array (0) | Variable (1)}
		Бит 2	{Absolute (0) | Relative (1)}

		Бит 3	{No Wrap (0) | Wrap (1)}
		Бит 4	{Linear (0) | Non Linear (1)}
		Бит 5	{Preferred State (0) | No Preferred (1)}
		Бит 6	{No Null position (0) | Null state(1)}
		Бит 7	{Non Volatile (0) | Volatile (1)}
		Бит 8	{Bit Field (0) | Buffered Bytes (1)}
		Биты 31-9	Reserved (0)
Feature	8’b1011_00_nn	Бит 0	{Data (0) | Constant (1)}
		Бит 1	{Array (0) | Variable (1)}
		Бит 2	{Absolute (0) | Relative (1)}
		Бит 3	{No Wrap (0) | Wrap (1)}
		Бит 4	{Linear (0) | Non Linear (1)}
		Бит 5	{Preferred State (0) | No Preferred (1)}
		Бит 6	{No Null position (0) | Null state(1)}
		Бит 7	{Non Volatile (0) | Volatile (1)}
		Бит 8	{Bit Field (0) | Buffered Bytes (1)}
		Биты 31-9	Reserved (0)
Collection	8’b1011_00_nn	0x00	Physical (group of axes)
		0x01	Application (mouse, keyboard)
		0x02	Logical (interrelated data)
		0x03	Report
		0x04	Named Array
		0x05	Usage Switch
		0x06	Usage Modifier
		0x07-0x7F	Reserved
		0x80-0xFF	Vendor-defined
End Collection	8’b1100_00_nn		Закрывает группу элементов
Reserved	8’b1101_00_nn - 8’b1111_00_nn		Не определено.
bType = Global			
Usage Page	8’b0000_01_nn		Беззнаковое 16-ти битное число номера «страницы» (Usage Page). Номер «страницы» месте с 16-ти битным номером «UsageID» определяют полное 32-х битное значение «UsageID».
Logical Minimum	8’b0001_01_nn		Минимальное логическое значение которое может принимать данный параметр или группа. Например, «мышь» может передавать координаты в диапазоне от 0-128, следовательно минимальное

			значение равно нулю. Задано в логических единица.
Logical Maximum	8’b0010_01_nn		Максимальное логическое значение параметра или группы.
Physical Minimum	8’b0011_01_nn		Минимальное физическое значение параметра.
Physical Maximum	8’b0100_01_nn		Максимальное физическое значение параметра.
Unit Exponent	8’b0101_01_nn		Значение экспоненты по основанию 10 задающей порядок единиц измеренния (микро, мили, кило, мега): 0x5 => 5, 0x6 => 6 0x7 => 7, 0x8 => -8 0x9 => -7, 0xA => -6 0xB => -5, 0xC => -4 0xD => -3, 0xE => -2 0xF => -1
Unit	8’b0110_01_nn		Единицы измерения (сантиметры, дюймы). см. таблицу на стр. 47 спецификации USB HID.
Report Size	8’b0111_01_nn		Цело беззнаковое число задающее размер полей (в битах) в рапорте.
Report ID	8’b1000_01_nn		Цело беззнаковое число задающее идентификатор рапорта.
Report Count	8’b1001_01_nn		Цело беззнаковое число задающее число полей в рапорте для данного элемента.
Push	8’b1010_01_nn		Помещает в стек текущее состояние глобальной таблицы элементов.
Pop	8’b1011_01_nn		Замещает текущую таблицу элементов значениями из вершины стека.
Reserved	8’b1100_01_nn - 8’b1111_01_nn		Зарезервировано.
bType = Local			
Usage	8’b0000_10_nn		Идентификатор UsageID предлагаемый для данного элемента.
Usage Minimum	8’b0001_10_nn		Минимальный UsageID для массива элементов.
Usage Maximum	8’b0010_10_nn		Максимальный UsageID для массива элементов.
Designator Index	8’b0011_10_nn		Идентификатор части тела (рука, нога) используемый для управления данным элементом.

			Ссылка на элемент структуры «Physical Descriptor».
Designator Minimum	8’b0100_10_nn		Минимальный идентификатор части тела, для массива элементов.
Designator Maximum	8’b0101_10_nn		Максимальный идентификатор части тела, для массива элементов.
String Index	8’b0111_10_nn		Ссылка на строку текста в «String Descriptor» описывающую данных элемент управления.
String Minimum	8’b1000_10_nn		Минимальный индекс строки описания массива элементов.
String Maximum	8’b1001_10_nn		Максимальный индекс строки описания массива элементов.
Delimiter	8’b1010_10_nn		Определяет начало и конце локальных параметров. 1 — начало, 0 — конец.
Reserved	8’b1011_10_nn - 8’b1111_10_nn		Зарезервировано.

Как уже было отмечено выше, элементы типа **Main Input**, **Main Output** и **Main Feature** используются для создания (описания) полей в рапорте — передаваемом пакете данных. Элементы типа **Input** описывают информацию о данных формируемых одним и более физическим средством управления (кнопки, регуляторы уровней, рычажки джойстика) и передаваемых от устройства к хосту. Элементы типа **Output** описывают данные передаваемые в сторону устройства, например состояния светодиодов или положение рычагов управления. Элементы типа **Feature** описывают конфигурационные параметры передаваемые от хоста к устройству.

Элементы типа **Collection** и **End Collection** используются для организации данных в группу (коллекцию). Например, «мышь» может быть описана как коллекция от двух до четырех элементов данных: **x**, **y**, **button1**, **button2**, где **x** и **y** — смещение соответствующей координаты, а **button1** и **button2** — битовые поля определяющие положение клавиш на манипуляторе. Элемент **Collection** создает группу, а **End Collection** закрывает её. Все элементы типа **Main** следующие за элементом **Collection** и до элемента **End Collection** входя в одну и ту же группу. Допускается создание вложенных групп.

Спецификацией USB HID определяется несколько типов групп (коллекций) задаваемых первым байтом в поле **dataX**. Их значения также приведены в таблице 11.4.

У «длинного» элемента формат отличается. В его первой части (в первом байте) поля **bTag**, **bType** и **bSize** всегда имеют фиксированное значение: **8`b1111_11_10** что является признаком «длинного» элемента. Следующий байт **bDataSize** задает длину блока данных, третий байт **bLongItemTag** задает функциональное назначение элемента. В таблице 11.3 показан общий формат «длинного» описательного элемента.

Таблица 11.3. Формат описательного элемента вида «long item».

Байт 3-261	Байт 2	Байт 1	Байт 0 (первая часть)		
[data длиной 3-258 байт]	**bLongItemTag**	**bDataSize**	**bTag =** 4“b1111	**bType =** (2“b11)	**bSize =** (2“b10)

Описательные элементы «длинного» вида в спецификации HID не рассматриваются, а данный формат отдается на откуп производителю аппаратуры, т. е. предполагается что драйвер устройства знает как работать с такими элементами.

Из приведенного выше описания структуры «Report Descriptor» можно сразу сделать вывод, что для её декодирования придется создать небольшую байт-код машину, которая проходит по данным дескриптора, изымает из него элементы один за одним, декодирует и «исполняет» их как программу. На сколько обоснована такая сложная система описания форматов данных мне судить сложно, но хочу заметить, что драйверы USB HID в составе ОС Linux и ОС FreeBSD этим не занимаются так как для широкого круга устройств ввода («мышь», клавиатура, «геймпады», сенсорные экраны) формат данных давно устоялся и не меняется. А раз так, то и нам этого делать тоже не потребуется. :-) Однако понимание того, как происходит разбор и интерпретация форматов данных в USB HID может оказаться полезным при работе с экзотическими устройствами при полном отсутствии документации. Ниже в листинге приведен пример интерпретации структуры «Report Descriptor» для устройства типа «геймпад» состоящего из рычага с двумя координатами X и Y, тремя кнопками и ручкой «газа».

Запросить дамп структуры «Report Descriptor» для HID устройства в ОС Linux можно с помощью утилиты **usbhid-dump**. В ОС FreeBSD аналогичная утилита называется **usbhidctl**. Кстати, последняя не только выводит дамп в шестнадцатеричной формате, но и декодирует его превращая в человеко-читаемый код.

Листинг 3. Пример интерпретации структуры «Report Descriptor» описывающей формат данных устройства типа «геймпад».

// Courtesy: https://wiki.osdev.org/USB_Human_Interface_Devices

```
static const uint8_t hidReportDescriptor [] =
{
    0x05, 0x01,    // UsagePage(Generic Desktop[1])
    0x09, 0x04,    // UsageId(Joystick[4])
    0xA1, 0x01,    // Collection(Application)
    0x85, 0x01,    //     ReportId(1)
    0x09, 0x01,    //     UsageId(Pointer[1])
    0xA1, 0x00,    //     Collection(Physical)
    0x09, 0x30,    //         UsageId(X[48])
    0x09, 0x31,    //         UsageId(Y[49])
    0x15, 0x80,    //         LogicalMinimum(-128)
    0x25, 0x7F,    //         LogicalMaximum(127)
    0x95, 0x02,    //         ReportCount(2)
    0x75, 0x08,    //         ReportSize(8)
    0x81, 0x02,    //         Input(Data, Variable, Absolute, NoWrap, Linear, PreferredState,
NoNullPosition, BitField)
    0x05, 0x09,    //         UsagePage(Button[9])
    0x19, 0x01,    //         UsageIdMin(Button 1[1])
    0x29, 0x03,    //         UsageIdMax(Button 3[3])
    0x15, 0x00,    //         LogicalMinimum(0)
    0x25, 0x01,    //         LogicalMaximum(1)
    0x95, 0x03,    //         ReportCount(3)
    0x75, 0x01,    //         ReportSize(1)
    0x81, 0x02,    //         Input(Data, Variable, Absolute, NoWrap, Linear, PreferredState,
NoNullPosition, BitField)
    0xC0,          //     EndCollection()
    0x05, 0x02,    //     UsagePage(Simulation Controls[2])
    0x09, 0xBB,    //     UsageId(Throttle[187])
    0x15, 0x80,    //     LogicalMinimum(-128)
    0x25, 0x7F,    //     LogicalMaximum(127)
    0x95, 0x01,    //     ReportCount(1)
    0x75, 0x08,    //     ReportSize(8)
    0x81, 0x02,    //     Input(Data, Variable, Absolute, NoWrap, Linear, PreferredState,
NoNullPosition, BitField)
    0x75, 0x05,    //     ReportSize(5)
```

```
    0x81, 0x03,    //     Input(Constant, Variable, Absolute, NoWrap, Linear, PreferredState,
NoNullPosition, BitField)
    0xC0,          // EndCollection()
};
```

3.8.3. Запросы к HID устройству

Согласно спецификации USB HID получение информации об устройстве, его конфигурации или состоянии, а так же изменение конфигурации или состояния отдельных «фич» осуществляется стандартными запросами типа «класс» отправляемых на конечную точку по-умолчанию («default endpoint», то есть ENDP = 0) предназначенную для управления и инициализацией устройством. При этом назначение полей в запросе следующее:

- **bmRequestType** может принимать одно из двух значений: **0b00100001** — для передачи данных Host→Device и **0b10100001** для Device→Host. В данных битовых значениях закодирован тип запроса: **Type = Class** и получатель: **Recipient = Interface**.
- **bRequest** содержит один байт кода запроса:
  - 0x01 — GET_REPORT;
  - 0x02 — GET_IDLE;
  - 0x03 — GET_PROTOCOL;
  - 0x09 — SET_REPORT;
  - 0x0A — SET_IDLE;
  - 0x0B — SET_PROTOCOL.

- **wValue** содержит два байта номера запрашиваемого параметра/значения (зависит от запроса).
- **wIndex** содержит два байта смещения при выдачи ответа.
- **wLength** содержит два байта указывающих на максимальную длину блока полезных данных при выдаче ответа.

Запрос «GET_REPORT Request» позволяет хосту получать рапорты через «default pipe». Рапорт представляет собой набор событий или измерений накопленных элементом управления (или группой элементов в составе одного ReportID) к моменту запроса или полученных в момент запроса. Данный запрос **не** предназначен для регулярного опроса устройства так как создает большой объем трафика, но может быть быть пригоден для получения состояния элементов управления, а также считывания и установки состояния однобитовых параметров («фич») в момент инициализации.

Таблица 12.1. Формат пакета «GET_REPORT Request».

Смещение	Поле	Размер, байт	Тип или значение	Назначение
0	bmRequestType	1	**0b10100001**	Бит 7 = **0b1** - Направление «Device to Host» Биты 6..5 = **0b01** - Тип запроса «Class» Биты 4..0 = **0b0001** — получатель «Interface».
1	bRequest	1	Value	**0x01** — GET_REPORT
2	wValue	2	Value	В младшем байте указывается запрашиваемый **Report ID** или 0 если не применимо.

				Старший байт содержит **Report Type**, устанавливается в одно из следующих значений: **0x01**, Input — запрос входных данных. Для этого вида запроса используется «Interrupt Transfer IN», а запрос может быть направлен только на конечную точку у которой установлен флаг поддержки «Interrupt IN». Формат данных в рапорте такой же как у «Interrupt Transfer». **0x02**, Output — установка значения параметра («фичи»). Например, изменение состояние индикаторных LED светодиодов. Аналогично запросу Input используется формат приняты для «Interrupt Transfer», а конечная точка должна иметь флаг «Interrupt OUT». Если на устройстве нет ни одной точки с данным флагом, то допускается отправить запрос Output на дефолтную конечную точку. **0x03**, Feature — позволяет запросить состояние однобитного параметра (фичи).
4	wIndex	2	Index/Offset	Номер интерфейса.
6	wLength	2	Count	Количество байт данных в ответе ожидаемых хостом.

Запрос «SET_REPORT Request» позволяет хосту отправить рапорт на устройство чтобы передать такие-то данные или установить состояние элементов управления. Формат аналогичен запросу «GET_REPORT Request». Устройство может игнорировать такой запрос если он не применим или не поддерживается. В некоторых устройствах данный вид запрос может использоваться для обнуления смещения (сброса и синхронизации) потока данных конечной точки.

Таблица 12.2. Формат пакета «SET_REPORT Request».

Смещение	**Поле**	**Размер, байт**	**Тип или значение**	**Назначение**
0	bmRequestType	1	**0b00100001**	Бит 7 = **0b0** - Направление «Host to Device» Биты 6..5 = **0b01** - Тип запроса «Class» Биты 4..0 = **0b0001** — получатель «Interface».
1	bRequest	1	Value	**0x09** — SET_REPORT
2	wValue	2	Value	Report Type и Report ID, см. «GET_REPORT Request».

4	wIndex	2	Index/Offset	Номер интерфейса.
6	wLength	2	Count	Количество байт данных в запросе (в пакете DATAx).

Запрос «GET_IDLE Request» позволяет хосту выяснить состояние, а **«SET_IDLE Requet»** - перевести устройство в пассивное состояние или снизить интенсивность потока формируемых прерываний. Хост может перевести устройство в пассивное состоянии на некоторое время. Если устройство находится в пассивном состоянии, то опрос данных с него будет возвращать NAK. Время пребывания устройства в таком состоянии указывается в поле **wValue** в его старшем байте (если 0 - бесконечное время). Младший байт поля **wValue** содержит ReportID (идентификатор рапорта) для которого требуется временно прекратить формирование отчета. Если ReportID равен нулю, то формирование рапортов прекращается для всего интерфейса. Поле **wIndex** содержит номер интерфейса. Поле **wLength = 0** для SET_IDLE и **1** для GET_IDLE. Возвращаемый ответ длиной один байт находится в блоке данных передаваемого следом пакета DATAx. Поле **bmRequestType** равно **0b00100001** для SET_IDLE и **0b10100001** для GET_IDLE.

Интересно, что по-умолчанию интервал формирования рапортов для клавиатуры равна 500 мс. Для джойстиков и «мышей» - бесконечность, т. е. рапорты формируются только при изменении состояния.

Таблица 12.3. Формат пакета «GET_IDLE Request».

Смеще-ние	Поле	Размер, байт	Тип или значение	Назначение
0	bmRequestType	1	**0b10100001**	Бит 7 = **0b1** - Направление «Device to Host» Биты 6..5 = **0b01** - Тип запроса «Class» Биты 4..0 = **0b0001** — получатель «Interface».
1	bRequest	1	Value	**0x02** — GET_IDLE.
2	wValue	2	Value	Старший байт = 0. Младший байт содержит Report ID для которого запрашивается интервал бездействия.
4	wIndex	2	Index/Offset	Номер интерфейса.
6	wLength	2	Count	1
	DATAx	1	0xXX	Содержит возвращаемое значение параметра Idle Duration в единицах по 4мс (доступе диапазон от 0.004 до 1.020 секунды).

Таблица 12.4. Формат пакета «SET_IDLE Request».

Смеще-ние	Поле	Размер, байт	Тип или значение	Назначение
0	bmRequestType	1	**0b00100001**	Бит 7 = **0b0** - Направление «Host to Device»

				Биты 6..5 = **0b01** - Тип запроса «Class» Биты 4..0 = **0b0001** — получатель «Interface».
1	bRequest	1	Value	**0x0A** — SET_IDLE.
2	wValue	2	Value	Старший байт содержит значение устанавливаемого Duration в единицах по 4мс, что дает диапазон от 0.004 до 1.020 секунды. Точно задания — 10% +/- 2мс. Младший байт содержит Report ID для которого требуется установить интервал бездействия. Если равен нулю, то данное значение Duration применяется ко всем ReportID.
4	wIndex	2	Index/Offset	Номер интерфейса.
6	wLength	2	Count	0

Запрос «GET_PROTOCOL Request» позволяет хосту выяснить какой протокол получения рапортов на данный момент активирован. Всего предусматривается два протокола: **«Report Protocol»** - через «GET_REPORT Request» и **«Boot Protocol»**. Значения полей следующие: **bmRequestType = 0b10100001**, **wValue = 0**, **wIndex** — номер интерфейса, **wLength = 1**.

Данный запрос возвращается один байт в блоке данных ответа в пакете DATAx. Его значение следующее: **0** — активирован «Boot Protocol», **1** — активирован «Report Protocol».

Таблица 12.5. Формат пакета «GET_PROTOCOL Request».

Смеще-ние	Поле	Размер, байт	Тип или значение	Назначение
0	bmRequestType	1	**0b10100001**	Бит 7 = **0b1** - Направление «Device to Host» Биты 6..5 = **0b01** - Тип запроса «Class» Биты 4..0 = **0b0001** — получатель «Interface».
1	bRequest	1	Value	**0x03** — GET_PROTOCOL
2	wValue	2	Value	0
4	wIndex	2	Index/Offset	Номер интерфейса.
6	wLength	2	Count	1
	DATAx	1	0xXX	Содержит возвращаемое значение: 0 — Boot Protocol, 1 — Report Protocol.

Запрос «SET_PROTOCOL Request» позволяет хосту активировать один из двух протоколов получения рапортов. Значения полей следующие: **bmRequestType = 0b00100001**,

wValue = 0 для «Boot Protocol» или **1** для «Report Protocol», wIndex = номер интерфейса, **wLength = 0**. Блок данных отсутствует (пустой).

Таблица 12.6. Формат пакета «SET_PROTOCOL Request».

Смещение	Поле	Размер, байт	Тип или значение	Назначение
0	bmRequestType	1	**0b00100001**	Бит 7 = **0b0** - Направление «Host to Device» Биты 6..5 = **0b01** - Тип запроса «Class» Биты 4..0 = **0b0001** — получатель «Interface».
1	bRequest	1	Value	**0x0B** — SET_PROTOCOL.
2	wValue	2	Value	0 — активировать Boot Protocol или 1 — активировать Report Protocol.
4	wIndex	2	Index/Offset	Номер интерфейса.
6	wLength	2	Count	0

Согласно спецификации, все USB HID устройства по-умолчанию активируют «Report Protocol». В спецификации также подчеркивается, что драйвер на стороне хоста не должен полагаться на этот ненадежный факт и обязан активировать удобный ему протокол явным способом.

3.8.4. USB HID «Report Protocol»

Как отмечалось выше «Report Protocol» создан для получения состояния элементов управления или результатов измерений накопленных (полученных) к моменту запроса. Данные рапорта пересылаются от устройства к хосту либо через «Interrupt Transfer IN», либо через запрос «Control Transfer IN» к дефолтной конечной точке (ENDP = 0). Также предусматривается отправка рапортов от хоста к устройству двумя аналогичными механизмами — через «Interrupt Transfer OUT» или через «Control Transfer OUT». Передача рапортов от хоста к устройству позволяет устанавливать положения элементов управления или значения приборов индикации (Indicators), например, включать LED светодиоды.

Первый байт рапорта содержит ReportID. В рапорте передаются данные только тех элементов управления, которые соотносятся с запрашиваемым ReportID (если в запросе ReportID = 0, то передаются данные всех элементов), упакованные по границе бита, т. е. без выравнивания и смещения, один за другим, в той последовательности как они описаны в «Report Descriptor». Размер (в битах) каждого элемента также определяется структурой «Report Descriptor».

Если в структуре «Report Descriptor» отсутствует определение ReportID, то считается что все данные относятся к одному и тому же рапорту, а ReportID не включается в пакет. То есть данные в рапорте следуют элемент за элементом, без тега ReportID. Общий вид формата рапорта приведен в таблице 13.1

Таблица 13.1. Формат блока данных рапорта (общий вид).

Биты: 23 22 21 20 19 18 17 16 14 14	Биты: 13 12 11 10 9 8	Биты: 7 6 5 4 3 2 1 0
Состояние элемента 1	Состояние элемента 0	ReportID (опционально)

Если у устройства есть несколько рапортов для выдачи хосту, то в рамках одного обмена данными, могу передаваться несколько рапортов, каждый со своим уникальным ReportID.

На формат рапорта передаваемого через «Report Protocol» накладываются следующие ограничения:

- Событие от элемента управления не может занимать более 4-х байт.
- Только один рапорт возможен в одной USB транзакции.
- Рапорт может занимать более одной транзакции. Например, рапорт занимающий 10 байт потребует две транзакции для «Low Speed» (блоки 8 и 2 байта).
- Конец передачи рапорта инициируется передачей укороченного пакета (остатка). Если рапорт полностью входит в максимальный размер пакета, т. е. его длина равна **wMaxPacketSize**, то ничего не передается.
- Рапорты выравнивают по границе байта. Если необходимо, в конец добавляют нулевые биты для создания полного байта.

3.8.5. «Report Protocol» для клавиатурных устройств

Рассмотрим как это работает для устройств снабженных массивом клавиш (клавиатуры и «геймпады»). В таких устройствах нажатие клавиши вызывает появления события с идентификатором **UsageID**, при этом каждой клавише присваивается свой уникальный UsageID. Идентификаторы событий UsageID от различных источников собраны и сгруппированы в таблицу называемую **Usage Pages**, так что события от одного вида источника (от клавиатуры) находятся на одной «странице» этой таблицы, то есть «Usage Page» определяет группу событий. Для клавиатур отведена страница «Keyboard/Keypad Page» с кодом 0x07, она содержит перечень всех идентификаторов событий которые могут формировать клавиатурные устройства. Согласно этой таблице, клавиша «Escape» формирует событие «Keyboard ESCAPE» с кодом UsageID = 0x29. Полный перечень всех действующих кодов UsageID приведен в документе «HID Usage Tables FOR Universal Serial Bus (USB)» который можно получить на сайте ассоциации «USB Implementation Forum (USB-IF)».

Среди множества кодов UsageID могут присутствовать служебные коды которые отображают общее состояние устройства (например, аварийную ситуацию). Обычно такие коды находятся в нижнем диапазоне значений UsageID (от нулевого) для данной страницы кодов. В таблице 13.2 приведены служебные и некоторые обычные коды UsageID для клавиатурных устройств.

Таблица 13.2. Некоторые коды UsageID для страницы 0x07 (Keyboard/Keypad).

UsageID	Наименование	Описание
0x00	Reserved	Зарезервированный код, обычно используется для заполнения пустых данных при отпускании клавиш.
0x01	Keyboard ErrorRollOver	Ошибочное состояние возникающее когда комбинация нажатых клавиш не может быть правильно закодирована матрицей клавиатуры.
0x02	Keyboard POSTFail	Ошибка аппаратуры возникающая на стадии POST тестирования.
0x03	Keyboard ErrorUndefined	Общая (не определенная) ошибка клавиатуры.
0x04	Keyboard a and A	Нажата клавиша «A»
0x05	Keyboard b and B	Нажата клавиша «B»

0x06	Keyboard c and C	Нажата клавиша «C»
…	…	…
0x28	Keyboard Return (ENTER)	Нажата клавиша «Enter»
0x29	Keyboard ESCAPE	Нажата клавиша «Escape»
0x2A	Keyboard DELETE (Backspace)	Нажата клавиша «Backspace»
0x2B	Keyboard Tab	Нажата клавиша «Tab»
…	…	...

При нажатии нескольких клавиш одновременно в рапорт попадают UsageID коды только тех клавиш которые находятся в нажатом («key closed») состоянии в момент запроса. Последовательность кодов в рапорте соответствует последовательности их нажатия. При отпускании клавиши UsageID коды отпущенных клавиш удаляются из рапорта, нажатые/удерживаемые клавиши при этом остаются и смещаются заполняя пустой «слот». Таким образом отслеживая данные двух последовательных рапортов можно определить последовательность нажатий и отпусканий любого числа клавиш. Однако, так как для USB «Low Speed» размер блока данных в пакете DATAx не может превышать 8 байт (напомню, что «Interrupt Transfer» предусматривает передачу только одного блока данных), то список событий ограничен 6-ю одновременно нажатыми клавишами.

Не смотря на то, что для клавиш-модификаторов предусмотрены свои коды UsageID, события от них передаются в виде группы битовых полей (массив из одно-битовых полей). Стандартные клавиатуры предусматривают всего 8 модификаторов, то есть передается один байт. Ниже в таблице 13.3 приведен перечен клавиш-модификаторов стандартной клавиатуры и ассоциированные с ними биты.

Таблица 13.3. Клавиши-модификаторы.

Номер бита	**Наименование клавиши**
0	LEFT CTRL
1	LEFT SHIFT
2	LEFT ALT
3	LEFT GUI
4	RIGTH CTRL
5	RIGHT SHIFT
6	RIGTH ALT
7	RIGTH GUI

Рассмотрим пример клавиатурного устройства формальное описание которого, т. е. содержимое структуры «Report Descriptor», приведено в листинге 4.

Листинг 4. Пример «Report Descriptor» для стандартной клавиатуры.

```
Usage Page (Generic Desktop),
Usage (Keyboard),
Collection (Application),
        Report Size (1),
        Report Count (8),
        Usage Page (Key Codes),
        Usage Minimum (224),
        Usage Maximum (231),
        Logical Minimum (0),
        Logical Maximum (1),
        Input (Data, Variable, Absolute),    // Клавиши-модификаторы
        Report Count (1),
        Report Size (8),
```

```
        Input (Constant), // Резервный байт
        Report Count (5),
        Report Size (1),
        Usage Page (LEDs),
        Usage Minimum (1),
        Usage Maximum (5),
        Output (Data, Variable, Absolute),   // Светодиодная индикация (LED report)
        Report Count (1),
        Report Size (3),
        Output (Constant),// Выравнивающие 3 бита после 5-ти битов индикации
        Report Count (6),
        Report Size (8),
        Logical Minimum (0),
        Logical Maximum(255),
        Usage Page (Key Codes),
        Usage Minimum (0),
        Usage Maximum (255),
        Input (Data, Array), // 6 шт UsageID кодов в диапазоне значений от 0 до 255
End Collection
```

Согласно приведенному выше описанию, рапорт от такого устройства будет содержать 8 байт: нулевой байт — битовый массив для клавиш-модификаторов, первый байт — резервный и равен 0x00, байты с 2 по 7 — данные (UsageIDs) о нажатых клавишах. Следует обратить внимание на то, что данным дескриптором не устанавливается ReportID, соответственно в рапорте он будет отсутствовать так как ReportID с идентификатором 0x00 не передается если в рапорте все данные от одного и того же ReportID. Пример рапорта такого формата приведен в таблице 13.3, он является стандартным для большинства клавиатур.

Таблица 13.3. Формат рапорт от стандартной клавиатуры, с примером.

Номера битов:	63 … 56	55 … 48	47 … 40	39 … 32	31 … 24	23 … 16	15 … 8	7 … 0
Назначение:	UsageID	UsageID	UsageID	UsageID	UsageID	UsageID	Reserved	Bitmap
Пример:	**0x00**	**0x00**	**0x00**	**0x00**	**0x00**	**0x29**	**0x00**	**0x14**
Описание:	Нет данных	Нет данных	Нет данных	Нет данных	Нет данных	ESCAPE	Нет данных	L_ALT R_CTRL

3.8.6. «Report Protocol» для манипуляторов «мышь»

В листинге 5 приведено формальное описание рапорта для устройства ввода типа «мышь». Рапорт для такого устройства будет состоять из одного байта ReportID = 0x0A), двух байт для относительных координат X и Y и одного байта состояния кнопок (младшие три бита). Всего в рапорте будет передано 4 байта. Пример такого рапорта приведен в таблице 13.4.

Листинг 5. Пример «Report Descriptor» для манипулятора «мышь».

```
Usage Page (Generic Desktop),
Usage (Mouse),
Collection (Application),
        Usage (Pointer),
        Collection (Physical),
                Report ID (0A), // Установить идентификатор ReportID = 0x0A
                Usage (X), Usage (Y), // два UsageID для позиционных устройств
                Logical Minimum (-127),
                Logical Maximum (127),
                Report Size (8), Report Count (2),
                Input (Data, Variable, Relative), // два байта позиционных данных (X и Y)
                Logical Minimum (0), // Диапазон значений: -127 … 127
                Logical Maximum (1),
```

```
                Report Count (3), Report Size (1),
                Usage Page (Button Page),
                Usage Minimum (1),
                Usage Maximum (3),
                Input (Data, Variable, Absolute), // три бита для кнопок (1, 2, 3)
                Report Size (5),
                Input (Constant), // 5 бит выравнивания после кнопок
        End Collection,
End Collection
```

Таблица 13.4. Формат рапорт от манипулятора «мышь», с примером.

Номера битов:	31 … 24	23 … 16	15 … 8	7 … 0
Назначение:	Bitmap	UsageID	UsageID	ReportID
Пример:	**0x01**	**0x1E**	**0xFB**	**0x0A**
Описание:	Нажата кнопка 1	Y=30	X = -5	Идентификатор рапорта

Для комбинированны (совмещенных) устройств в описании может присутствовать две и более коллекции с различными ReportID и разным набором элементов. Такие устройства генерируют более сложные рапорты, разбирать (парсить) которые не так то просто без наличия точного формального описания, а как мы уже поняли, получить и разобрать описание «Report Descriptor» - задача не из простых. Однако же, на практике, подавляющая часть комбинированных устройств типа «клавиатура+мышь» представляют несколько интерфейсов с различными конечными точками, по одной на каждое средство ввода, что позволяет драйверу работать с такими устройствами как с отдельными независимыми устройствами сводя задачу разбора рапортов к двум выше приведенным примерам.

3.8.7. USB HID «Boot Protocol»

В какой-то момент разработчики спецификации USB HID осознали, что придуманный ими универсальный способ описания формата передаваемых данных оказался крайне не прост при реализации на «голом железе» без сложной системы драйверов и может создать массу проблем совместимости. Примером такой системы может быть BIOS исполняемый при загрузке ПК, код которой ограничен размером микросхемы ПЗУ (в конце 90-х объем микросхем ПЗУ был небольшого размера). Иными словами, потребовалось вводить упрощения и зафиксировать (сделать неизменяемым) несколько широко используемых форматов для устройств ввода, таких как клавиатура и «мышь», с целью дать возможность использования их в BIOS-ах и в загрузчиках операционных систем для организации интерфейса пользователя в процессе загрузки. Устройства, работу с которыми нужно обеспечить в момент до запуска на ПК полноценной операционной системы, принято называть «Boot Devices».

В спецификацию USB HID были добавлены приложения (**Appendix B.1** и **Appendix B.2**) описывающие упрощенные форматы дескрипторов для клавиатуры и манипулятора «мышь». Эти форматы жестко зафиксированы и не изменяются, что позволяет неизощренному программному обеспечению легко парсить данных от «Boot» устройств. Чтобы устройство начало выдавать данные в таком предопределенном формате, необходимо установить тип активного протокола в состояние «Boot Protocol» с помощью запроса «SET_PROTOCOL Request» (**wValue = 0**). Сам механизм запроса и передачи данных ничем не отличается от описанного выше способа определенного для «Report Protocol».

Ниже в листингах 6 и 7 приведены описания форматов устройств ввода определенных согласно приложениям **B.1** (клавиатура) и **B.2** («мышь») спецификации. В таблицах 13.5 и

13.6 приведены примеры рапортов поступающих от таких устройств в режиме «Boot Protocol». Этими «упрощенными» форматами мы воспользуемся далее при написании простейшего драйвера.

Листинг 6. «Report Descriptor» для стандартной клавиатуры согласно Appendix B.1.

```
Usage Page (Generic Desktop),
Usage (Keyboard),
Collection (Application),
	Report Size (1),
	Report Count (8),
	Usage Page (Key Codes),
	Usage Minimum (224),
	Usage Maximum (231),
	Logical Minimum (0),
	Logical Maximum (1),
	Input (Data, Variable, Absolute), ;Modifier byte
	Report Count (1),
	Report Size (8),
	Input (Constant), ;Reserved byte
	Report Count (5),
	Report Size (1),
	Usage Page (LEDs),
	Usage Minimum (1),
	Usage Maximum (5),
	Output (Data, Variable, Absolute), ;LED report
	Report Count (1),
	Report Size (3),
	Output (Constant), ;LED report padding
	Report Count (6),
	Report Size (8),
	Logical Minimum (0),
	Logical Maximum(255),
	Usage Page (Key Codes),
	Usage Minimum (0),
	Usage Maximum (255),
	Input (Data, Array),
End Collection
```

Листинг 7. «Report Descriptor» для манипулятора мышь согласно Appendix B.2.

```
Usage Page (Generic Desktop),
Usage (Mouse),
Collection (Application),
	Usage (Pointer),
	Collection (Physical),
		Report Count (3),
		Report Size (1),
		Usage Page (Buttons),
		Usage Minimum (1),
		Usage Maximum (3),
		Logical Minimum (0),
		Logical Maximum (1),
		Input (Data, Variable, Absolute),
		Report Count (1),
		Report Size (5),
		Input (Constant),
		Report Size (8),
		Report Count (2),
		Usage Page (Generic Desktop),
		Usage (X),
		Usage (Y),
		Logical Minimum (-127),
		Logical Maximum (127),
		Input (Data, Variable, Relative),
	End Collection,
End Collection
```

Таблица 13.5. Формат рапорт от стандартной клавиатуры, согласно Appendix B.1.

Номера битов:	63 … 56	55 … 48	47 … 40	39 … 32	31 … 24	23 … 16	15 … 8	7 … 0
Назначение:	UsageID	UsageID	UsageID	UsageID	UsageID	UsageID	Reserved	Bitmap
Пример:	**0x00**	**0x00**	**0x00**	**0x00**	**0x00**	**0x29**	**0x00**	**0x14**
Описание:	Нет данных	Нет данных	Нет данных	Нет данных	Нет данных	ESCAPE	Резерв	L_ALT R_CTRL

Таблица 13.6. Формат рапорт от манипулятора «мышь», согласно Appendix B.2.

Номера битов:	31 … 24	23 … 16	15 … 8	7 … 0
Назначение:	(Optional)	UsageID	UsageID	Bitmap
Пример:	**0x00**	**0x1E**	**0xFB**	**0x01**
Описание:	Состояние «колеса»	Y=30	X = -5	Нажата кнопка 1

3.8.8. Пример запроса к HID устройству: установка состояния светодиодной индикации

Чтобы стало более понятно как работают запросы к HID устройствам предлагаю рассмотреть один из самых простых и, на мой взгляд, очень полезных запросов — запрос к USB HID клавиатуре на установку состояния светодиодной индикации (CapsLock, NumLock, ScrollLock LEDs). Сразу перейдем к делу и посмотрим на осциллограммы представленные на рис. 12.1, 12.2 и 12.3.

Видно, что хост сначала отправляет токен типа SETUP и пакет DATA0, содержащий тело запроса, на нулевую (дефолтную) конечную точку устройства с адресом #1. Устройство успешно приняло этот пакет, что подтверждается последующим токеном ACK.

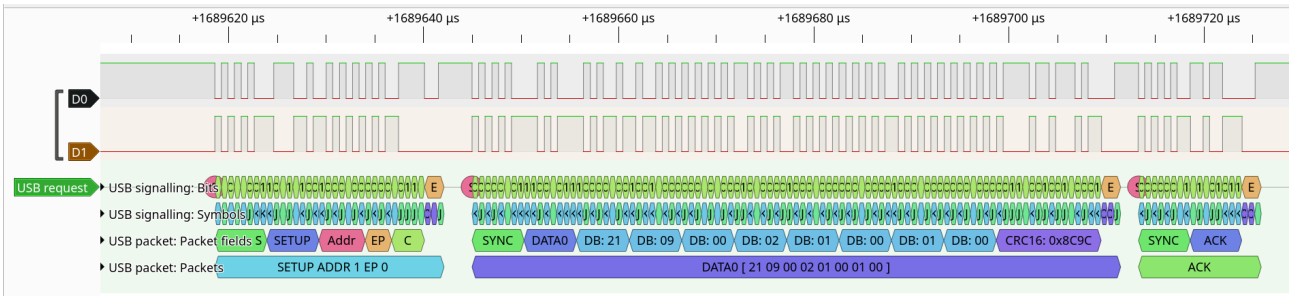

Рис. 12.1. Запрос «SET_REPORT Request» к устройству #1 на дефолтную конечную точку (ENDP=0).

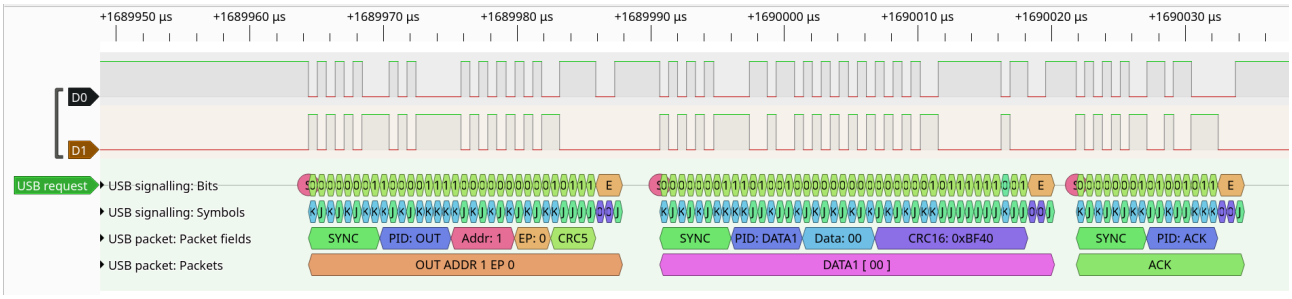

Рис. 12.2. Передача блока данных для запроса «SET_REPORT Request». Содержит один байт полезной нагрузки.

Следом за токенами SETUP хост отправляет токен OUT и тут же отправляет блок данных в пакете DATA1. Блок данных содержит всего один байт полезной нагрузки равный **0x00**. Прием этого пакета тоже успешно подтверждается устройством.

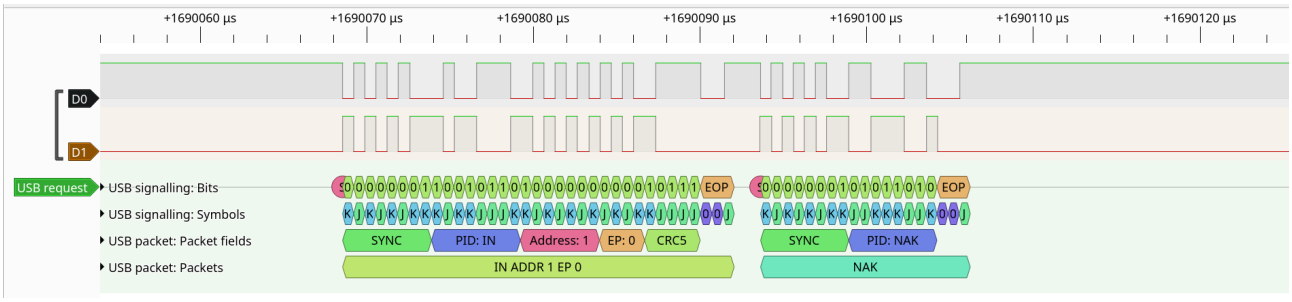

Рис. 12.3. Завершение обмена после запроса «SET_REPORT Request». Устройство отвечает токеном NAK сообщая, что никаких данных у него для хоста нет.

Далее хост посылает токен IN на этот же адрес (ADDR:ENDP = 1:0) с целю завершить обмен в рамках данного запроса, на что устройство отвечает токеном NAK сообщая хосту, что данных для него нет и обмен в рамках данного запроса успешно закончен. На первый взгляд может показаться, что эта финализирующая последовательность IN/NAK является чисто ритуальной, но это не совсем так. Хост должен убедиться, что данные которые он передал не только успешно дошли до устройства, но и были корректно обработаны им. Если данные некорректные или устройство не смогло их обработать по какой-то причине, то в ответ на IN хост получит токен STALL и для дальнейшей работы устройство потребует сброса и инициализации! Таким образом этот ритуальный IN/NAK дает хосту понять, что с устройством все в порядке и можно продолжать работу. Опыт показал, что если хост не посылает IN/NAK, то большинство устройств продолжают корректно работать, но приятые данные могут не примениться/активироваться - некоторые клавиатуры, например, не изменяют состояние светодиодной индикации.

Выпишем данные из запроса, наложим их на структуру «USB Request» (см. таблицу 10) и получим следующие значения полей запроса:

Запрос: 0x21, 0x09, 0x00, 0x02, 0x01, 0x00, 0x01, 0x00.

- Поле **bmRequestType = 0x21** — запрос типа «Class», данные будут передаваться от хоста к устройству, получателем является интерфейс.
- Поле **bRequest = 0x09** — запрос вида SET_REPORT.
- Поле **wValue = 0x0200** — содержит ReportType = 0x02 (Out), ReportID = 0x00.
- Поле **wIndex = 0x0001** — содержит номер интерфейса #1.
- Поле **wLength = 0x0001** — указывает на то, что следом прилагается один байт полезных данных.

Мы видим, что этот запрос представляет собой USB HID «SET_REPORT Request», следом за которым передается тело рапорта состоящего из одного байта. Рапорт типа «Out» для клавиатурных устройство это установка светодиодной индикации. Передаваемый в рапорте байт содержит битовый массив данных для элементов отображения, его формат определен на странице «Usage Page (LEDs)» и приведен в таблице 13.5 ниже. Установка бита в лог «1» или в «0» зажигает и гасит соответствующие битам индикаторы. Согласно таблице мы видим, что передав значение **0x00** хост желает погасить все светодиоды на клавиатуре.

Таблица 13.5. Светодиодные индикаторы для клавиатурных устройств.

Номер бита	Наименование клавиши
0	NUM LOCK
1	CAPS LOCK
2	SCROLL LOCK
3	COMPOSE
4	KANA
5	Reserved
6	Reserved
7	Reserved

3.8.9. Пример запроса к HID устройству: получение состояния клавиатуры согласно Appendix B.1

Чтобы считать состояния нажатия клавиш необходимо либо послать «GET_REPORT Request» на дефолтную конечную точку, либо, если устройство было сконфигурировано (активирована конфигурация), то достаточно (и даже необходимо) регулярно опрашивать его состояние используя «Interrupt Transfer IN». Напомню, что в этом случае хост передает токен IN адресуя его заданному устройству и конечной точке с номером, определенном в (возвращаемом) конфигурационном дескрипторе. Для широкой массы клавиатур обычно это конечная точка с номером **#1**. Устройство отвечает пакетом данных DATAx, приём которого подтверждается токеном ACK со стороны хоста.

В примерах на рис. 12.4 и 12.5 приведены осциллограммы нескольких таких опросов содержащих ответ состояния клавиатурного устройства в формате согласно **Appendix B.1**, т. е. при инициализации был установлен флаг «Boot protocol».

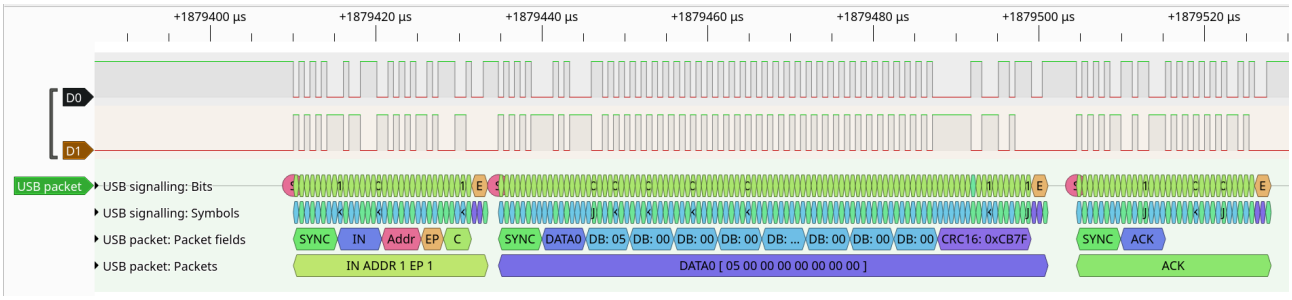

Рис. 12.4. Опрос состояния клавиатуры отправлен на ADDR:ENDP=1:1. Устройство отвечает рапортом в формате определенном Appendix B.1. Нулевой байт значением 0x05 показывает что нажаты две клавиши модификаторов.

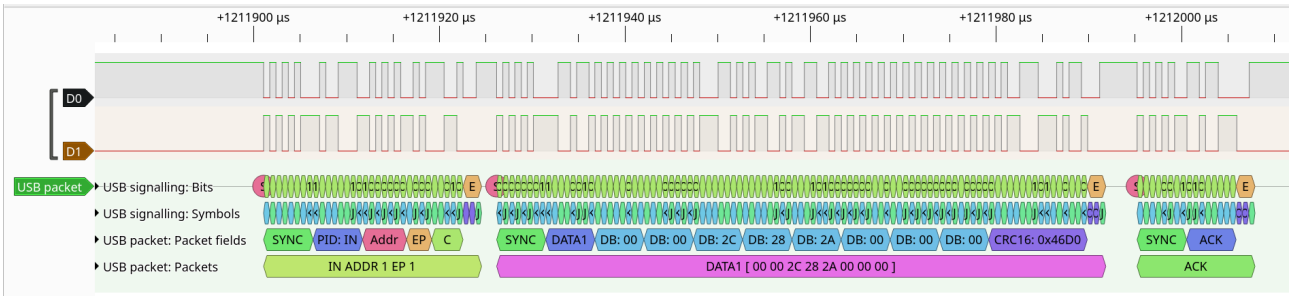

Рис. 12.5. Опрос состояния клавиатуры отправлен на ADDR:ENDP=1:1. Устройство отвечает рапортом в формате определенном Appendix B.1 — одновременно нажаты три клавиши с кодами 0x2C, 0x28 и 0x2A.

Из первой осциллограммы (рис. 12.4) видно, что в момент опроса на клавиатуре были нажаты две клавиши модификаторов, что сообщается битовой маской значением **0x05**. Если мы обратимся к таблице 13.3, то легко определим, что это клавиши **Left Ctrl** и **Left Alt**.

Из второй осциллограммы (рис. 12.5) мы видим, что все клавиши модификаторов отпущены (нулевой байт равен **0x00**), а в слотах 0, 1 и 2 находятся ненулевые значения UsageID нажатых и удерживаемых клавиш: **0x2C — Spacebar**, **0x28 — Enter** и **0x2A — Backspace**. Причем, последовательность нажатий повторяет номера слотов, т. е. были последовательно нажаты клавиши **Spacebar**, **Enter** и потом **Backspace**.

Если при опросе хостом у устройства нет данных (или устройство не готово), то в ответ на токен IN устройство выдаст токен NAK. Пример такого ответа показан на рис. 12.6 — хост в очередной раз запросил клавиатуру выдать состояние, но получил в ответ NAK. Это

не является ошибочным состоянием, а говорит о том, что хосту следует повторить попытку чуть позже.

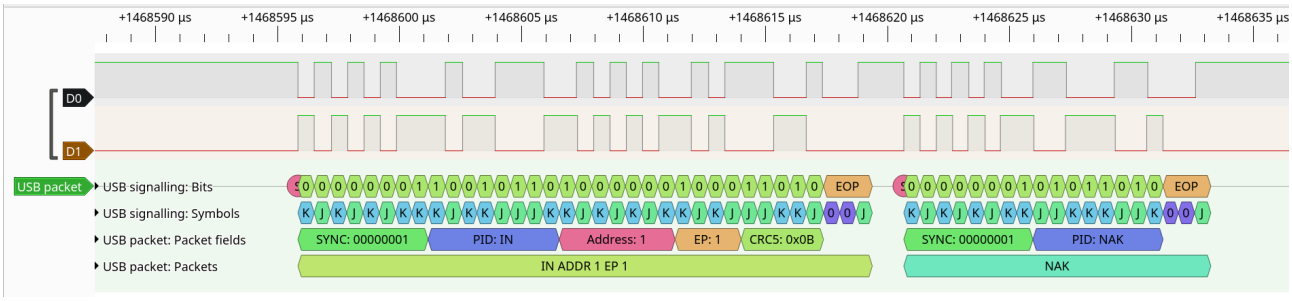

Рис. 12.6. Опрос состояния клавиатуры отправлен на ADDR:ENDP=1:1. Устройство отвечает отсутствием готовых данных (NAK).

3.8.10. Пример запроса к HID устройству: получение состояния манипулятора «мышь» согласно Appendix B.2

Для манипуляторов типа «мышь» (а также для «тачпадов» и «трекболлов») подход точно такой же — чтобы узнать состояние устройства можно либо выполнить «GET_REPORT Request» на дефолтную конечную точку, либо, если у устройства активирована конфигурация, производить регулярные опросы его состояние используя «Interrupt Transfer IN». Как и для клавиатур, на «Interrupt Transfer IN» обычно отвечает конечная точка с номером **#1**. Ниже на рис 12.7 приведен пример такого опроса.

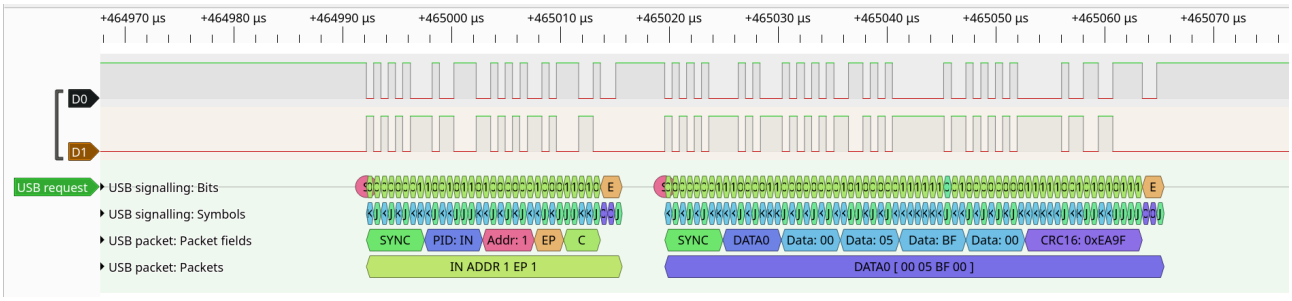

Рис. 12.7. Опрос состояния манипулятора «мышь» отправлен на ADDR:ENDP=1:1. Устройство отвечает рапортом в формате определенном Appendix B.2.

Внимательный читатель заметит, что в данной осциллограмме отсутствует токен ACK подтверждающий успешность приёма данных на стороне хоста. Судя по прошедшему интервалу времени после получения DATA0, на стороне хоста была выявлена ошибка CRC16 и хост не выслал подтверждение. Предполагается, что по истечению таймаута хост выполнит повторный опрос и получит данные этого же рапорта.

Используя таблицу 13.6 определим, что с момента последнего опроса манипулятора было произведено его физическое смещение по оси **X** на **-41** единицу (**0xBF**), по оси **Y** — на **+5** единиц (**0x05**) и ни одна из кнопок не нажата в момент опроса.

Использование сокращенного формата («Boot Protocol») - это часто распространенная практика, даже драйвер клавиатуры в ОС Linux использует его вместо сложного формата «GET_REPORT Request», который требует предварительного разбора структуры «Report Descriptor».

На этом, пожалуй, закончим с разбором форматов данных HID устройств и приступим к разработке USB контроллера.

4. Программный интерфейс USB хост-контроллера

4.1. Существующие программные интерфейсы хост-контроллеров

Аппаратное устройство обеспечивающее функцию USB хоста в составе персональной ЭВМ (напомню, что USB шина появилась как средство унификации коммуникационных интерфейсов IBM PC совместимых ЭВМ в середине 1990-х) принято называть USB хост контроллером (USB Host Controller). Для того, чтобы программа или драйвер в составе операционной системы могли взаимодействовать с хост-контроллером, необходим программно-аппаратный интерфейс — набор регистров данных, статуса и команд для доступа к шине USB и реализации её функций в программном обеспечении. Рассмотрим некоторые из известных реализаций интерфейса USB хост-контроллера.

4.1.1. Хост-контроллер UHCI

Компания Intel была первой кто озадачился вопросом программного интерфейса взаимодействия с USB хост-контроллером, её инженеры разработали и запатентовали интерфейс который получил название «Universal Host Controller Interface» (**UHCI**). Интерфейс UHCI был представлен вместе с первой версией спецификации шины USB 1.0. Изначально Intel не производила микросхемы хост-контроллеров, а решила сначала зарабатывать на лицензировании спецификации UHCI другим компаниям-производителям чипов. Первый UHCI хост-контроллер непосредственно разработки Intel появился в составе «южного моста» чип-сета Intel 82801BA (I/O Controller Hub 2) только в сентябре 2001 года.

UHCI-совместимый хост-контроллер поддерживал два режима работа: USB 1.0 «Low Speed» (1,5 Мбит/сек) и «Full Speed» (12 Мбит/сек) и обеспечивал подключение двух USB устройств (двух независимых портов). UHCI выполнял достаточно ограниченный набор функций связанных непосредственной с отправкой и приемом пакетов по шине USB, расчетом контрольных сумм, и позволял выполнять автоматический опрос шины с заданным интервалом. Реализация основной логики (машины состояний) протокола ложилась на плечи драйвера, который обязан был обеспечить обработку событий от контроллера (и хабов), обрабатывать подключение и отключение устройств, выполнять процедуру инициализации, а также реализовывать всю логику более высокого уровня.

Технически, вся спецификация интерфейса UHCI сводилась к описанию восьми отображаемых в память регистров управления и статуса, их список приведен в таблице 14.1. Драйвер UHCI устанавливал обработчик прерывания от контроллера USB, в котором он посылал контроллеру команды через регистр «USB Command», опрашивал регистры статуса и получал или передавал пакеты с данными. Передача пакетов данных осуществлялась через общие блоки памяти (буфера), список пакетов к отправке задавался в виде массива указателей на дескрипторы буферов (до 1024 шт) и занимал целую страницу памяти вычислительной системы. Адрес головы массива задавался в регистре «Frame List Base Address». UHCI работал в 32-х битном режиме.

Таблица 14.1. Регистры интерфейса USB хост-контроллера UHCI.

Смещение от базового адреса	Наименование регистра	Назначение регистра	Размерность
0x00	USBCMD	USB Command	2 bytes
0x02	USBSTS	USB Status	2 bytes
0x04	USBINTR	USB Interrupt Enable	2 bytes

0x06	FRNUM	Frame Number	2 bytes
0x08	FRBASEADD	Frame List Base Address	4 bytes
0x0C	SOFMOD	Start Of Frame Modify	1 byte
0x10	PORTSC1	Port 1 Status/Control	2 bytes
0x12	PORTSC2	Port 2 Status/Control	2 byte

Более подробную информацию об управлении USB хост-контроллером через UHCI интерфейс можно получить на сайте **OSDEV.wiki** в разделе «Universal Host Controller Interface».

4.1.2. Хост-контроллер OHCI

Монополия Intel на интерфейс USB хост-контроллера длилась недолго. Через пару лет, в 1999 году, по инициативе компании Compaq, при поддержке National Semiconductor и Microsoft был разработан и опубликован открытый стандарт который получил название «Open Host Controller Interface for USB» или сокращенно **OHCI**. Разумеется, появление открытого стандарта заставило руководство Intel переосмыслить решение по взиманию денег и опубликовать спецификацию UHCI как открытый стандарт.

OHCI интерфейс предлагал пользователю (программисту создающему драйвер) оперировать потоками на уровне конечных точек (endpoints), таким образом возлагая на аппаратуру хост-контроллера всю сложную логику по обеспечению целостности транзакций и разграничения потоков данных. Помимо этого, OHCI совместимый хост контроллер должен брать на себя все функции по инициализации устройств, обеспечению менеджмента пропускной способности шины и приоритизации трафика разных видов. Напомню, что в USB «Interrupt Transfer» имеет наивысший приоритет, так как несет в себе информацию о прерываниях формируемых устройством. Еще одним важным нововведением OHCI было то, что согласно спецификации, OHCI совместимый хост-контроллер обязан был иметь уже встроенный корневой USB хаб («USB root hub»). Очевидно, что реализовать такую сложную машину состояний с помощью только аппаратной (RTL) логики задача крайне непростая, поэтому в хост-контроллерах OHCI появилось программируемое микропроцессорное устройство — своя миниатюрная ЭВМ со своей программой («прошивкой» или «firmware»).

Технически, OHCI интерфейс представляет драйверу набор из 21-го отображаемого в память 32-х битного регистра управления и разделяемой области оперативной памяти для хранения связанного списка буферов. Сам хост-контроллер представляет собой периферийное устройство в составе IBM PC совместимой машины подключенное к шине Peripheral Component Interconnect (PCI). В рамках PCI, контроллеру OHCI присваивается **Class ID = 0x0C**, **subClass ID = 0x03** и **Interface ID = 0x10**. Драйвер должен найти на PCI шине такое устройство и считать регистр базового адреса (**BAR0**), содержимое которого указывает на область памяти отображаемых регистров управления. Перечень регистров OHCI интерфейса приведен в таблице 14.2. Так же как UHCI, интерфейс OHCI оперирует в 32-х битном режиме.

Таблица 14.2. Регистры интерфейса USB хост-контроллера OHCI.

Смещение от базового адреса	Наименование регистра	Назначение регистра	Размерность
0x00	HcRevision	Хранит номер версии спецификации которой соответствует хост-контроллер в формате BCD (0x11 для версии 1.1).	4 bytes

0x04	HcControl	Определяет режим работы хост-контроллера. Значащие биты [11:0].	4 bytes
0x08	HcCommandStatus	Регистр для оправки команд в хост-контроллер	4 bytes
0x0C	HcInterruptStatus	Содержит битовые поля индицирующие возникновение различных событий (прерываний).	4 bytes
0x10	HcInterruptEnable	Содержит битовые маски для разрешение прерываний от соответствующих источников событий (запись «1» разрешает).	4 bytes
0x14	HcInterruptDisable	Содержит битовые маски для запрета прерываний (запись «1» запрещает).	4 bytes
0x18	HcHCCA	Содержит (задает) физический адрес разделяемого блока памяти «Host Controller Communication Area». Этот блок содержит списки буферов с данными и прочие управляющие структуры (Endpoint Descriptors).	4 bytes
0x1C	HcPeriodCurrentED	Содержит физический адрес текущего «Endpoint Descriptor» (ED) для «Isochronous Transfer» или «Interrupt Transfer».	4 bytes
0x20	HcControlHeadED	Содержит физический адрес головного «Endpoint Descriptor» в списке Control List.	4 bytes
0x24	HcControlCurrentED	Содержит физический адрес текущего «Endpoint Descriptor» используемого для обхода списка Control List.	4 bytes
0x28	HcBulkHeadED	Содержит физический адрес головного «Endpoint Descriptor» в списке Bulk List.	4 bytes
0x2C	HcBulkCurrentED	Содержит физический адрес текущего «Endpoint Descriptor» используемого для обхода списка Bulk List.	4 bytes
0x30	HcDoneHead	Содержит физический адрес первого «Transfer Descriptor» в списке Done Queue.	4 bytes
0x34	HcFmInterval	Младшие 14 бит задают время между двумя последовательными SOF. Еще 15 бит задают максимальный размер пакета которым будет оперировать контроллер в режиме «Full Speed».	4 bytes
0x38	HcFmRemaining	Непрерывно уменьшающийся счетчик времени до следующего SOF.	4 bytes
0x3C	HcFmNumber	16 битный счетчик используемый как референс для формирования номера	4 bytes

		фрейма в SOF.	
0x40	HcPeriodicStart	14 битный счетчик задает интервал времени с которым контроллер обрабатывает периодические список.	4 bytes
0x44	HcLSThreshold	Служебный регистр для «Low Speed» режима.	4 bytes
0x48	HcRhDescriptorA	Первый регистр описывающий характеристики Root Hub.	4 bytes
0x4C	HcRhDescriptorB	Второй регистр описывающий характеристики Root Hub.	4 bytes
0x50	HcRhStatus	Содержит биты состояния и биты управления для Root Hub.	4 bytes

Помимо приведенных выше регистров, начиная со смещения 0x54 добавляется по одному регистру для каждого из портов Root Hub-a, эти регистры называют **HcRhPortStatus[1:NDP]**, где NDP — число портов корневого хаба, определяется полем **NumberDownstreamPorts** в регистре характеристик хаба **HcRhDescriptorA**. Каждый такой регистр содержит битовые поля для считывания статуса порта или управления им. С более подробным описанием OHCI интерфейса также можно ознакомиться на сайте **OSDEV.wiki** в разделе «Open_Host_Controller_Interface».

Исходя из приведенного в таблице описания регистров управления OHCI можно заметить, что всё управление хост-контроллером сводится к формированию списков с дескрипторами конечных точек (Endpoint Descriptors) и своевременной манипуляцией очередями. Такой подход существенно облегчает задачи драйвера перекладывая большую часть работы на аппаратуру контроллера.

4.1.3. Другие хост-контроллеры: EHCI и xHCI

С принятием спецификации USB версии 2.0 появился и соответствующий ей интерфейс, он получил название «Enhenced Host Controller Interface» (EHCI). Не смотря на то, что хост-контроллеры согласно спецификации USB 2.0 были «вниз совместимы» и поддерживали работу с устройствами USB 1.0, интерфейс EHCI не обязан обеспечивать работу с USB 1.0. В связи с этим, каждый хост-контроллер USB 2.0 дополнительно поддерживает UHCI или OHCI интерфейсы, или оба сразу. Для EHCI также используется PCI Class ID = 0x0C и subClass ID = 0x03, но Interface ID = **0x20**.

Интерфейс EHCI по своему функционалу похож на своего младшего собрата OHCI, он также оперирует потоками данных между конечными точками, а всё программирование сводится к составлению списков с дескрипторами и отслеживанию событий (прерываний) от контроллера. Данный вид интерфейса для USB хост-контроллера является самым распространенным, в том числе он часто используется в простых микроконтроллерных устройствах.

С приходом стандарта на шину USB версии 3.0 была принята спецификация нового универсального интерфейса - «eXtensible Host Controller Interface» (xHCI). Данный вид интерфейса поддерживает работу со всеми предыдущими версиями шины, то есть USB 1.x, 2.x и 3.x, без необходимости в дополнительном «компаньоне». Для xHCI установлены PCI Class ID = 0x0C, subClass ID = 0x03 и Interface ID = **0x30**. Данный интерфейс является 64-х битным, поэтому базовый адрес расположения таблицы регистров в памяти получается

конкатенацией двух 32-х битных слов содержащихся в конфигурационных регистрах **BAR1** и **BAR0** шины PCI.

Несмотря на то, что каждый новый интерфейс перекладывал всё больше функций на аппаратуру (на хост-контроллер), сложность взаимодействия с контроллером постоянно возрастала, а объем кода драйвера для соответствующего интерфейса существенно увеличивался. Это несложно проследить по исходным кодам ядра ОС Linux. Для этого в корневом каталоге репозитория с исходными кодами ядра введем следующие команды:

```
rz@devbox:~/linux-kernels/linux-5.0-truncated$ cat drivers/usb/host/uhci* | wc
   5210   18552  142434

rz@devbox:~/linux-kernels/linux-5.0-truncated$ cat drivers/usb/host/ohci* | wc
  11582   35746  345118

rz@devbox:~/linux-kernels/linux-5.0-truncated$ cat drivers/usb/host/ehci* | wc
  15637   49897  471188

rz@devbox:~/linux-kernels/linux-5.0-truncated$ cat drivers/usb/host/xhci* | wc
  27952   92787  796012
```

Команда **wc** выдает три числа: количество строк, количество слов и количество байт текста.

Парадоксально, но UHCI имеет самую маленькую кодовую базу даже при том, что он выполняет гораздо больше работы по менеджменту шины. Объясняется это простотой принятых концепций.

xHCI интерфейс почти вытеснил все остальные виды интерфейсов на рынке ПК, но смею предположить, что UHCI/OHCI и EHCI еще очень долго будут с нами в сфере «embedded» и IoT, где простота реализации имеет немаловажное значение!

4.2. Концепция интерфейса реализуемого хост-контроллера

Очевидно, что реализовывать хост-контроллер с поддержкой интерфейсов типа EHCI и xHCI мы не станем по причине того, что эти интерфейсы предназначены для версий USB 2.x и 3.x, что выходит за рамки поставленной задачи. Остается два варианта контроллеров: с «UHCI-подобным» и «OHCI-подобным» интерфейсом. Как было сказано выше, UHCI реализует только базовые функции передачи единичного пакета и некоторый менеджмент пропускной способности, в то время как OHCI берет на себя задачи по организации потоков данных между конечными точками. Хост-контроллер с интерфейсом типа OHCI более привлекателен с точки зрения программиста, так как позволяет переложить на аппаратуру бо́льший пласт работ. В тоже время OHCI контроллер имеет существенно более сложное устройство — он содержит отдельный программируемый вычислитель который требует разработки специализированного программного обеспечения («firmware») для него. Оценив всю сложность контроллера с OHCI и приняв во внимание ограниченность ресурсов микросхемы ПЛИС установленной на плате «Карно», я принял решение не углубляться в разработку еще одного вычислительного ядра, а возложить все функции управления потоками на центральный процессор, оставив при этом самый минимум аппаратуры реализующей следующие задачи:

1. Детектирование подключения и отключения устройства.
2. Автомат выполнение сброса шины.
3. Автомат регулярного формирования сигнала «KeepAlive».
4. Автомат передачи коротких токенов (ACK, NAK, STALL).

5. Автомат передачи длинных токенов (SETUP, IN, OUT) с аппаратным расчетом CRC5.
6. Автомат передачи пакета данных (DATA0, DATA1) с аппаратным расчетом CRC16.
7. Автомат приёма блока данных произвольной длины (токена или пакета), с аппаратным расчетом CRC16 для получаемого блока данных, и формирования флага корректности принятых данных.

На программную часть (низкоуровневый драйвер), которая подлежит реализации на центральном вычислительном ядре архитектуры RV32IMAC, возлагаются следующие задачи:

1. Мониторинг состояния шины, запуск процедуры инициализации в случае обнаружения устройства.
2. Выполнение одной транзакции с конечной точкой.
3. Выполнение комплекса транзакций при исполнении произвольного запроса.
4. Выполнение процедуры инициализации устройства (комплекса запросов), получение и сохранение дескрипторов.
5. Регулярный опрос устройства транзакцией «Interrupt IN» с заданным интервалом времени.
6. Менеджмент шины — контроль пропускной способности и отбираемой мощности (тока).

Сразу сделаю замечание по поводу менеджмента шины. Так как в рамках данной задачи мы ориентируемся только на HID устройства и без поддержки хабов, то контролем за заполнение шины трафиком можно смело пренебречь, потому как такие устройства не могут создавать большой объем передаваемых по шине данных. Аналогично обстоят дела с электропитанием — большинство HID устройств потребляют от шины ток до 100 мА, что ниже лимита в 500мА, а значит этот вопрос тоже можно отложить «на потом».

Программный интерфейс разрабатываемого нами USB хост-контроллера будет сильно упрощенным вариантом интеловского UHCI предназначенным только для USB 1.0 «Low Speed». Так как максимальный размер полезной нагрузки в пакете DATAx для этой версии USB протокола не превышает 8 байт, то обмен пользовательскими данными будет реализован через два 32-х битных регистра без FIFO буферизации. У контроллера будет два набора регистров данных — один для передачи, другой для приема. Контроллер будет помещать в «регистр принятых данных» принимаемый от устройства блок данных и устанавливать флаги: «пакет принят» и «валидность CRC принятого пакета». При необходимости отправить блок данных, драйвер должен будет записать в «регистр отправляемых данных» передаваемый блок данных и выполнить команду «передать блок данных». После принятия команды, контроллер установит флаг «занят», а в случае успешной отправки пакета — взведет флаг «репорт». Флаг «пакет принят» также будет взводить флаг «репорт». При установке флага «репорт», контроллер будет формировать аппаратное прерывание по соответствующей линии. Контроллер будет поддерживать команды «отправить длинный токен» (содержит CRC5), «отправить короткий токен» (без CRC) и «сброс шины».

Помимо низкоуровневого драйвера работающего с шиной USB, нам придется реализовать драйвер HID устройств («клавиатура», «мышь» и «геймпад»), а также пользовательское приложение обработки потока входящих данных.

4.3. Описание регистров и команд

Первым делом опишем структуру с регистрами разрабатываемого хост-контроллера. Ниже в таблице 15.1 приведен перечень из 9-ти аппаратных регистров статуса и управления, по 32 бита каждый. Все регистры будут доступны на чтение и запись.

Хост-контроллером будут поддерживаться четыре основные команды приведенные таблице 15.2. Хост-контроллер будет оперировать 8-ю обязательными токенами приведенными в таблице 15.3.

Таблица 15.1. Описание регистров управления хост-контроллера USB 1.0

Смещение от базового адреса	Наименование регистра	Назначение регистра	Размерность
0x00	STATUS	Содержит флаги и код машины состояния (FSM): Бит 30 — ERROR. Если «1», то произошла ошибка на шине (устройство отключено). Если «0» - устройство подключено и контроллер готов к работе. Бит 29 — REPORT. Если «1», то есть прерывание от контроллера. Обработчик должен сбросить этот флаг по завершению. Бит 28 — BUSY. Если «1», контроллер занять исполнением команды или приемом данных. Бит 27 — RECEIVED. Если «1», контроллер принял пакет и данные присутствуют в регистре принятых данных. Бит 26 — CRC16_OK. Валидность CRC принятого пакета. Если «1», то в принятом пакете нет ошибок CRC16. Биты [7:0] — STATE. Код машины состояния (FSM).	32 бита
0x04	COMMAND	Регистр команд: Бит 31 — установкой в «1» запускается выполнение команды. Биты [30:24] — Адрес устройства (поле ADDR) для передачи. Биты [23:20] — Номер конечной точки (поле ENDP) для передачи. Биты [19:8] — LEN. Длина передаваемого блока данных в битах. Биты [7:4] — PID. Поле Packet ID для	32 бита

		передачи. Биты [3:0] — CMD. Содержит код команды.	
0x08	RECV_DATA_LOW	Регистр принятых данных, младшие 32 бита.	32 бита
0x0C	RECV_DATA_HIGH	Регистр принятых данных, старшие 32 бита.	32 бита
0x10	SEND_DATA_LOW	Регистр передаваемых данных, младшие 32 бита.	32 бита
0x14	SEND_DATA_HIGH	Регистр передаваемых данных, старшие 32 бита.	32 бита
0x18	RX_STATUS	Первый регистр статуса приемника: Биты [7:0] — LEN. Длина (в битах) принятого пакета, включая CRC16. Биты [15:8] — PID. Идентификатор Packet ID принятого пакета. Биты [31:16] — CRC16 принятого пакета переданное устройством.	32 бита
0x1C	CONTROL	Регистр управления: Бит 31 — ENABLE. Разрешает работу контроллера если «1». Бит 30 — KEEPALIVE — Разрешает регулярную отправку состояния «Keepalive» если «1». Биты [15:0] — RESET_DELAY. Длительность сигнала «сброс» в тактах шины.	32 бита
0x20	RX_STATUS2	Второй регистр статуса приемника: Биты [15:0] — CALCULATED_CRC16. Содержат вычисленное значение CRC16 для принимаемого пакета. Данное значение сравнивается со значением RX_STATUS.CRC16 и оба значения совпадают, то устанавливается флаг STATUS.CRC16_OK.	32 бита

Таблица 15.2. Описание команд хост-контроллера USB 1.0

Наименование команды	Код команды	Назначение
NOP	0x00	Нет операции. Используется для отладки.
SEND_LONG_TOKEN	0x01	Отправить по шине длинный токен содержащий CRC5 (SETUP, IN, OUT).
SEND_SHORT_TOKEN	0x02	Отправить по шине короткий токен без CRC (ACK, NAK, STALL).
SEND_DATA	0x03	Отправить по шине пакет данных (DATA0, DATA1).
BUS_RESET	0x04	Выполнить процедуру сброса шины.

Таблица 15.3. Типы пакетов (токенов) поддерживаемые хост-контроллером USB 1.0

Наименование токена	Значение (8 бит)	Назначение
SETUP	0b00101101	Установка адреса и отправка запрос к устройству.
DATA0	0b11000011	Пакет с данными DATA0.
DATA1	0b01001011	Пакет с данными DATA1.
IN	0b01101001	Установка адреса для передачи данных от устройства к хосту.
OUT	0b11100001	Установка адреса для передачи данных от хосту к устройству.
ACK	0b11010010	Подтверждение безошибочного получения пакета с данными.
NAK	0b01011010	Признак того, что пакет с данными/запросом не принят, требуется перепосылка.
STALL	0b00011110	Передача невозможна, возникла ошибка.

4.4. Описание состояний хост-контроллера

Основной автомат хост-контроллера, взаимодействие с которым осуществляет драйвер, будет иметь восемь состояний, их описание приведено в таблице 15.4. Состояние **StateWaitCMDorSYNC** является нормальным состоянием контроллера при подключенном устройстве и отсутствии активности на шине. Если на шине отсутствуют устройства, то контроллер переходит в состояние **StateUnconnected** и устанавливает бит **ERROR** в регистре статуса **STATUS**. Остальные состояния контроллера являются временными, переход в них выполняется краткосрочно на момент исполнения команды или приема блока данных от устройства.

Таблица 15.4. Коды состояний хост-контроллера USB 1.0

Наименование состояния	Код	Описание
StateUnconnected	0	Нет подключений к шине, обе линии D+/D- в «LOW».
StateWaitCMDorSYNC	1	Контроллер готов. Ожидает либо команды от приложения, либо сигнала SYNC под шина для начала приема пакета.
StateKeepAlive	2	Контроллер выполняет функцию «Keepalive».

StateSendLongToken	3	Контроллер находится в процессе отправки длинного (CRC5) токена.
StateSendShortToken	4	Контроллер находится в процессе отправки короткого (без CRC) токена.
StateSendData	5	Контроллер находится в процессе отправки пакета данных (отправляется токен с CRC16).
StateSendReset	6	Контроллер выполняет процедуру сброса шины.
StateReceive	7	Контроллер находится в состоянии приема данных.

На этом описание программного интерфейса USB хост-контроллера заканчивается. Детали и тонкости его использования мы рассмотрим в главе 6. «*Реализация драйвера USB хост-контроллера*», а сейчас перейдем к аппаратной реализации.

5. Аппаратная реализация USB хост-контроллера

5.1. Структура конечного автомата USB хост-контроллера

В хост-контроллере реализуются следующие конечные автоматы:

1. Основная машина состояний — **USBMain**. Занимается оркестрацией нескольких вспомогательны машин и контролем состояния шины USB.
2. Вспомогательная машина передачи короткого токена без CRC — **USBSendShortToken**. Передает по шине 16 бит: 8 бит преамбулы и 4 бита PID + его инверсное значение.
3. Вспомогательная машина передачи длинного токена и CRC5 — **USBSendLongToken**. Передает по шине 32 бита: 8 бит преамбулы, 4+4 бита PID, 7 бит адреса устройства (ADDR), 4 бита номера конечной точки (ENDP) и 5 бит CRC5 рассчитанного из передаваемого значения ADDR и ENDP.
4. Вспомогательная машина передачи пакета данных — **USBSendData**. Передает 8 бит преамбулы, 4+4 бита PID, произвольное (до 64) количество бит данных и 16 бит рассчитанного значение кода CRC16.
5. Вспомогательная машина приема блока данных — **USBReceiver**. Принимает пакет данных от устройства и попутно вычисляет код CRC16 по получаемым данным.
6. Вспомогательная машина выполнения процедуры «KeepAlive».
7. Вспомогательная машина выполнения сброса (Bus Reset).

Ниже на рис. 13.1 представлена диаграмма переходов состояний для основного конечного автомата (основной машины состояний) USB хост-контроллера реализуемого в рамках данного проекта и его связи со вспомогательными автоматами. На рис. 13.2-13.6 представлены диаграммы переходов состояний для вспомогательных автоматов.

В представленных далее диаграммах переходов состояний используется следующая нотация:

- **Скругленный прямоугольник** представляет одно из состояний автомата. В верхней части фигуры дано кодовое название состояния. В нижней дано описание инициализации (действия при переходе КА в данное состояние) на формальном языке.

- **Ромбовидная фигура** представляет собой комбинационную схему кодирующую условие. Выполнение этого условия или установка указанного в фигуре флага приводит к активации следующего участка схемы автомата. Стрелка «вниз влево» показывает переход если условие выполняется, «вниз вправо» - не выполняется.

- **Прямоугольник** задает действие которое происходит при активном входном условии. В верхней части дается название данному действию, в нижней — описание действия на формальном языке. Может порождать как комбинационную, так и последовательностную схему.

Примечание: представленные ниже диаграммы подготовлены с помощью утилиты **GraphViz**, их исходный код на языке **.dot** находится в репозитории в каталоге ./doc/graphs/USB10/.

5.1.1. Основной конечный автомат USBMain

Если к шине USB не подключено устройство, то основная машина **USBMain** пребывает в состоянии **StateUnconnected**, куда она попадает по сбросу или при возникновения ошибки на шине. Если к шине подключается устройство, то машина **USBMain** переходит в состояние **StateWaitCMDorSYNC**.

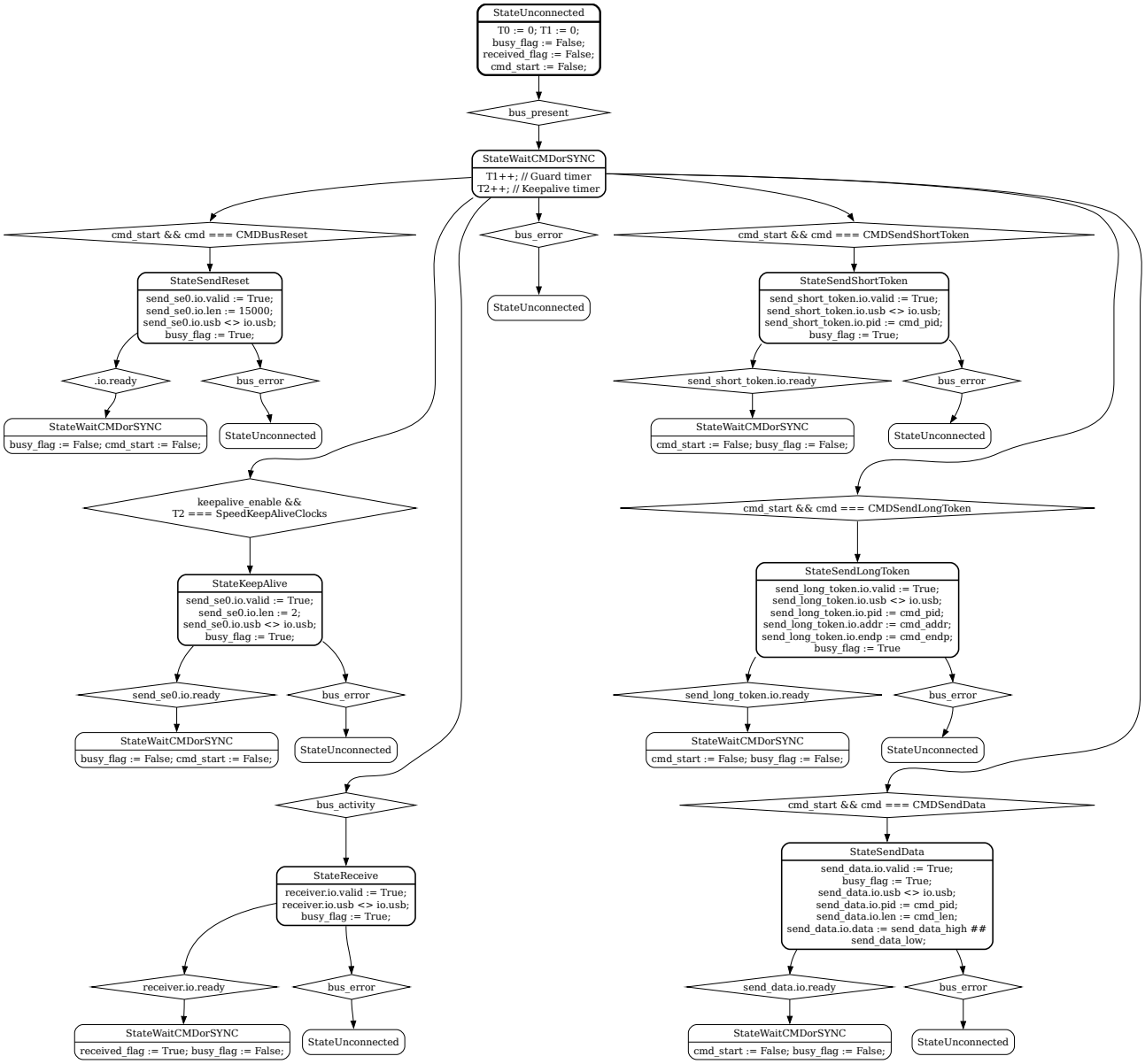

Рис. 13.1 Диаграмма переходов состояний основного конечного автомата USBMain.

Находясь в своем основном состоянии **StateWaitCMDorSYNC** машина непрерывно анализирует состояние битового флага **cmd_start** регистра команда COMMAND. Если флаг **cmd_start** установлен, то в зависимости от кода команды находящейся в 4-х битовом поле **cmd** этого же регистра производится смена состояния основной машины на одно из: **StateSendReset**, **StateSendShortToken**, **StateSendLongToken** или **StateSendData,** при этом активируется одна из вспомогательных машин установкой сигнала **io.valid** на её входе и производится коммутация шины USB - комплексный сигнал **io.usb.usb** основного КА, содержащий **dm** и **dp**, связывается с аналогичным комплексным сигналом вспомогательного

КА (оператор «<>» на диаграмме). Вместе с этим на вход вспомогательной машины подаются требуемые для её работы входные данные: Packet ID (**cmd_pid**), размер пакета (**cmd_len**), данные для передачи (**send_data_high** и **send_data_low**), адрес устройства (**cmd_addr**) и номер конечной точки (**cmd_endp**).

Вспомогательные машины выполняют свои задания и индицируют окончание работы в основную машину с помощью сигнала **ready**. Снятие сигнала **valid** со стороны основной машины переводит вспомогательную машину в начальное состояние, при этом сигнал **ready** сбрасывается внутри вспомогательных машин.

Переход основной машины в состояние **StateReceive** производится при срабатывании флага **bus_activity**, который активируется если был зафиксирован переход шины USB из пассивного состояния «J» в состояние «K», что рассматривается как признак входящего пакета данных. В состоянии **StateReceive** активируется вспомогательная машина **USBReceiver**. Машина **USBReceiver** полностью принимает входящий пакет данных вместе с блоком CRC16. В процессе приема машина **USBReceiver** ведет расчет своего внутреннего значения CRC16 по принятым данным. По завершению приема машина **USBReceiver** помещает в выходной буфер принятый пакет данных, выдает рассчитанный ей код CRC16 и активирует выходной флаг **ready**. Основная машина вычисляет флаг валидности принятого пакета **crc16_ok** сравнивая принятый код CRC16 с рассчитанным, переносит полученные данные и флаги в регистры доступные пользователю (RECV_DATA_LOW, RECV_DATA_HIGH, STATUS и RX_STATUS) и возвращается в состояние **StateWaitCMDorSYNC**.

Переход основной машины в состояние **StateKeepAlive** производится по таймеру **T2**. В этом состоянии активируется сигнал **valid** у вспомогательной машины которая кратковременно (на два битовых интервала) переводит шину USB в состояние «SE0», что сигнализирует устройствам на шине об активности хоста. Выход из этого состояния также производится по сигналу готовности **ready** от вспомогательной машины.

5.1.2. Флаги основного конечного автомата USBMain

Основная машина состояний содержит несколько флагов индицирующих её текущее состояние и доступные пользователю на программном уровне.

Флаг **busy_flag** устанавливается каждый раз когда основная машина переходит в состояние отличное от **StateWaitCMDorSYNC**, то есть активируется один из вспомогательных автоматов. Флаг **busy_flag** сбрасывается когда основная машина возвращается в состояние **StateWaitCMDorSYNC**. Пользователь может проверить этот флаг перед тем как отправить команду в хост-контроллер чтобы убедиться, в том, что контроллер не занят какой-то работой (например отправкой KeepAlive или приемом пакета данных).

Флаг **cmd_start** устанавливается пользователем программно через бит 31 регистра команд COMMAND и сбрасывается основной машиной после завершение выполнения команды. Пользователь может использовать данный бит регистра COMMAND для ожидания завершения выполнения команды.

Флаг **received_flag** сбрасывается когда основная машины входит в состояние приема пакета (**StateReceive**) и устанавливается после завершения работы вспомогательного автомата **USBReceiver**. Этот флаг отображается в бит 27 регистра статуса STATUS и доступен пользователю. Пользователь может опрашивать этот бит регистра статуса чтобы определить факт получения пакета.

На изменения любого из этих флагов может быть подключен триггер вызывающий сработку контроллера прерываний.

Основная машина **USBMain** содержит несколько внутренних флагов генерируемых в каждый такт и недоступных пользователю:

Флаг **bus_error** устанавливается если шина USB находится в состоянии «SE0». Данный флаг используется для перевода машины в состояние **StateUnconnected** по достижению защитного таймера **T1** предельного значения.

Флаг **bus_present** устанавливается если шина USB находится в пассивном («J») состоянии. Используется для перевода машины в состояние **StateWaitCMDorSYNC** из состояния **StateUnconnected**.

Флаг **bus_activity** устанавливается если состояние шины изменяется на «K», что рассматривается как признак активности на шине (попытка устройства инициировать передачу данных). Используется для перевода машины в состояние **StateReceive** из состояния **StateWaitCMDorSYNC**.

5.1.3. Вспомогательный конечный автомат USBSendSE0

Отправку сигналов «Bus Reset» и «KeepAlive» можно реализовать всего одной примитивной машиной состояний содержащей два последовательных счетчика. Первый (**clock_div**) для формирования сигнала строба **clock_strobe** следующего с временными интервалами равными одному битовому интервалу на шине (длительность одного бита). Второй счетчик — **bit_count**, для подсчета числа битов. Назовем такой конечный автомат **USBSendSE0** так как его задача переводить шину USB в состояние «SE0» на заданное количество битовых интервалов. При активации этот автомат будет переводить шину USB в состояние «SE0» и по завершению счета битов возвращать её в состояние «J» (пассивное состояние шины).

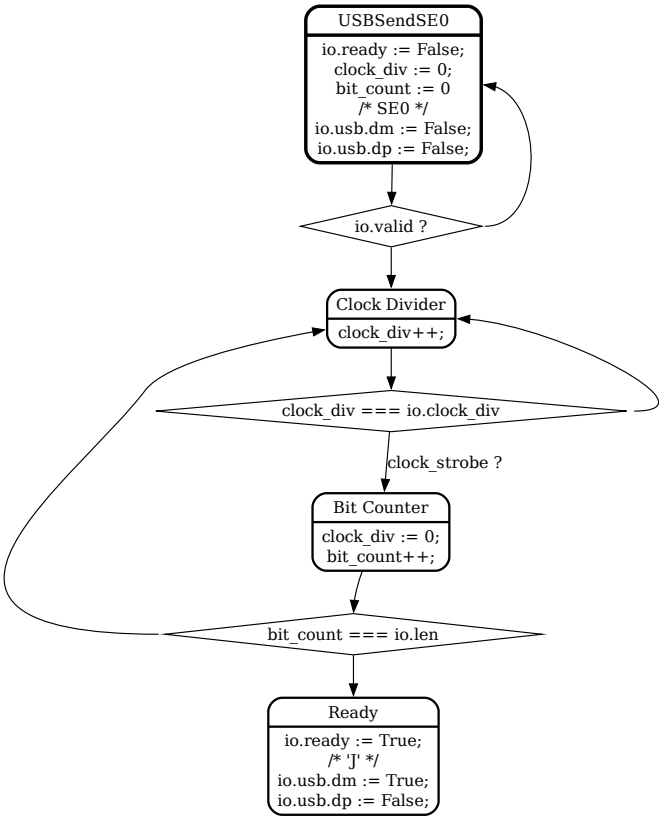

Рис. 13.2 Диаграмма переходов состояний вспомогательного автомата USBSendSE0 (используется для «Bus Reset» и «KeepAlive»).

Диаграмма состояний для машина **USBSendSE0** приведена на рис. 13.2. Эта машина будет активироваться из основной машины состояний **USBMain** в двух случаях: в состоянии **StateSendReset** для формирования «Сброса» на длительность эквивалентную 10 мс (15000 отсчетов **bit_count** для скорости 1,5Мбит/сек USB 1.0 «Low Speed»); и в состоянии **StateKeepAlive** для посылки сигнала «KeepAlive» длительность которого, согласно спецификации, составляет 2 битовых интервала.

5.1.4. Вспомогательный конечный автомат USBSendShortToken

Задача машины состояний **USBSendShortToken**, диаграмма переходов состояний которой представлена на рис. 13.3, сводится к передаче по шине короткого пакета (токена) длиной всего 16 бит: 8 бит SYNC (преамбула) + 4 бита PID + 4 бита инвертированный PID. Идентификатор пакета (PID) поступает на ход по 4-х битовому сигналу **io.pid** вместе с сигналом активации **io.valid**. Из входных данных формируется внутренний буфер **buffer** путем конкатенации PID и битового представления преамбулы, из этого буфера изымаются данные для передачи (синтаксическая конструкция ## означает «битовая конкатенация»). По завершению работы машина устанавливает выходной сигнал **io.ready** и остается в таком состоянии до сброса входного **io.valid**.

Состояние машины **USBSendShortToken** представляется одним регистром — счетчиком переданных битов данных **bit_count**. Счетчик приращивается на единицу с каждым переданным битом данных. В процессе передачи бита данных производится его кодирование в символ «J» (представляется как **usb.dm := True**, **usb.dp := False**) или «K» (**usb.dm := False**, **usb.dp := True**) в зависимости от того, какой предыдущий символ был передан. Для сохранения предыдущего переданного по шине символа используется одно-битовый регистр **last_kj**. Процедура «бит-стаффинга» в данной машине не осуществляется, так как значения PID подобраны таким образом, чтобы исключить появление на шине шести последовательных единиц.

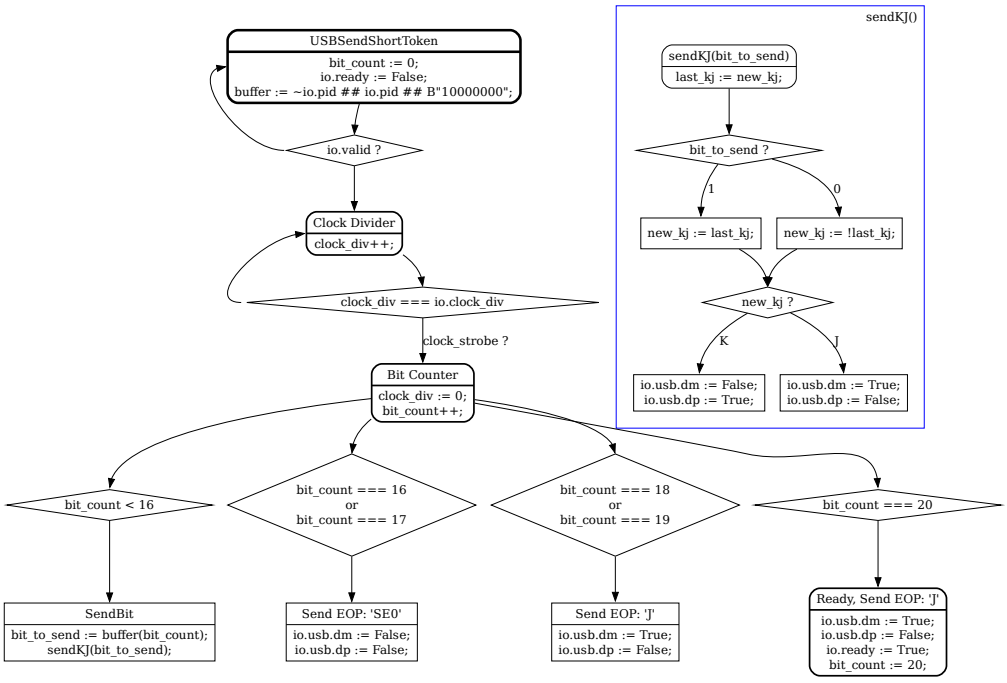

Рис. 13.3 Диаграмма переходов состояний вспомогательного автомата USBSendShortToken.

Когда счетчик **bit_count** достигает значения 16, машина **USBSendShortToken** передает по шине USB сначала два символа «SE0», а следом два символа «J». Эта последовательность является признаком конца передачи пакета (EOP). Таким образом передав 20 символов машина заканчивает работу и поднимает сигнал **io.ready**.

Процесс передачи данных представлен на диаграмме функцией **sendKJ()**, она принимает на вход значение текучего передаваемого бита данных и производит все перечисленные преобразования в символ. Данная функция порождает комбинационную и последовательностную схему и может рассматриваться как вложенный КА.

5.1.5. Вспомогательный конечный автомат USBSendLongToken

Функционирование машины состояний **USBSendLongToken,** диаграмма которой представлена на рис. 13.4, во многом схоже с машиной **USBSendShortToken**. Различие состоит в том, что длина передаваемого блока данных составляет 32 бита и включает дополнительные поля: **io.addr** — 7 бит адреса устройства (ADDR), **io.endp** — 4 бита номера конечной точки (ENDP) и вычисляемого значения **crc5_out** длиной 5 бит. Значение **crc5_out** может быть рассчитано по формуле представленной на диаграмме функцией **calc_crc5_usb()** порождающей комбинационную схему. Внутренний буфер **buffer**, из которого изымаются данные для передачи, аналогично формируется путем конкатенации входных данных в одно большое 32-х битное слово.

В машине **USBSendLongToken** также используется регистр-счетчик **bit_count** в качестве внутреннего регистра состояний и подсчета переданных символов. При достижении **bit_count** значения 32 происходит последовательная передача двух символов «SE0», затем двух символов «J» и подъем флага готовности **io.ready**.

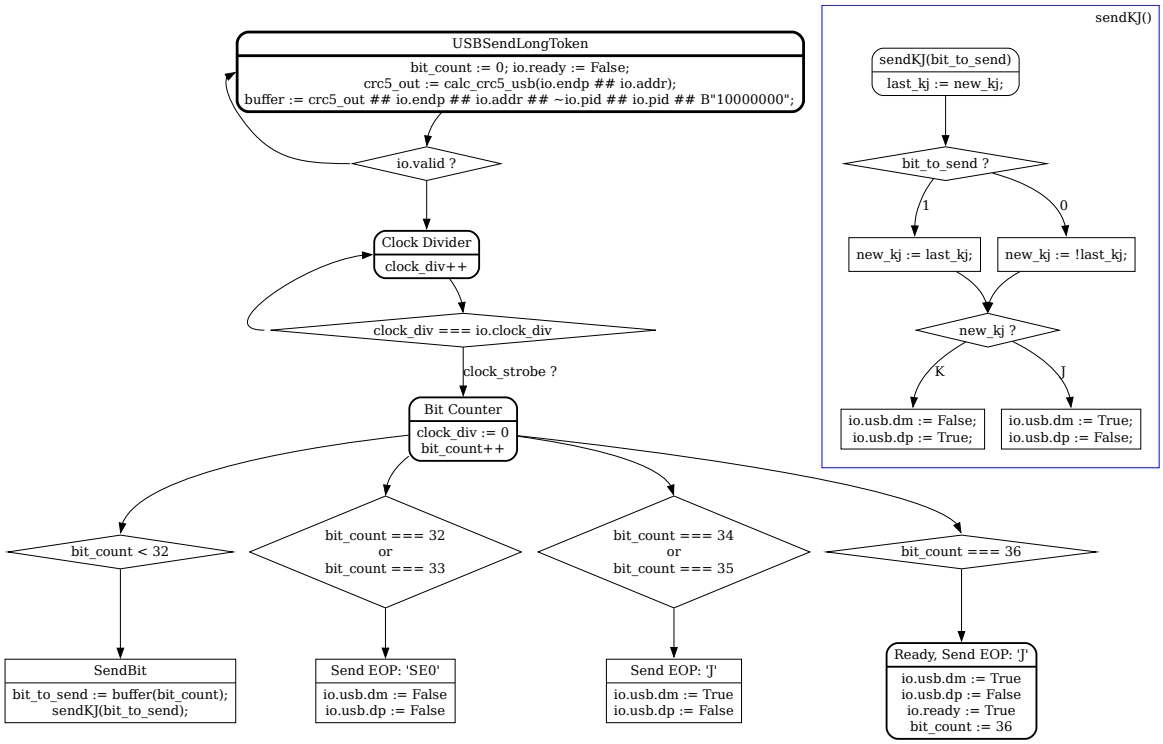

Рис. 13.4 Диаграмма переходов состояний вспомогательного автомата USBSendLongToken.

5.1.6. Вспомогательный конечный автомат USBSendData

Машина состояний **USBSendData** решает задачу отправки по шине USB пакета данных произвольной длины. Напомню, что для USB 1.0 размер полезной нагрузки ограничен 64 битами. На вход данная машина принимает следующие сигналы: **io.pid** — 4 бита идентификатор PacketID, **io.data** — 64 бита полезных данных, **io.len** — 6 бит длина полезной нагрузки в битах, и **io.valid** — разрешающий сигнал.

Функционирование машины состояний **USBSendData** во многом похоже на работу предыдущих двух, но есть ряд серьезных отличий. Во-первых, процесс передачи пакета разделяется на четыре фазы: фаза передачи преамбулы и PID, фаза передачи полезных данных и вычисления CRC16, фаза передачи рассчитанного кода CRC16 и фаза передачи завершающей последовательности EOP.

Во-вторых, в виду переменной длины блока данных рассчитать CRC16 сразу на весь пакет не представляется возможным, поэтому процесс вычисления контрольной суммы выполняется блоками по 8 бит. В процессе передачи биты данных задвигаются во внутренний регистр **crc_byte** и каждый 8-й бит активируется расчет. Рассчитанный код CRC16 помещается во внутренний регистр **crc16_buf**.

И в-третьих, в процессе кодирования данных (преобразования в символы) требуется осуществлять процедуру «бит-стаффинга», т. е. после каждых шести непрерывно переданных единиц необходимо вставить один ноль. Напомню, что кодирование нуля представляется как смена состояния шины на противоположное (с «J» на «K» или с «K» на «J»), а единицы как повторение предыдущего символа.

На рис. 13.5 показана диаграмма состояний машины **USBSendData**. В ней видны все фазы работы машины определяемых регистром состояний **state**, в том числе три основных фазы: «**State 0: Send SYNC and PID**», «**State 1: Send data bits**» и «**State 2: Send CRC bits**». Для каждой из этих трех фаз используется свой буфер передаваемых данных: **sync_pid_buffer**, **io.data** и **crc16_buf** соответственно. Буфер **sync_pid_buffer** формируется путем конкатенации **io.pid** и преамбулы. Буфер **crc16_buf** вычисляется функцией **calc_crc16_usb()** путем подачи на вход предыдущего значения контрольной суммы и блока передаваемых данных длиной 8 бит накопленного в регистре **crc_byte**. Данная функция порождается в сложную комбинационную схему, её мы более подробно рассмотрим далее.

Процедура «бит-стаффинга» реализуется внутри функции **sendKJ()** вызовом еще одной функции — **make_stuffing()**. Эта функция производит учет числа непрерывно переданных единиц в регистре **ones** и, если необходимо выполнить «бит-стаффинг», заменяет полезный бит на нулевое значение, при этом устанавливает флаг **stuffing**. Этот флаг используется для временной остановки счетчика **bit_count**, что приводит к передаче по шине одного дополнительного бита данных равного нулю. После чего счетчик накопленных единиц обнуляется. флаг **stuffing** сбрасывается и процесс передачи полезных данных продолжается. Функция **make_stuffing()** порождает еще один вложенный КА состояние которого определяется регистром **ones**.

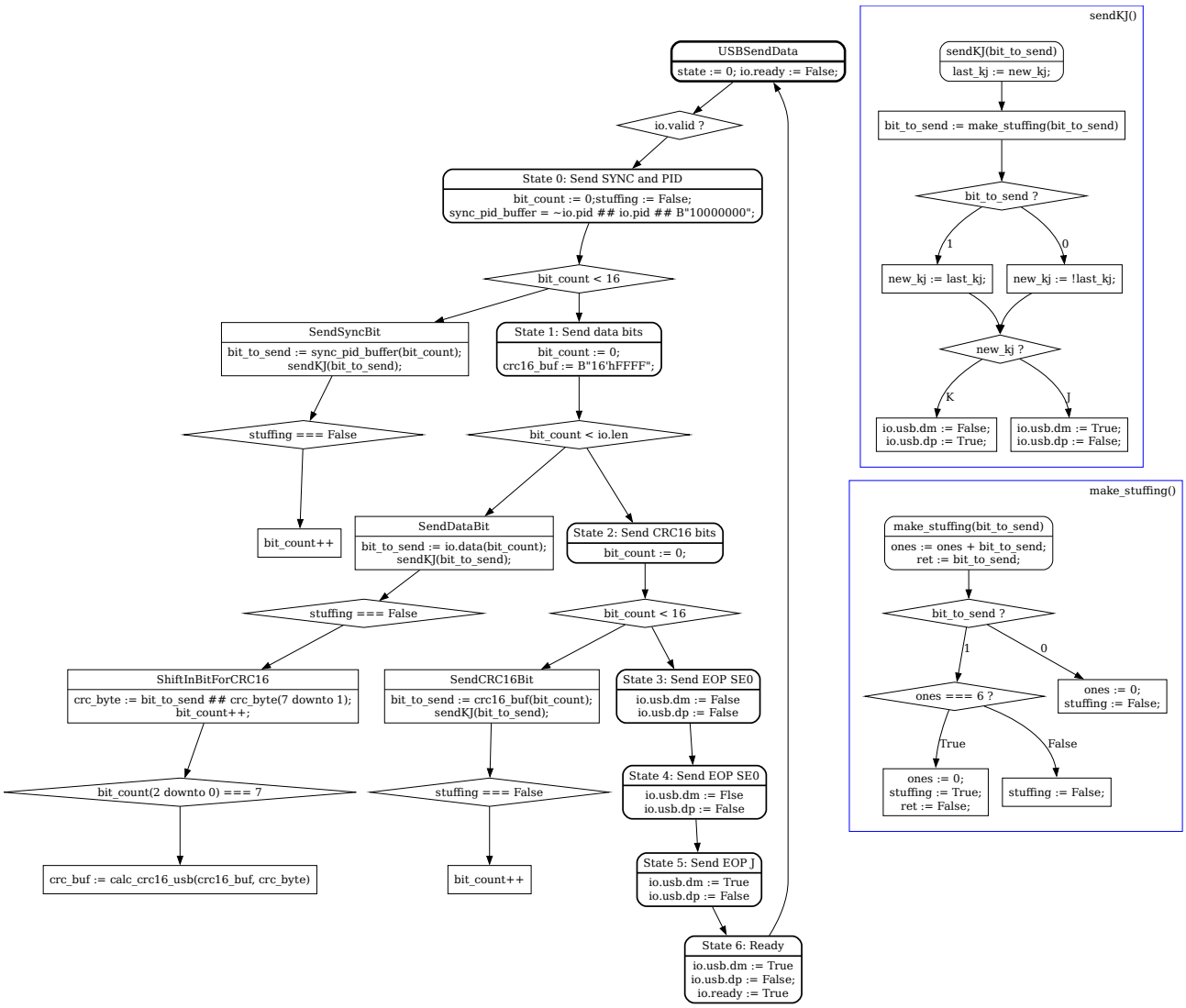

Рис. 13.5 Диаграмма переходов состояний вспомогательного автомата USBSendData.

5.1.7. Вспомогательный конечный автомат USBReceiver

Перед тем как описывать машину состояний для автомата приема пакета данных от USB устройства, следует изложить идею на которой он базируется. Основная проблема при приеме данных состоит в том, что у нас нет четкого референсного тактового сигнала по которому мы можем сэмплировать шину и конвертировать её состояния («символы») в биты данных. У нас есть свой внутренний тактовый сигнал который может немного расходиться по частоте и иметь смещение по фазе относительно передаваемых по шине символов, что вызовет появление случайных ошибок сэмплирования. Вторая проблема которую предстоит решить — это удаление лишних нулей («де-стаффинг»). И третья — расчет CRC16 для входного потока данных. Как действовать с вычислением CRC16 в целом понятно — необходимо складывать входные биты данных в отдельный 8-ми битный буфер и когда он заполнится (каждый 8-бит) передавать его в блок расчета CRC16, при этом сохраняя результат в еще одном буфере. Удаление лишних нулей задача тоже несложная — будем учитывать количество единиц в отдельном регистре **ones** и по достижению его значения числа 6 будем пропускать один бит данных, по аналогии с тем как это выполнено в машине состояний **USBSendData**. Но что делать с тактовым сигналом для сэмплирования данных?

Изучив несколько простых реализаций USB протокола выложенных на Github-е, я с удивлением обнаружил, что многие разработчик предпочитают проблему формирования тактового сигнала на приемной стороне «замести под коврик», то есть действуя по следующему принципу: выбирают внутреннюю частоту сэмплирования равной или близкой к частоте следования битов на шине (1,5 МГц в нашем случае), дожидаются появления на шине символа «K» (смена полярности сигналов **usb.dm** и **usb.dp** из состояния 1/0 в 0/1) и начинают сэмплирование по одном биту за такт внутренней частоты. Такой подход имеет ряд изъянов. Во-первых, получить частоту равную 1,5 МГц даже с помощью PLL удается не всегда (нет подходящих делителей), частота на принимающей стороне всегда отличается, и достаточно существенно, от частоты сэмплирования на передающей стороне. Во-вторых, старт приема данных по первому же фронту D+ (или спаду D-) не всегда приводит к попаданию в начало битового интервала (на начало бита) за счет разницы фаз тактовых сигналов на приемной и передающей сторонах. Все это приводит к большой вероятности возникновения ошибки приема, т. е. к большому числу пакетов принятых с ошибкой. А некоторые опенсорсные реализации настолько аскетичны, что даже не имеют расчета и проверки кода CRC16.

Одна из таких реализаций USB 1.0 контроллера, широко разбежавшаяся по любительским проектам, представлена пользователем Gitub-а с ником **hi631** (Hiromichi Kitahara), создателем проекта эмулятора игровой приставки NES для платы TangNano-9K. В целом, данная реализация даже работает, но очень и очень ненадежно. Особенно проблемы заметны при подключении USB клавиатуры — каждое второе нажатие выдается с ошибочным кодом нажатой клавиши. Тем не менее данный проект и реализация USB 1.0 в нём очень интересны, а его изучение натолкнуло меня на мысль как сделать прием данных при расхождении в частотах и фазах более стабильным. Реализация USB 1.0 от Hiromichi Kitahara интересна еще и тем, что она самая минималистичная из имеющихся на сегодня в свободном доступе. Её код на языке Verilog составляет порядка 600 строк. Плюс 200 строк кода программы на специализированном ассемблере предназначенном для внутреннего вычислителя. Плюс небольшой компилятор на языке Perl (около 100 строк кода) для преобразования ассемблера в машинный код. Проект определенно заслуживает внимания, но не пытайтесь по нему понять устройство USB хост-контроллера. Уверяю, у Вас ничего не получится, так как представленный Hiromichi Kitahara код сильно заоптимизирован, написан небрежно и с очень странными двух-трех буквенными обозначениями регистров. Понять его радикально нельзя!

Чтобы решить проблему частоты сэмплирования я решил воспользоваться тем фактом, что все пакеты передаваемые по шине USB 1.0 содержат преамбулу SYNC состоящую из 3-х последовательных переходов «K-J» (их можно выразить так: «K-J-K-J-K-J»), после которых следуют два символа «K-K». Мы можем использовать два перехода «K-J» для того, чтобы определить длительность одного битового интервала **bit_duration** выраженную в тактах базовой частоты (только два из трех потому, что делить сумму на 4 можно сдвигом). Эта базовая частота должна быть в 8 (а лучше в 16) раз выше частоты следования символов, чтобы в один битовый интервал помещалось достаточное число тактов базовой частоты для корректировки смещения фазы внутри битового интервала. Зная длину битового интервала в тактах мы может с хорошей точностью определять середину битового интервала путем отсчитывания половины значения **bit_duration** от начала фронта (или спада) на линиях **usb_dm**/**usb_dp** — это и будет момент сэмплирования. Получается, что наш приемник будет пересинхронизироваться с каждым новым фронтом (или спадом) вычисляя его середину битового интервала для считывания передаваемого по шине символа. Сдвиг фазы на один, два или даже три такта базовой частоты никак не повлияет на качество приема передаваемого бита так, как момент сэмплирования все равно попадет в битовый интервал .

Такой алгоритм хорош еще и тем, что частота следования символов может «плавать» в достаточно широких пределах.

Таким образом работа машины состояний реализующей автомат **USBReceiver** для приема данных по шине USB 1.0 в режиме «Low Speed» будет складываться из трех фаз:

- Фаза I: «Калибровка» и расчет длительности битового интервала выраженной в тактах базовой частоты.
- Фаза II: «Ожидание конца преамбулы SYNC» - подождать пока закончатся два символа «K» преамбулы.
- Фаза III: «Прием данных» - сэмплирование и декодирование принимаемых данных посередине битового интервала, включая передаваемый блок CRC16. Расчет своего значения CRC16 по ходу приема.

Работа машины состояний **USBReceiver** прекращается когда на USB шине детектируется состояние «SE0» и оно продолжается непрерывно в течении одного битового интервала (что учитывается таймером **T1**);

Ниже на рис. 13.6.1, 13.6.2 и 13.6.3 приведены диаграммы переходов состояний раздельно для каждой из трех фаз конечного автомата **USBReceiver**.

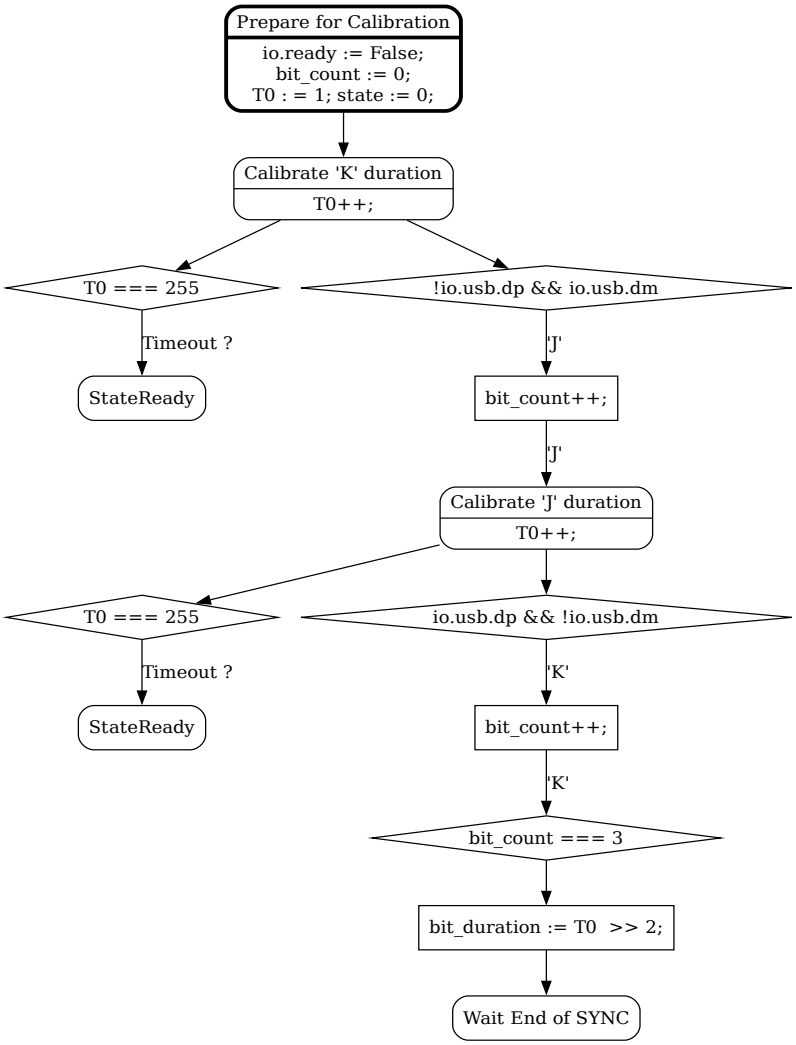

Рис. 13.6.1. Диаграмма переходов состояний вспомогательного автомата USBReceiver для фазы I «Калибровка».

В фазе I «Калибровка» машина состояний **USBReceiver** начинает работать когда на шине присутствует символ «K» (первый символ в преамбуле). Она последовательно переходит из состояния ожидания символа «J» к ожиданию следующего символа «K» используя счетчик **T0** для подсчета числа тактов базового тактового сигнала. Счетчик **bit_count** используется для учета числа циклов. Как только прошло **два** полных цикла «K-J», то есть значение **T0 === 3**, машина вычисляет значение длительности одного битового интервала путем деления **T0** на 4 (смещение на два бита вправо), сохраняет это значение во внутренний регистр **bit_duration** и переходит к фазе 2. Очевидно, что для простоты реализации целесообразно использовать только два цикла «K-J» из трех, так как делить на 6 аппаратно гораздо сложнее чем на 4. Но даже 4-х битовых интервалов (двух циклов «K-J») достаточно чтобы с хорошей точностью вычислить длину одного битового интервала. Чтобы компенсировать один тактовый сигнал, который не учитывается в состоянии инициализации, мы инициализируем **T0** единицей, а не нулем.

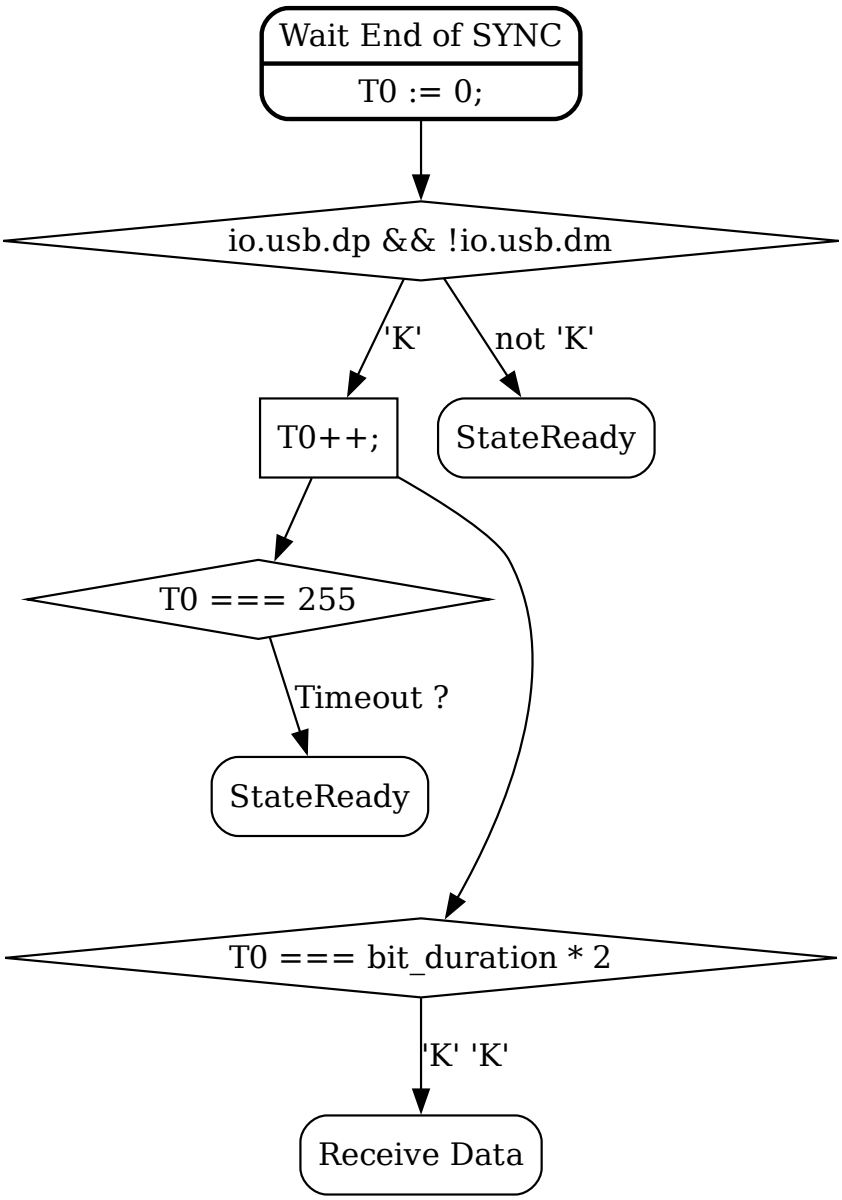

Рис. 13.6.2. Диаграмма переходов состояний вспомогательного автомата USBReceiver для фазы II «Ожидание конца преамбулы».

В фазе II «Ожидание конца преамбулы» автомат дожидается появления на шине USB символа «K» и вычисляет его длительность с помощью счетчика **T0**. Если она равна удвоенному значению **bit_duration**, то следующий битовый интервал будет содержать полезные данные которые следует принимать в буфер, а значит автомат переходит в фазу III.

В фаза III «Прием данных» устроена несколько сложнее. В этой фазе машина использует счетчик T0 для того чтобы вычислить середину битового интервала (**T0 === bit_duration/2**) при этом счетчик **T0** сбрасывается каждый раз когда на шине происходит смена полярности сигнала **io.usb.dp**, для чего его предыдущее значение сохраняется во внутреннем регистре **last_dp**. Если счетчик досчитал до середины битового интервала, то происходит декодирование текущего передаваемого символа в бит данных **bit_received**. Если счетчик **T0** достиг предельного значения **255**, то такая ситуация принимается за аварийную и машина состояний завершает свою работу, то есть переходит в состояние **StateReady** и поднимает флаг готовности **io.ready**.

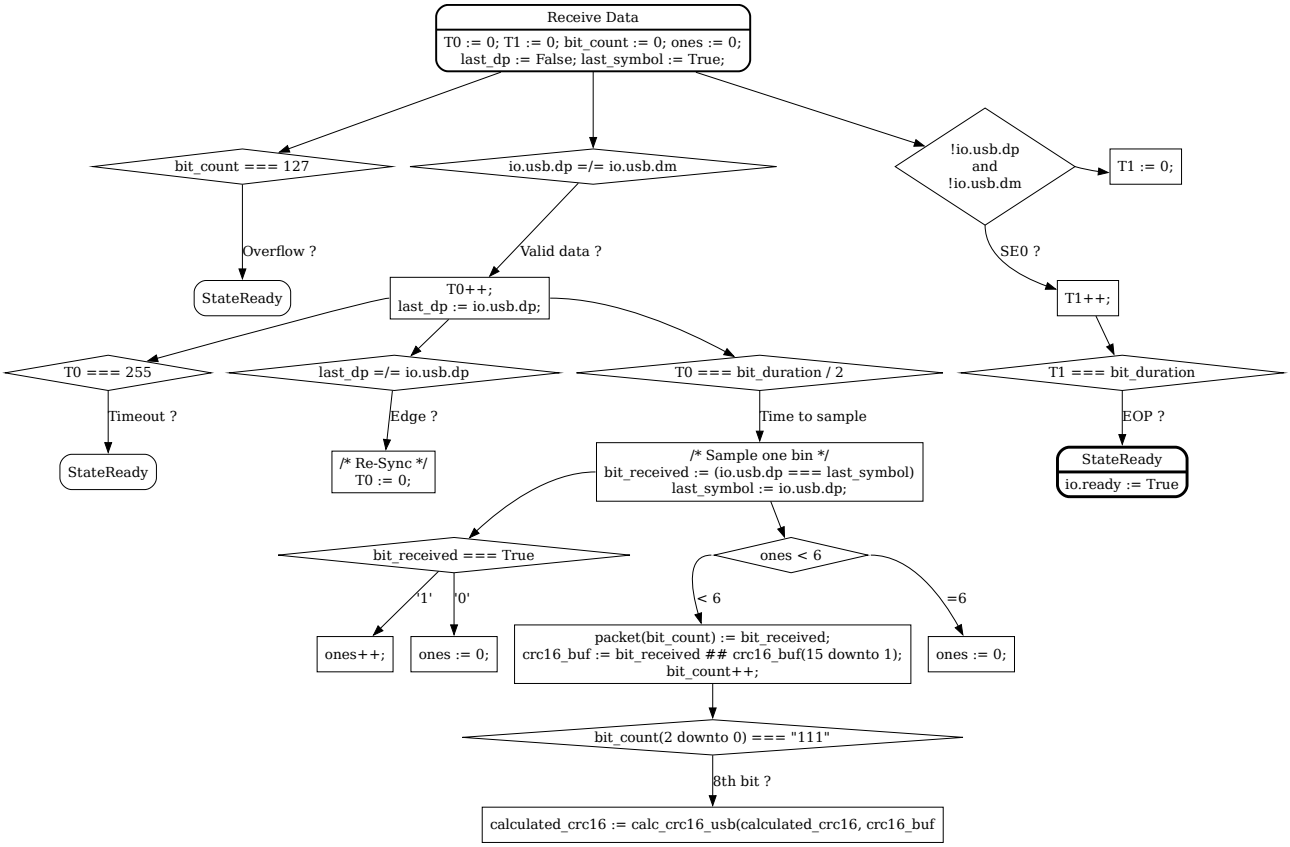

Рис. 13.6.3. Диаграмма переходов состояний вспомогательного автомата USBReceiver для фазы III «Прием данных».

На каждый принятый бит данных машина производит учет числа непрерывно принимаемых единиц в регистре **ones**. Если это число меньше 6, то производится инкремент счетчика принятых битов данных **bit_count**, принятый бит данных складывается в выходной буфер **packet** и одновременно задвигается во внутренний регистр **crc16_buf** для последующего использования в расчете кода CRC16. Функция **calc_crc16_usb()** выполняющая этот расчет вызывается раз в 8 бит (т. е. на каждый принятый байт). Результат расчета помещается в выходной регистр **io.calculated_crc16**. Если значение числа

последовательно принятых единиц достигается 6, то инкремент **bit_count** не производится, а регистр **ones** сбрасывается в ноль. Таким образом пропускается один принятый бит данных.

В фазе III автомат отслеживает появление на шине символа «SE0» и длительность его пребывания путем инкремента счетчика **T1** в этом состоянии шины. Если **T1** достиг значения равного длительности битового интервала (**T1 === bit_duration**), то такое событие воспринимается как конец передачи пакета (EOP) и машина переходит в состояние **StateReady** с поднятием флага **io.ready**.

5.1.8. Тактирование конечных автоматов хост-контроллера

Тактирование основного автомата **USBMain** может производится от любого внутрисистемного тактового сигнала который должен быть на порядок (в 8 и более раз) выше по частоте чем скорость передачи данных (частота следования символов «K» и «J») по шине USB и должен быть кратным скорости передачи данных. Для шины USB 1.0 «Low Speed» скорость передачи данных по которой равной 1,5 Мбит/сек, вполне достаточно тактового сигнала частотой 12 МГц.

Тактирование вспомогательных автоматов производится от внутреннего строба **clock_strobe** который вырабатывается счетчиком-делителем частоты **clock_div**. Предельное значение счетчика передается всем вспомогательным автоматам на вход **io.clock_div**. Это значение может быть вычислено на стадии компиляции в зависимости от значения частоты тактового сигнала используемого для тактирования основного автомата. Далее мы посмотрим как это можно описать на языке SpinalHDL.

5.2. Имплементация хост-контроллера на языке SpinalHDL

Так как вся синтезируемая система-на-кристалле (KarnixSoC) создается на языке описания аппаратуры SpinalHDL, то логичным будет реализовать наш USB хост-контроллер на этом же языке. В предыдущих статьях этого цикла я уже подробно рассматривал синтаксис и возможности этого языка. Поэтому не буду заострять внимания на этом моменте, а сразу перейду к описанию аппаратного интерфейса шины USB. Замечу лишь, что приведенный в диаграммах текст на формальном языке — это и есть код на SpinalHDL взятый прямо из реализации, а сами диаграммы построены пост-фактум и полностью отражают текущую реализацию хост-контроллера, за исключением моментов связанных с интеграцией в шину APB3 синтезируемой СнК.

5.2.1. Интерфейсные классы USBInterface и USB_IO

Опишем интерфейсный класс **USBInterface**, унаследованный от библиотечного класса **ImasterSlave**, содержащий два сигнала шины USB: **dm** (D-) и **dp** (D+). По своей природе шина USB двунаправленная и может менять своё состояние в любой момент времени, поэтому сигналы **dm** и **dp** объявим двунаправленными (**inout**) и аналоговыми (**Analog**). Еще нам потребуется переопределить процедуру **asMaster()** базового интерфейсного класса, которая меняет направления сигналов для «мастер» компонентов, так чтобы она оставляла сигналы **dm** и **dp** двунаправленными (или вообще не изменяла из амплуа). Эта процедура неявно вызывается из разных мест связующей логики СнК, поэтому данный момент является важным.

Итак, интерфейсный класс **USBInterface** на SpinalHDL:

Листинг 8.1. Интерфейсный класс USBInterface.

```
case class USBInterface() extends Bundle with IMasterSlave{
  val dm = inout(Analog(Bool()))
  val dp = inout(Analog(Bool()))

  override def asMaster(): Unit = {
    inout(dm, dp) // make dm and dp always bi-directional
  }
}
```

Опишем еще один интерфейсный класс **USB_IO** который будем использовать для определения внешних связей различных частей нашего хост-контроллера. Добавим в него комплексный сигнал **usb** описываемый только что созданным интерфейсным классом **USBInterface**, также добавим общие сигналы управления: входной сигнал **valid**; выходной сигнал **ready**; входной сигнал **clock_div**, содержащий рассчитанный параметр делителя частоты для вырабатывания строба; и один выходной тестовый сигнал **test**. Данный класс будет общим для всех вспомогательных машин состояний нашего хост-контроллера. На SpinalHDL его описание примет следующий вид:

Листинг 8.2. Интерфейсный класс USB_IO.

```
class USB_IO extends Bundle {
    val usb       = USBInterface()
    val valid     = in Bool()
    val ready     = out Bool()
    val clock_div = in UInt(8 bits)
    val test      = out Bool()

    ready := False // default value, it will be changed in implementation
}
```

Сигналу **ready** тут же присваивается значение по умолчанию — **False**. В последствии, при реализации конкретного автомата, мы будем расширять класс **USB_IO** добавляя в него требуемые нам интерфейсные сигналы.

5.2.2. Базовый класс USBSendReceive

Из приведенных выше диаграмм видно, что все вспомогательные конечные автоматы построены единообразно, а значит их реализации будут содержать общие (пересекающиеся) куски кода. Вполне логичным является собрать весь пересекающийся функционал в общий базовый класс и использовать его как основание для всех вспомогательных машин состояний. Назовем этот класс **USBSendReceive** и поместим в него следующие поля (переменные представляющие собой сигналы и регистры) и методы (функции и процедуры):

- Сигнал строба **clock_strobe** формируемый делителем частоты.
- Регистр делителя частоты **clock_div**.
- Регистр **bit_count** для счетчика обработанных (переданных/принятых) битов данных.
- Регистр **last_kj** для запоминания последнего переданного по шине символа.
- Регистр **ones** для учета числа непрерывно переданных/принятых единиц.
- Регистр флага **stuffing** указывающий на необходимость вставить (или пропустить) один дополнительный нулевой бит («бит-стаффинг»).
- Регистр **crc16** для сохранение промежуточного значения рассчитываемой контрольной суммы.
- Процедуру **make_clock_strobe()** для выработки сигнала **строба clock_strobe**.
- Процедуру **inc_bit_count()** для приращения счетчика обработанных битов данных **bit_count**.
- Процедуру **make_stuffing()** для подсчета числа непрерывно переданных единиц в регистре **ones** и управления флагом **stuffing**.
- Процедура **sendSE0()** для перевода шины в состояние «SE0» (io.usb.dp := False; io.usb.dm := False).
- Процедура **sendK()** для перевода шины в состояние «K» (io.usb.dp := True; io.usb.dm := False).
- Процедура **sendJ()** для перевода шины в состояние «J» - противоположное «K».
- Процедуру **sendKJ()** для вычисления текущего передаваемого символа («K» или «J») исходя из предыдущего переданного символа и текущего передаваемого бита данных и сохранения последнего переданного символа в регистр **last_kj**. Использует процедуры **sendK()** и **sendJ()**.
- Процедуру **reset_last_kj()** для сброса регистра **last_kj** в начальное состояние по сигналу **io.valid**.
- Функцию **calc_crc16_usb()** для расчета промежуточного значения CRC16 для принятого или переданного байта данных.

Так как разные автоматы будут пользоваться разным набором этих фич, то для того, чтобы в синтезируемой цифровой схеме не дублировались бесполезные (не задействованные) куски аппаратуры, сделаем их отключаемыми. Для этого введем в конструктор базового класса **USBSendReceive()** следующие параметры:

- **hasStrobe : Boolean** — включает в класс регистр **clock_div** и сигнал **clock_strobe,** а также процедуру **make_clock_strobe()**;
- **szBitcount : Int** — задает размер (битность) счетчика **bit_count**, сделаем его равным 0 по умолчанию, чтобы отключать **bit_count** если он не задействован;

- **hasSendKJ : Boolean** — включает регистры **last_kj**, **ones** и **stuffing**, а также процедуры **reset_last_kj(), sendKJ()** и **make_stuffing()** которые работают с этими регистрами;
- **hasCRC16 : Boolean** — включает в класс функцию расчета **calc_crc16_usb()**.

В результате получим следующий код для класса **USBSendReceive**:

Листинг 8.3. Базовый класс USBSendReceive.

```
class USBSendReceive(var hasStrobe  : Boolean = true,
                     var hasSendKJ  : Boolean = true,
                     var szBitcount : Int = 0,
                     var hasCRC16   : Boolean = false) extends Component {

    if(hasSendKJ) { hasStrobe = true; if(szBitcount == 0) szBitcount = 13;  }

    val clock_div    = (hasStrobe   ) generate Reg(UInt(8 bits)).addTag(crossClockDomain)
    val clock_strobe = (hasStrobe   ) generate False
    val bit_count    = (szBitcount>0) generate Reg(UInt(szBitcount
bits)).addTag(crossClockDomain)
    val stuffing     = (hasSendKJ   ) generate False
    val ones         = (hasSendKJ   ) generate Reg(UInt(3 bits)).addTag(crossClockDomain)
    val last_kj      = (hasSendKJ   ) generate Reg(Bool()).addTag(crossClockDomain)
    val crc16        = (hasCRC16    ) generate Reg(Bits(16 bits)).addTag(crossClockDomain)

    def reset_last_kj(io: USB_IO) = hasSendKJ generate {
      when(!io.valid) {
        last_kj := False // 'J'
      }
    }

    def sendK(io: USB_IO) = {
      io.usb.dm := False
      io.usb.dp := True
    }

   def sendSE0(io: USB_IO) = {
      io.usb.dm := False
      io.usb.dp := False
    }

    def sendJ(io: USB_IO) = {
      io.usb.dm := True
      io.usb.dp := False
    }

    def sendKJ(io: USB_IO, input_bit: Bool) = hasSendKJ generate {
      val new_kj = Bool()

      // convert bit to K/J symbol depending on last symbol sent
      when(clock_strobe && ((input_bit === False) || stuffing)) {
        new_kj := !last_kj // Transition
        last_kj := new_kj
      } otherwise { // No transition
        new_kj := last_kj
      }

      // Transmit symbol, USB 1.0 Low Speed
      when(new_kj) { // 'K' (True)
        sendK(io)
      } otherwise { // 'J' (False)
        sendJ(io)
      }
    }

    def calc_crc16_usb(crc_in: Bits, din: Bits) : Bits = hasCRC16 generate {
      val ret = Bits(16 bits)

        ret(0) :=       din(7) ^ din(6) ^ din(5) ^ din(4) ^ din(3) ^
                        din(2) ^ din(1) ^ din(0) ^ crc_in(8) ^ crc_in(9) ^
                        crc_in(10) ^ crc_in(11) ^ crc_in(12) ^ crc_in(13) ^
                        crc_in(14) ^ crc_in(15)
```

```
    ret(1) :=       din(7) ^ din(6) ^ din(5) ^ din(4) ^ din(3) ^ din(2) ^
                    din(1) ^ crc_in(9) ^ crc_in(10) ^ crc_in(11) ^
                    crc_in(12) ^ crc_in(13) ^ crc_in(14) ^ crc_in(15)
    ret(2) :=       din(1) ^ din(0) ^ crc_in(8) ^ crc_in(9)
    ret(3) :=       din(2) ^ din(1) ^ crc_in(9) ^ crc_in(10)
    ret(4) :=       din(3) ^ din(2) ^ crc_in(10) ^ crc_in(11)
    ret(5) :=       din(4) ^ din(3) ^ crc_in(11) ^ crc_in(12)
    ret(6) :=       din(5) ^ din(4) ^ crc_in(12) ^ crc_in(13)
    ret(7) :=       din(6) ^ din(5) ^ crc_in(13) ^ crc_in(14)
    ret(8) :=       din(7) ^ din(6) ^ crc_in(0) ^ crc_in(14) ^ crc_in(15)
    ret(9) :=       din(7) ^ crc_in(1) ^ crc_in(15)
    ret(10) :=      crc_in(2)
    ret(11) :=      crc_in(3)
    ret(12) :=      crc_in(4)
    ret(13) :=      crc_in(5)
    ret(14) :=      crc_in(6)
    ret(15) :=      din(7) ^ din(6) ^ din(5) ^ din(4) ^ din(3) ^ din(2) ^
                    din(1) ^ din(0) ^ crc_in(7) ^ crc_in(8) ^ crc_in(9) ^
                    crc_in(10) ^ crc_in(11) ^ crc_in(12) ^ crc_in(13) ^
                    crc_in(14) ^ crc_in(15)

  return ret
}
def make_clock_strobe(io: USB_IO) = hasStrobe generate {
  when(io.valid) {
    clock_div := clock_div + 1
    when(clock_div === io.clock_div) {
      clock_div := 0
    }
    when(clock_div === 0) {
      clock_strobe := True
    }
  } otherwise {
    clock_div := 0
  }
}

def inc_bit_count(io: USB_IO) = (szBitcount > 0) generate {
  when(io.valid) {
    when(clock_strobe) {
      bit_count := bit_count + 1
    }
  } otherwise {
    bit_count := 0
  }
}
def make_stuffing(io: USB_IO, bit_to_send: Bool) = hasSendKJ generate {
  when(io.valid) {

    when(ones === 6) {
      stuffing := True
    }

    when(clock_strobe) {

      when(bit_to_send) { // Count ones if bit_to_send == '1'
        ones := ones + 1
      } otherwise {
        ones := 0
      }

      when(stuffing) {
        ones := 0
      }

      when(!stuffing) { // Advance bit_count only if not stuffing zero
        bit_count := bit_count + 1
      }
    }

    when(stuffing) { // Substitute current bit to '0' if stuffing
      bit_to_send := False
    }

  } otherwise {
    ones := 0
    bit_count := 0
```

```
        }
      }
    }
```

Небольшое замечание стоит сделать относительно конструкции вида **(hasStrobe) generate** — таким способом проверяется значение параметра **hasStrobe** и включает (или отключает) следующий за ней код если параметр установлен в **true**. Аналогичном образом работает оператор **generate** стоящий в заголовке функции или процедуры. В остальном, приведенный выше код на SpinalHDL легко читается любым программистом даже без знания синтаксиса языка Scala и не требует развернутого комментария.

Но стоит немного обсудить выбор дефолтных значений для параметров конструктора определяющих условную генерацию аппаратуры. Большинство вспомогательных конечных автоматов занимаются пересылкой данных по шине, а значит они зависят от процедуры **sendKJ()** и её регистров, поэтому значение по умолчанию для параметра **hasSendKJ = True**. Аналогичным образом выбрано значение для параметр **hasStrobe = True** — все автоматы требуют генерации сигнала строба. Размер регистра **bit_count** во всех автоматах может быть разным, поэтому по умолчанию параметру **szBitcount** присвоим 0. Процедура расчета CRC16 используется в двух из пяти вспомогательных автоматах, а значит по умолчанию установим значение параметра **hasCRC16 = False**. Значения этих параметров далее мы будем переопределять при реализации конкретных КА.

Также стоит сказать пару слов про функцию **calc_crc16_usb()**. Данная функция реализует расчет полинома CRC16 по алгоритму (математической формуле) который был рассмотрен в главе 3.1.4. «*Алгоритмы расчета контрольных сумм CRC5 и CRC16*». Эта функция принимает на вход два параметра **crc_in** и **din**. Первый параметр содержит предыдущее рассчитанное значение кода CRC16, оно каждый раз сохраняется в регистре **crc16** конечного автомата. При активации автомата этот регистр инициализируется в начальное значение **16'hFFFF** согласно алгоритму, а при завершении работы на выходы подается его отображенное (reversed) значение также согласно алгоритму. Второй параметр **din** содержит 16 бит принятых (или передаваемых) данных которые необходимо обработать алгоритмом. Эти данные накапливаются (задвигаются) во внутреннем регистре автомата в процессе передачи (или приеме) и «скармливаются» в функцию **calc_crc16_usb()** когда набрано полное слово. Это можно наблюдать в приведенных выше диаграммах переходов состояний для вспомогательных автоматов **USBReceiver** и **USBSendData**.

5.2.3. Класс USBSendSE0

Следующим шагом мы рассмотрим реализацию самого простого из вспомогательных конечных автоматов с одноименным названием. Напомню, что задача данного автомата перевести шину USB в состояние «SE0» - это когда обе сигнальных линии (D+ и D-) подтянуты к «земле» («к нулю» или «находятся в состоянии False») на заданное количество битовых интервалов, где один битовый интервал равен по времени продолжительности передачи одного бита данных (одного символа).

Автомат **USBSendSE0** зависит от сигнала строба **clock_strobe** и требует для работы счетчик битовых интервалов **bit_count**, причем размер счетчика должен быть достаточно большим чтобы сформировать сигнал «Bus Reset» в течении более 10 мс (более 15000 отсчетов). Данный автомат не занимается пересылкой или приемом данных, а значит не зависит от процедуры **sendKJ()** и от всего что с ней связно. Исходя из этого, в конструкторе класса реализующего данный автомат, укажем следующие параметры: **hasStrobe = True**, **szBitcount = 16** и **hasSendKJ = False**. Полная реализация класса **USBSendSE0** выглядит следующим образом:

Листинг 8.4. Класс вспомогательного автомата USBSendSE0.

```
case class USBSendSE0() extends USBSendReceive(hasSendKJ = false, hasStrobe = true,
szBitcount = 16) {
    val io = new USB_IO {
        val len     = in UInt(16 bits) // number of bit intervals
    }

    make_clock_strobe(io)
    inc_bit_count(io)

    when(io.valid) {

      when(clock_strobe && bit_count === io.len) { // Ready, EOP: 'J'
        sendJ(io)
        io.ready := True
      } otherwise {
        sendSE0(io)
      }
    }
}
```

Работа данного автомата активируется по сигналу **io.valid** и завершается установкой **io.ready := True** в момент, когда регистр **bit_count** достигает значения **io.len** передаваемого во входном комплексном сигнале **io** определяемым интерфейсным классом **USB_IO** и дополненным новым параметром **len**.

В процессе работы у автомата порождаемого данным кодом постоянно (на каждый такт базового сигнала) вызывается процедура **make_clock_strobe()** которая порождает аппаратуру вращающую счетчик **clock_dev** и формирующую строб **clock_strobe**. Также на каждый такт вызывается процедура **inc_bit_count()**, она порождает аппаратуру для приращения значение регистра **bit_count** на единицу по сигналу строба.

Если посмотреть на диаграмму изображенную на рис. 13.2, то можно заметить, что почти весь код данного класса представлен на этой диаграмме, а операторы языка образуют связи между частями кода что выражено на диаграмме в виде стрелок-связей. Оператор **when()** на диаграммах представлен «ромбом» и является оператором условного исполнения. При реализации остальных вспомогательных автоматов все будет выглядеть аналогичным образом.

5.2.4. Класс USBSendShortToken

Задача вспомогательного автомата **USBSendShortToken** состоит в том, чтобы переслать по шине 16 бит данных включая преамбулу и PID. Чтобы передать на вход значение PID, реализация класса **USBSendShortToken** расширяет интерфейсный класс **USB_IO** дополнительным сигналом **pid** размерностью 4 бита. В тело класса добавляется регистр **buffer** — буфер из которого изымаются данные для передачи и сигнал **bit_to_send** ссылающийся на бит буфера содержащий текущий передаваемый бит данных.

Автомат **USBSendShortToken** зависит от сигнала строба **clock_strobe**, требует для работы счетчик битовых интервалов **bit_count** размером 5 бит и использует процедуру **sendKJ()**. Потому в параметрах конструктора укажем только **szBitcount = 5**, так как все остальные параметры по умолчанию имеют требуемые значения. Реализация класса **USBSendShortToken** представлена ниже.

Листинг 8.5. Класс вспомогательного автомата USBSendShortToken.

```
case class USBSendShortToken() extends USBSendReceive(szBitcount = 5) {
    val io = new USB_IO {
        val pid       = in Bits(4 bits)
    }
```

```
    val buffer = ~io.pid ## io.pid ## B"10000000"
    val bit_to_send = buffer(bit_count(3 downto 0))

    make_clock_strobe(io)
    inc_bit_count(io)
    reset_last_kj(io)

    when(io.valid) {
      when(bit_count === 20) { // Ready, EOP: 'J'
        sendJ(io)
        io.ready := True
        bit_count := 20
      } elsewhen(bit_count === 19) { // EOP: 'J'
        sendJ(io)
      } elsewhen(bit_count === 18 && clock_strobe) { // EOP: 'SE0' - coner case
        sendSE0(io)
      } elsewhen((bit_count === 17) || (bit_count === 18)) { // EOP: 'SE0'
        sendSE0(io)
      } elsewhen(bit_count === 16 && clock_strobe) { // EOP: 'SE0' - coner case
        sendSE0(io)
      } otherwise {
        sendKJ(io, bit_to_send)
      }
    }
}
```

В отличии от предыдущего, более простого автомата, в реализацию данного добавился вызов процедуры **reset_last_kj()** которая порождает аппаратуру инициализации регистра **last_kj** при сбросе сигнала **io.valid**.

Буфер представляет собой **buffer** 16-ти битный сигнал получаемый конкатенацией инверсного значения ~**io.pid**, прямого значения **io.pid** и преамбулы в битовом выражении которая равна **"10000000"** .

Напомню, что данные по шине USB передаются от младших битов к старшим, то есть сначала будут передаваться 7 нулей за которыми проследует одна единица. Так, как начальное значение **last_kj** устанавливается в «J», то передача первого нуля приведет к смены состояния линии на противоположное, а значит первый отправленный символ будет «K». После чего следующий ноль еще раз сменит состояние линии на символ «J» и так несколько раз. В результате по шине будет передана последовательность символов «KJKJKJKK», где последний символ «K» это переданная единица из преамбулы.

После того как автомат передаст по шине 16 бит данных, он передаст сначала два символа «SE0», а потом два символа «J», что информирует принимающую сторону о состоянии EOP (конец пакета). Автомат остановится на значении **bit_count === 20** с поднятым сигналом **io.ready** и состоянием «J» на шине.

5.2.5. Класс USBSendLongToken

Задача вспомогательного автомата **USBSendLongToken** состоит в том, чтобы переслать по шине расширенный токен размером 32 бита содержащий помимо PID еще 7 бит адреса устройства (ADDR) и 4 бита номера конечной точки (ENDP). В конце пакета добавляется 5 битов код CRC5 рассчитываемого по 11 битам данных (только ADDR + ENDP, идентификатор PID не участвует в расчете CRC5).

Для решения поставленной задачи имплементация класса **USBSendLongToken** расширяет комплексный сигнал **io** представляемый интерфейсным классом **USB_IO** тремя дополнительными сигналами: **pid** (4 бита), **addr** (7 бит) и **endp** (4 бита) и добавляет в тело класса функцию **calc_crc5_usb()** с одним входным параметром **din** размерностью 11 бит. Данная функция порождает комбинационную схему вычисляющую значение CRC5 по алгоритму описание которого рассмотрено в главе 3.1.4. «*Алгоритмы расчета контрольных сумм CRC5 и CRC16*».

Буфер входных данных **buffer**, как и у предыдущего автомата, формируется путем конкатенации входных данных. К ним добавляется кода CRC5 получаемый функцией **calc_crc5_usb()**.

Реализация автомата **USBSendLongToken** использует те же блоки, что и **USBSendShortToken**, поэтому параметры конструктора аналогичны. Полный код реализации класса **USBSendLongToken** выглядит следующим образом:

Листинг 8.6. Класс вспомогательного автомата USBSendLongToken.

```
case class USBSendLongToken() extends USBSendReceive(szBitcount = 6) {

    val io = new USB_IO {
        val pid       = in Bits(4 bits)
        val addr      = in Bits(7 bits)
        val endp      = in Bits(4 bits)
    }

    def calc_crc5_usb(din: Bits) : Bits = {
      val ret = Bits(5 bits)
      val din_rev = din.reversed;
      ret(0) := din_rev(10) ^ din_rev(9) ^ din_rev(6) ^ din_rev(5) ^ din_rev(3) ^ din_rev(0)
^ True
      ret(1) := din_rev(10) ^ din_rev(7) ^ din_rev(6) ^ din_rev(4) ^ din_rev(1) ^ True
      ret(2) := din_rev(10) ^ din_rev(9) ^ din_rev(8) ^ din_rev(7) ^ din_rev(6) ^ din_rev(3)
^ din_rev(2) ^ din_rev(0) ^ True
      ret(3) := din_rev(10) ^ din_rev(9) ^ din_rev(8) ^ din_rev(7) ^ din_rev(4) ^ din_rev(3)
^ din_rev(1)
      ret(4) := din_rev(10) ^ din_rev(9) ^ din_rev(8) ^ din_rev(5) ^ din_rev(4) ^ din_rev(2)
^ True
      return ret.reversed ^ B"11111"
    }

    val crc5_out = calc_crc5_usb(io.endp ## io.addr)
    val buffer = crc5_out ## io.endp ## io.addr ## ~io.pid ## io.pid ## B"10000000"
    val bit_to_send = buffer(bit_count(4 downto 0))

    make_clock_strobe(io)
    inc_bit_count(io)
    reset_last_kj(io)

    when(io.valid) {
      when(bit_count === 36) { // Ready, EOP: 'J'
        sendJ(io)
        io.ready := True
        bit_count := 36
      } elsewhen(bit_count === 35) { // EOP: 'J'
        sendJ(io)
      } elsewhen((bit_count === 33) || (bit_count === 34)) { // EOP: 'SE0'
        sendSE0(io)
      } elsewhen(bit_count === 32 && clock_strobe) { // EOP: 'SE0' - coner case
        sendSE0(io)
      } otherwise {
        sendKJ(io, bit_to_send)
      }
    }
}
```

Алгоритм действия при передачи у данного автомата такой же как и у **USBSendShortToken**, разница лишь в большем числе передаваемых битов данных (32 вместо 16). Автомат **USBSendLongToken**, останавливается когда счетчик **bit_count** достигает значения 36. В этом состоянии производится подъем сигнала **io.ready**, а на шине удерживается состояния «J».

5.2.6. Класс USBSendData

Класс реализующий вспомогательный автомат **USBSendData** для передачи по шине пользовательского блока данных тоже расширяет структуру сигнала **io** путем добавления в интерфейсный класс **USB_IO** следующих полей: поле **pid** для 4-х битов Packet ID, поле **len** размером 7 бит для указания длины передаваемого блока данных, и поле **data** размерностью 64 бита содержащее передаваемую по шине последовательность данных (напомню, что для USB 1.0 «Low Speed» максимальный размер полезной нагрузки составляет 64 бит).

Реализация класса **USBSendData** использует следующие функциональные блоки: процедуру **make_clock_strobe()** порождающую делитель частоты и формирующую строб **clock_strobe**; счетчик битов **bit_count** размерность 8 бит и процедуру его приращения **inc_bit_count()**; процедуру **sendKJ()** для преобразования данных в символы и отправки их на шину, а также процедуру инициализации регистра **last_kj()**. При передаче данных автомат **USBSendData** использует процедуру **cacl_crc16_usb()** для расчета кода CRC16 для передаваемого по шине блока данных. Исходя из этого, конструктор класса **USBSendData** будет содержать параметры со следующими значениями: **hasCRC16 = true, szBitcount = 8**. Все остальные параметры остаются в своих дефолтных значениях.

Процесс пересылки данных обсуждался в главе 5.1.6, а диаграмма переходов состояний приведена на рис. 13.5. Он состоит из нескольких фаз, в том числе: фаза **State 0: Send SYNC and PID**» - передается преамбула и два раза PID (прямой и инверсный), фаза «**State 1: Send data bits**» - кодирование и собственно передача битов полезной с учетом «бит-стаффинга», фаза «**State 2: Send CRC bits**» - передача 16-ти бит рассчитанного значения кода CRC16, и нескольких коротких фаз посылки завершающей последовательности завершения EOP.

Для кодирования текущей фазы потребуется 3-х битовый регистр **state** который инициализируется в нулевое значение при сбросе входного сигнала **io.valid**. Для передачи преамбулы в нулевой фазе потребуется буфер **sync_pid_buffer** построенный путем конкатенации входных данных **io.pid** и битовым представлением преамбулы. Так как расчет CRC16 ведется по-байтово, то в реализации автомата придется завести внутренний 8-ми битный регистр **crc_byte** для накопления переданных бит. В результате получим следующий код для класса **USBSendData** реализующий одноименный вспомогательный автомат:

Листинг 8.7. Класс вспомогательного автомата USBSendData.

```
case class USBSendData() extends USBSendReceive(hasCRC16 = true, szBitcount = 8) {
    val io = new USB_IO {
        val pid       = in Bits(4 bits)
        val data      = in Bits(64 bits)
        val len       = in UInt(7 bits)
    }

    val crc_byte = Reg(Bits(8 bits))
    val sync_pid_buffer = ~io.pid ## io.pid ## B"10000000"
    val state = Reg(UInt(3 bits)).addTag(crossClockDomain) init(0)
    val bit_to_send = False ; // value of data bit to be sent

    make_clock_strobe(io)
    make_stuffing(io, bit_to_send)
    reset_last_kj(io)

    when(io.valid) {

      switch(state) {
        is(0) { // sending SYNC + PID
          bit_to_send := sync_pid_buffer(bit_count(3 downto 0))
          sendKJ(io, bit_to_send)
          when(clock_strobe && bit_count === 15) {
            bit_count := 0
            state := 1 // send data
```

```
            when(io.len === 0x7f) { // max len in this phase means send empty packet
              state := 2 // send CRC16 right away
            }
          }
        }
        is(1) { // sending DATA block
          bit_to_send := io.data(bit_count(5 downto 0))
          when(clock_strobe && !stuffing) {
            val crc_byte_next = bit_to_send ## crc_byte(7 downto 1)
            when(bit_count(2 downto 0) === U"111") {
              crc16 := calc_crc16_usb(crc16, crc_byte_next.reversed)
            }
            when(bit_count === io.len) {
              state := 2
              bit_count := 0
            }
            crc_byte := crc_byte_next
          }
          // J: D- = 1, D+ = 0, K: D- = 0, D+ = 1
          // KJKJKJKK + PID + DATA + CRC16
          sendKJ(io, bit_to_send)
        }
        is(2) { // sending CRC16 block
          // CRC out is reversed and XORed
          val crc_rev = ~crc16.reversed
          bit_to_send := crc_rev(bit_count(3 downto 0))
          when(clock_strobe && bit_count === 16) {
            sendSE0(io);
            state := 3
          } otherwise {
            sendKJ(io, bit_to_send)
          }
        }
        is(3) { // sending EOP: 'SE0'
          sendSE0(io)
          when(clock_strobe) {
            state := 4
          }
        }
        is(4) { // sending EOP: 'SE0'
          sendSE0(io)
          when(clock_strobe) { // sending EOP: 'J'
            sendJ(io)
            state := 5
          }
        }
        is(5) { // sending EOP: 'J'
          sendJ(io)
          when(clock_strobe) {
            state := 6
          }
        }
        is(6) { // Ready, EOP: 'J'
          sendJ(io)
          io.ready := True
        }
      }

    } otherwise {
      state := 0
      crc16 := B"16'hFFFF"
    }
  }
```

Очевидно, что кода для этого автомата получилось несколько поболее, чем для автоматов пересылки токенов. Тем не менее логика его работы хорошо прослеживается.

5.2.7. Класс USBReceiver

Класс реализующий вспомогательный автомат для приема данных добавляет следующие поля в комплексный сигнал **io** расширяя интерфейсный класс **USB_IO**:

- поле **io.packet** представляет собой выходной сигнал размерностью 128 бит для принятого блока даных (включая CRC16);
- поле **io.bits_recv** — выходной сигнал размерностью 7 бит содержащий счетчик числа принятых бит (записанных во внутренний буфер **packet**);
- поле **io.calculated_crc16** — выходной сигнал размерностью 16 бит содержащий код CRC16 рассчитанный в процессе приема по получаемым данным;
- и поле **io.received_crc16** — выходной сигнал размерностью 16 бит содержащий последние 16 принятых битов данных, в нормальной ситуации это принятый код CRC16 рассчитанный и переданный удаленной стороной.

Процесс приема данных по шине USB представлен на диаграмма на рис. 13.6.1 — 16.6.3 и описан в главе 5.1.7. «*Вспомогательный конечный автомат USBReceiver*». Он тоже состоит из нескольких фаз, в том числе: фазы «I — Калибровка» в процессе которой производится расчет длительности битового интервала по принимаемой преамбуле; фаза «II — Ожидание конца преамбулы SYNC», фазы «III — Прием данных» в процесс которой происходит декодирование принимаемых символов в биты данных и расчет CRC16; и нескольких завершающих фаз.

Работа вспомогательного автомата **USBReceiver** сильно отличается от всех предыдущих, из общих блоков используется только **bit_count** и процедура расчета кода CRC16. Значения параметров для конструктора класса **USBReceiver** будут следующими: **hasStrobe = false, hasSendKJ = false, hasCRC16 = true, szBitcount = 7**.

Вспомогательный автомат **USBReceiver** содержит три счетчика-таймера: **T0** — для вычисления и последующего учеты длительности битового интервала, **T1** — для учета длительности EOP последовательности и **T2** — защитный таймер для аварийного завершения приема если удаленная сторона отказывается передавать данные.

Также для работы приемника потребуются следующие внутренние буферы и регистры: **state** — для хранения состояния/фазы работы автомата, **received_crc16** — для накопления рассчитываемого кода CRC16, **bit_duration** — для сохранения рассчитанного на стадии калибровки значения длительности битовго интервала, **ready** для буферизирования выходного сигнала готовности, **packet** — для хранения получаемых данных, **last_dp** — для запоминания последнего состояния шины, и **last_symbol** — для запоминания последнего принятого символа. Регистр **last_dp** требуется для выделения фронта или спада по линии **io.usb.dp** которая используется для сэмплирования данных, а смена состояния на этой линии используется для пересинхронизации и сброса таймера битового интервала **T0**. Регистр **last_symbol** необходим для преобразования получаемого потока символов в биты данных. Регистр **ones** используется для подсчете числа непрерывно следующих единиц.

Теперь, когда мы знаем назначение всех регистров и сигналов, посмотрим на код реализации вспомогательного автомата **USBReceiver** представленный ниже и выраженный на SpinalHDL:

Листинг 8.8. Класс вспомогательного автомата USBReceiver.

```
case class USBReceiver() extends USBSendReceive(hasStrobe = false, hasSendKJ = false,
hasCRC16 = true, szBitcount = 7) {
    val io = new USB_IO {
        val packet    = out Bits(128 bits) // PID(8) + DATA(64) + CRC16(16) + ALIGN
        val bits_recv = out UInt(7 bits)
        val calculated_crc16 = out Bits(16 bits)
        val received_crc16 = out Bits(16 bits)
    }

    override val ones = Reg(UInt(3 bits)).addTag(crossClockDomain)
    val received_crc16 = Reg(Bits(16 bits)).addTag(crossClockDomain) init(0)
```

```
val bit_duration = Reg(UInt(8 bits)).addTag(crossClockDomain) init(0)
val state = Reg(UInt(3 bits)).addTag(crossClockDomain) init(0)
val T0 = Reg(UInt(8 bits)).addTag(crossClockDomain) init(0) // Bit timer
val T1 = Reg(UInt(8 bits)) init(0) // EOP timer
val T2 = Reg(UInt(12 bits)) init(0) // Guard timer

val ready = Reg(Bool()).addTag(crossClockDomain) init(False)
val packet = Reg(Bits(128 bits)).addTag(crossClockDomain) init(0)
val last_dp = Reg(Bool()) init(True)
val last_symbol = Reg(Bool()) init(True)

io.packet := packet
io.bits_recv := bit_count
io.calculated_crc16 := ~crc16.reversed
io.received_crc16 := received_crc16

when(io.valid) {

  io.ready := ready

  switch(state) {

    is(0) { // SYNC: begin calibration
      when(io.usb.dp && !io.usb.dm) { // First 'K' - start calibration
        state := 1
        bit_count := 0
        ones := 0
        bit_duration := 7 // default is 8 clocks
        packet := 0
        crc16 := B"16'hFFFF"
        ready := False
        last_dp := True
        last_symbol := True
        T0 := 1 // compensation for init state which takes just one clock
        T1 := 0
        T2 := 0
      }
    }

    is(1) { // SYNC: 'K' is going, waiting for 'J'
      T0 := T0 + 1
      when(!io.usb.dp && io.usb.dm) { // 'J' received
        bit_count := bit_count + 1
        state := 2
      }
      when(T0 === 255) { // Too much, error
        state := 7
      }
    }

    is(2) { // SYNC: 'J' is going, waiting for 'K'
      T0 := T0 + 1
      when(io.usb.dp && !io.usb.dm) { // 'K' received
        bit_count := bit_count + 1
        state := 1
        when(bit_count === 3) { // TWO 'KJ' cycles received
          bit_duration := (T0 >> 2).resized // calculate bit duration: div by 4
          state := 3
          T0 := 0
        }
      }
      when(T0 === 255) { // Too much, error
        state := 7
      }
    }

    is(3) { // Wait for end of SYNC: two 'K's
      when(io.usb.dp && !io.usb.dm) { // 'K' received
        T0 := T0 + 1
        when(T0.asBits === bit_duration(6 downto 0) ## B"0") { // T0 === bit_duration*2
          T0 := 0
          state := 4
          bit_count := 0
        }
      } otherwise {
        T0 := 0
      }
```

```
          when(T0 === 255) { // Too much, error
            state := 7
          }
        }

        is(4) { // Receiving data
          when(io.usb.dp =/= io.usb.dm) { // Valid data are only when DP != DM
            T0 := T0 + 1
            last_dp := io.usb.dp
            when(last_dp =/= io.usb.dp) { // sync on each edge
              T0 := 0
            }
            when(T0 === bit_duration) { // end of symbol ?
              T0 := 0
            }
            when(T0 === bit_duration(7 downto 1).resized) { // sample one symbol in the
middle of tick
              var bit_received = (io.usb.dp === last_symbol) // convert symbol to bit
              last_symbol := io.usb.dp
              when(bit_received) {
                ones := ones + 1
              } otherwise {
                ones := 0
              }
              when(ones =/= 6) { // save current bit
                bit_count := bit_count + 1
                packet(bit_count) := bit_received
                received_crc16 := bit_received ## received_crc16(15 downto 1)
                when(bit_count > 23 && bit_count(2 downto 0) === U("000")) {
                        crc16 := calc_crc16_usb(crc16, received_crc16(7 downto 0).reversed)
                }
              } otherwise { // skip current bit because it's stuffing bit
                ones := 0
              }
              when(bit_count === 127) { // max data size achieved
                state := 7
              }
              io.test := True
            }
          }
        }

        is(6) { // EOP received
          T0 := T0 + 1
          when(T0 === bit_duration) { // wait for 'J' after SE0
            state := 7
          }
        }

        is(7) { // Ready
          ready := True
          io.bits_recv := bit_count
          io.test := True
        }

        default {
          state := 7
          packet(87 downto 80) := state.asBits.resized
          io.test := True
        }

      }

      // Check for EOP (SE0)
      when(!io.usb.dp && !io.usb.dm) {
        T1 := T1 + 1
        when(T1 === ((bit_duration << 1) - U(2))) { // is SE0 for two bit intervals - report
EOP
          state := 6
          T0 := 0
        }
      } otherwise {
        T1 := 0
      }
      // Check for hung state ('J')
      when(!io.usb.dp && io.usb.dm) {
        T2 := T2 + 1
```

```
          when(T2 === (bit_duration << 3)) { // is 'J' for more than 8 clocks - report error!
            state := 7
          }
        } otherwise {
          T2 := 0
        }

      } otherwise {
        state := 0
        ready := False
      }
  }
```

5.2.8. Класс Apb3USB10Ctrl

Класс **Apb3USB10Ctrl** будет выполнять сразу две функции: во-первых, в нем будет содержаться реализация основного конечного автомата для хост-контроллера, и во-вторых, в нем будет находиться вся связующая логика для интеграции контроллера в шину APB3 синтезируемой СнК.

Но начнем мы разбирать устройство этого класса с описания двух перечислений: **USBMain** - для перечня состояний основного автомата, и **USBCommand** — для списка поддерживаемых команд:

Листинг 8.9. Поименованные перечни состояний и команд.

```
object USBMain extends SpinalEnum{
  val StateUnconnected, StateWaitCMDorSYNC, StateKeepAlive,
      StateSendLongToken, StateSendShortToken,
      StateSendData, StateSendReset, StateReceive
      = newElement()
}

object USBCommand extends SpinalEnum(defaultEncoding = binarySequential){
  val CMDNone, CMDSendToken, CMDSendShortToken, CMDSendData, CMDBusReset
      = newElement()
}
```

На стадии компиляции состояниям **StateUnconnected**, **StateWaitCMDorSYNC** и т. д. Будет присвоены константы начиная с нуля, это аналогично тому, как работает оператор **enum** в языках C/C++. Все тоже самое с перечнем команд. Команде **CMDNone** будет присвоена константа 0, **CMDSendToken** = 1, **CMDSendShortToken** = 2 и т. д.

В этом месте стоит сделать небольшое замечание по стилю. Пытливый читатель может задать справедливый вопрос: почему для автомата **USBMain** в коде хост-контроллера используются поименованные состояния, а для всех вспомогательных КА используются только числовые константы ? Ответ простой — нет смысла плодить лишние сущности. У нас всего два вспомогательных автомата с выделенной переменной (регистром **state**) для хранения состояния, это автоматы **USBSendData** и **USBReceiver**. В обоих автоматах переход из одного состояния в другое производится последовательно с увеличением номера фазы, в них нет произвольных переходов между разными состояниями, в каждом состоянии автомат пребывает только один раз. Поэтому мне показалось удобным использовать последовательно пронумерованные числовые состояния. В главном конечном автомате **USBMain** логика переходов между различными состояниями может быть весьма запутанной, поэтому все состояния этого автомата пронумерованы и им присвоены осмысленные названия.

Класс **Apb3USB10Ctrl** является аппаратным компонентом (сложным функциональным блоком), а значит наследуется от класса **Component** являющегося частью библиотеки SpinalHDL. Конструктор класса принимает один параметр **usbFrequency**

задающий частоту основного тактового сигнала от которого тактируются компоненты контроллера. По умолчанию будем использовать удобную нам частоту 12 МГц. Структура комплексного сигнала **io**, описывающая внешние интерфейсы этого компонента, будет содержать следующие сигналы и шины:

- Шина **usb** — описывается интерфейсным классом **USBInterface** инкапсулирующим два сигнала **usb_dp** и **usb_dm** шины USB. Компонент **Apb3USB10Ctrl** является мастером на шине, поэтому прогоним конструктор класса **USBInterface()** через библиотечную функцию **master()**, она перенастроит дефолтное направления сигналов на шине соответствующим образом и вернет «модифицированный» интерфейсный класс.
- Шина **apb** — описывается интерфейсным классом **Apb3**, содержит все необходимые сигналы для подключения компонента к шине APB3. Класс **Apb3** является частью библиотеки SpinalHDL, Так как компонент **Apb3USB10Ctrl** будет являться периферийным устройством на шине APB3, то перенастроим шину соответственно прогнав конструктор **Apb3()** через библиотечную функцию **slave()**. В конструктор Apb3() при этом укажем параметры задающие размер адресного пространства компонента и ширину шины данных: **addressWidth = 12**, **dataWidth = 32**.
- Входной сигнал **usb_clk** для тактирования компонентов шины (12 МГц) сформированный где-то уровнем выше в архитектурной иерархии СнК. Далее, на стадии интеграции, мы посмотрим как этот сигнал формируется в «KarnixSoC».
- Выходной одно-битовый сигнал **interrupt** для формирования запроса прерывания в контроллер прерываний.
- Выходной одно-битовый сигнал test который будем использовать в целях отладки для проброса на физический вывод микросхемы ПЛИС любого внутреннего одно-битового сигнала или флага. Это даст возможность подсмотреть за поведением логики с помощью анализатора сигналов или осциллографа.

Всё выше сказанное излагается на SpinalHDL в несколько строк. Ниже приведен участок кода заголовка для класса **Apb3USB10Ctrl** описывающий интерфейс компонента.

Листинг 8.10. Класс Apb3USB10Ctrl: заголовок.

```
case class Apb3USB10Ctrl(usbFrequency : HertzNumber = 12.0 MHz) extends Component {
  val io = new Bundle {
    val apb       = slave(Apb3(addressWidth = 12, dataWidth = 32))
    val usb       = master(USBInterface())
    val interrupt = out Bool()
    val usb_clk   = in Bool()
    val test      = out Bool()
  }
...
```

Следующим шагом будет подключение к шине APB3. Делается это одной строкой кода:

```
val busCtrl = Apb3SlaveFactory(io.apb)
```

Эта строка создает нам объект **busCtrl** класса **Apb3SlaveFactory** который помимо того, что сам является электронным компонентом, предоставляет набор функций и процедур для работы с шиной APB3, в частности - подключение видимых вычислительному ядру (и программисту) регистров.

Воспользуемся свойствами **busCtrl**, опишем все регистры программного API нашего хост-контроллера (см. главу 4.3. «*Описание регистров и команд*») и, для удобства

дальнейшей работы, разобьем регистры на отдельные поименованные поля, которые будут представлять собой сигналы разной битности:

Листинг 8.11. Класс Apb3USB10Ctrl: объявление программно доступных регистров.

```
  // 0x00 - STATUS
   val usbStatusWord = busCtrl.createReadOnly(Bits(32 bits), address = 0) init(0)
   val error_flag = usbStatusWord(30).addTag(crossClockDomain)
   val report_flag = usbStatusWord(29).addTag(crossClockDomain)
   val busy_flag = usbStatusWord(28).addTag(crossClockDomain)
   val received_flag = usbStatusWord(27).addTag(crossClockDomain)
   val crc16_ok_flag = usbStatusWord(26).addTag(crossClockDomain)
   // ... more flags here
   //val received_pid = usbStatusWord(23 downto 16).addTag(crossClockDomain)
   // ... reserved for FSM states
   val fsm_state = usbStatusWord(2 downto 0).addTag(crossClockDomain)

   // 0x04 - COMMAND
   val usbCommandWord = busCtrl.createReadWrite(Bits(32 bits), address = 4) init(0)
   val cmd_start = usbCommandWord(31).addTag(crossClockDomain)
   val cmd_addr = usbCommandWord(30 downto 24).addTag(crossClockDomain)
   val cmd_endp = usbCommandWord(23 downto 20).addTag(crossClockDomain)
   val cmd_len = usbCommandWord(14 downto 8).asUInt.addTag(crossClockDomain) // len in bits -
1
   val cmd_pid = usbCommandWord(7 downto 4).addTag(crossClockDomain)
   val cmd = usbCommandWord(3 downto 0).addTag(crossClockDomain)

   // 0x08 - RECV_DATA_LOW
   val usbDataReceivedLowWord = busCtrl.createReadOnly(Bits(32 bits), address = 8) init(0)
   val received_data_low = usbDataReceivedLowWord(31 downto 0).addTag(crossClockDomain)

   // 0x0C - RECV_DATA_HIGH
   val usbDataReceivedHighWord = busCtrl.createReadOnly(Bits(32 bits), address = 12) init(0)
   val received_data_high = usbDataReceivedHighWord(31 downto 0).addTag(crossClockDomain)

   // 0x10 - SEND_DATA_LOW
   val usbSendLowWord = busCtrl.createReadWrite(Bits(32 bits), address = 16) init(0)
   val send_data_low = usbSendLowWord(31 downto 0).addTag(crossClockDomain)

   // 0x14 - SEND_DATA_HIGH
   val usbSendHighWord = busCtrl.createReadWrite(Bits(32 bits), address = 20) init(0)
   val send_data_high = usbSendHighWord(31 downto 0).addTag(crossClockDomain)

   // 0x18 - RX_STATUS
   val usbReceiverStatusWord = busCtrl.createReadOnly(Bits(32 bits), address = 24) init(0)
   val received_bits = usbReceiverStatusWord(7 downto 0).addTag(crossClockDomain)
   val received_pid = usbReceiverStatusWord(15 downto 8).addTag(crossClockDomain)
   val received_crc16 = usbReceiverStatusWord(31 downto 16).addTag(crossClockDomain)

   // 0x1C - CONTROL
   val usbControlWord = busCtrl.createReadWrite(Bits(32 bits), address = 28) init(22500) //
Reset delay: 15 ms at 1.5 Mbit/sec
   val bus_enable = usbControlWord(31).addTag(crossClockDomain)
   val keepalive_enable = usbControlWord(30).addTag(crossClockDomain)
   val reset_delay_bits = usbControlWord(15 downto 0).asUInt.addTag(crossClockDomain)

   // 0x20 - RX_STATUS2
   val usbReceiverStatusWord2 = busCtrl.createReadOnly(Bits(32 bits), address = 32) init(0)
   val calculated_crc16 = usbReceiverStatusWord2(15 downto 0).addTag(crossClockDomain)
```

Метод (функция) **createReadOnly()** класса **Apb3SlaveFactory** создает и подключает к шине APB3 регистр заданной разрядностью и с заданным в параметре **address** смещением в адресном пространстве относительно начала блока выделенного для данного компонента (относительно базового адреса). Данный регистр будет доступен вычислительному ядру только на чтение. Привязка компонента **Apb3USB10Ctrl** к шинам вычислительного ядра будет выполнена уровнем выше.

Метод **createReadWrite()** класса **Apb3SlaveFactory** создает регистр доступный вычислительному ядру как на чтение, так и на запись.

Метод **init()** у регистра позволяет задать его начальное (по сбросу) значение. Почти для всех регистров API, кроме регистра **usbControlWord** (0x1C - CONTROL) устанавливается нулевое значение. Для регистра **usbControlWord** задается дефолтное значение длительности сигнала «Bus Reset» выраженное в количестве битовых интервалов при скорости шины USB равной 1,5 МБит/сек.

Вызов метода **addTag(crossClockDomain)** у регистра или сигнала поясняет библиотеке SpinalHDL, что доступ к нему может осуществляться из различных тактовых доменов, а следовательно при генерации аппаратуры необходимо это учитывать и, по возможности, добавлять механизмы CDC. Все регистры и сигналы программного интерфейса USB хост-контроллера будут использоваться как из системного тактового домена (60 МГц для платы «Карно»), в котором работает вычислительное ядро, так и из своего внутреннего (12 МГц), поэтому все они помечены тэгом **crossClockDomain**.

Следующим шагом с помощью класса **ClockDomain** создадим и опишем тактовый домен для работы хост-контроллера. Этот тактовый домен будет тактироваться от сигнала передаваемого по интерфейсной линии **io.usb_clk**, а сигнал сброса будет порождаться битом **bus_enable** регистра **usbControlWord** (31-й бит регистра 0x1C — CONTROL):

Листинг 8.12. Класс Apb3USB10Ctrl: объявление тактового домена usbClockDomain.

```
val usbClockDomain = ClockDomain(
  clock = io.usb_clk,
  reset = bus_enable,
  config = ClockDomainConfig(resetKind = SYNC, resetActiveLevel = LOW),
  frequency = FixedFrequency(usbFrequency)
)
```

Теперь с помощью класса **ClockingArea** опишем область аппаратуры которая будет входить в этот тактовый домен:

```
val usb_area = new ClockingArea(usbClockDomain) {

  // … здесь разместится весь код реализующий основной конечный автомат USBMain

}
```

Между фигурными скобками, то есть в блоке имплементации класса **ClockingArea**, будем помещать весь код основного КА нашего хост-контроллера.

5.2.9. Реализация основного конечного автомата USBMain

Как уже стало понятно, код основного КА хост-контроллера располагается в своем тактовом домене и получать тактирование от входного интерфейсного сигнала **io.usb_clk**. Все вспомогательные КА будут находиться в этом же тактовом домене и унаследуют его тактовый сигнал и сигнал сброса.

Перед реализацией конечного автомата **USBMain** определим три параметра необходимых для его функционирования, значения которых вычислим средствами языка Scala:

- Переменная **usbslowspeedclockdiv** и сигнал **USBSlowSpeedClockDiv** (8 бит) — содержат рассчитанное число тактов делителя частоты для формирования сигнала строба битового интервала во вспомогательных конечных автоматах. Строб должен

формироваться с частотой равной частоте работы шины USB, в нашем случае — 1,5 МГц. Значение этого параметра будет передаваться на вход вспомогательным КА для использования в делителе частоты **clock_div**.

- Переменна **usblowspeedkeepaliveclocks** и сигнал **USBLowSpeedKeepAliveClocks** (16 бит) — содержат рассчитанное число битовых интервалов для формирования сигнала «KeepAlive» выраженное в битовых интервалах. Согласно спецификации USB 1.0 «Low Speed», сигнал «KeepAlive» должен появляться на шине не реже чем каждую 1 мс. Нам необходимо знать сколько тактов базовой частоты текущего тактового домена это составляет.

- Переменная **usblowspeederrorclocks** и сигнал **USBLowSpeedErrorClocks** (16 бит) — содержат рассчитываемое число тактов текущего тактового домена в течении которых присутствие на шине USB состояния «SE0» будет рассматриваться как аварийное (время детектирования отключения устройства). В спецификации нет конкретных рекомендаций на сей счет, по этому выберем интервал близкий к периоду «KeepAliave», т. е. равное 0,8 мс.

Все три параметра используют входной параметр **usbFrequency**. В процессе вычислений будем выводить рассчитанные значения на консоль с помощью библиотечной функции **println()**. Мы сможем наблюдать за выводом на стадии генерации Verilog при исполнении SpinalHDL программы в Java VM. Код для расчета этих параметров приведен ниже:

Листинг 8.13. Класс Apb3USB10Ctrl: расчет параметров.

```
    println("Apb3USB10Ctrl::usbFrequency = %d Hz".format(usbFrequency.toInt));

    val USBSlowSpeedClockDiv = UInt(8 bits)
    val low_speed_baudrate : HertzNumber = 1.5 MHz;
    val usbslowspeedclockdiv = (ClockDomain.current.frequency.getValue / low_speed_baudrate +
0.5).toBigInt - 1
    USBSlowSpeedClockDiv := usbslowspeedclockdiv
    println("Apb3USB10Ctrl::USBSlowSpeedClockDiv = %d".format(usbslowspeedclockdiv));

    val USBLowSpeedKeepAliveClocks = UInt(16 bits)
    val low_speed_keepalive : TimeNumber = 1.0 ms; // Send KeepAlive interval
    val usblowspeedkeepaliveclocks = (ClockDomain.current.frequency.getValue * low_speed_keepalive +
0.5).toBigInt - 1
    USBLowSpeedKeepAliveClocks := usblowspeedkeepaliveclocks
    println("Apb3USB10Ctrl::USBLowSpeedKeepAliveClocks = %d".format(usblowspeedkeepaliveclocks));

    val USBLowSpeedErrorClocks = UInt(16 bits)
    val low_speed_error : TimeNumber = 0.8 ms; // Time to detect disconnect or error
    val usblowspeederrorclocks = (ClockDomain.current.frequency.getValue * low_speed_error +
0.5).toBigInt - 1
    USBLowSpeedErrorClocks := usblowspeederrorclocks
    println("Apb3USB10Ctrl::USBLowSpeedErrorClocks = %d".format(usblowspeederrorclocks));
```

Теперь перейдем собственно к описанию аппаратуры основного конечного автомата. Сначала объявим регистр состояния **state** и два таймера **T1** и **T2**, после чего добавим объявление одно-битовых флаговых регистров используемых внутри КА:

Листинг 8.14. Класс Apb3USB10Ctrl: объявление внутренних регистров и таймеров.

```
    val state = RegInit(StateUnconnected).addTag(crossClockDomain)
    val T1 = Reg(UInt(16 bits)).addTag(crossClockDomain) init(0) // Guard timer
    val T2 = Reg(UInt(16 bits)).addTag(crossClockDomain) init(0) // Low-Speed Keep-Alive timer
```

```
val error = Reg(Bool()).addTag(crossClockDomain) init(False)
val report = Reg(Bool()).addTag(crossClockDomain) init(False)
val busy = Reg(Bool()).addTag(crossClockDomain) init(False)
val received = Reg(Bool()).addTag(crossClockDomain) init(False)
val crc16_ok = Reg(Bool()).addTag(crossClockDomain) init(False)
```

Транслируем значения этих внутренних регистров в битовые поля регистров доступных пользователю, которые мы описали в предыдущей главе при подключении к шине APB3:

Листинг 8.15. Класс Apb3USB10Ctrl: трансляция значений внутренних регистров в программно доступные регистры.

```
error_flag := error
report_flag := report
received_flag := received
busy_flag := busy
crc16_ok_flag := crc16_ok
fsm_state := state.asBits
```

Подвяжем флаг **report_flag** к сигналу ведущему к контроллеру прерываний:

```
io.interrupt := report_flag
```

Далее в тексте реализации будем устанавливать внутренний флаг **report** всякий раз когда необходимо оформить программное прерывание.

Объявим все вспомогательные автоматы и присвоим дефолтные значения для их входных сигналов. Сигналы ***.io.valid** установим по умолчанию в состояние **False**, то есть вспомогательные КА не активны. Также передадим на входы ***.io.clock_div** значение рассчитанного выше параметра **USBSlowSpeedClockDiv**:

Листинг 8.16. Класс Apb3USB10Ctrl: подключение вспомогательных КА

```
val send_long_token = new USBSendLongToken()
send_long_token.io.pid := 0
send_long_token.io.addr := 0
send_long_token.io.endp := 0
send_long_token.io.valid := False
send_long_token.io.clock_div := USBSlowSpeedClockDiv

val send_short_token = new USBSendShortToken()
send_short_token.io.pid := 0
send_short_token.io.valid := False
send_short_token.io.clock_div := USBSlowSpeedClockDiv

val send_data = new USBSendData()
send_data.io.pid := 0
send_data.io.data := 0
send_data.io.len := 0x7f // zero data bits
send_data.io.valid := False
send_data.io.clock_div := USBSlowSpeedClockDiv

val send_se0 = new USBSendSE0()
send_se0.io.valid := False
send_se0.io.len := 0
send_se0.io.clock_div := USBSlowSpeedClockDiv

val receiver = new USBReceiver()
receiver.io.valid := False
```

Опишем вспомогательные сигналы представляющие различные состояния шины USB:

Листинг 8.17. Класс Apb3USB10Ctrl: объявление вспомогательных сигналов.

```
val bus_error = !io.usb.dm && !io.usb.dp; // Both DP and DM low means nothing is connected
val bus_present = io.usb.dm && !io.usb.dp; // Low Speed device connected
val bus_activity = !io.usb.dm && io.usb.dp; // Polarity changed to 'K' indicates some activity
```

Реализуем защитный таймер **T1** который служит для перевода основной машины в состояние **StateUnconnected** если на шине детектируется состояние bus_error и при этом сам автомат не проводит сброс:

Листинг 8.18. Класс Apb3USB10Ctrl: реализация защитного таймера T1.

```
// Guard timer T1 checks for error state on the bus

when(bus_error && state =/= StateSendReset) {
  T1 := T1 + 1
  when(T1 === USBLowSpeedErrorClocks) { // DM/DP is low for quite some time ?
    T1 := 0
    state := StateUnconnected
    error := True
    report := True
    received := False
    busy := False
    cmd := CMDNone.asBits.resized // clear last cmd
    cmd_start := False
  }
} otherwise {
  T1 := 0
}
```

Теперь приступим к обработке событий в различных состояниях. Первым делом обработаем состояние **StateUnconnected**. Напомню, что в этом состоянии основной КА находится после сброса и попадает в него при отсутствии на шине USB устройств. Из этого состояния КА переходит только в состояние **StateWaitCMDorSYNC** при условии если установлен флаг **bus_present**. Во всех случаях, при смене состояния поднимается флаг **report** который связан сигналом для контроллера прерываний.

Листинг 8.19. Класс Apb3USB10Ctrl: обработка состояния StateUnconnected.

```
// Main FSM

switch(state) {

  is(StateUnconnected) { // Unconnected
    report := False

    when(bus_present) {
      busy := False
      cmd_start := False
      error := False
      report := True
      state := StateWaitCMDorSYNC // Low Speed device just connected
    }
  }
```

Обработаем состояние **StateWaitCMDorSYNC**. В нем основной КА может исполнять команды подаваемые пользователем в битовом поле **cmd** и сопровождая их установкой бита **cmd_start**. Также в этом состоянии основной КА всегда сбрасывает сигнал **report**, таким образом предполагается что это пассивное состояние машины не требующее внимания программы.

Листинг 8.20. Класс Apb3USB10Ctrl: обработка состояния StateWaitCMDandSYNC.

```
is(StateWaitCMDorSYNC) { // Wait command or SYNC
  report := False

  when(cmd_start) {
    switch(cmd) {
      is(CMDSendToken.asBits.resize(4)) {
        state := StateSendLongToken
        busy := True
        received := False
      }
      is(CMDSendShortToken.asBits.resize(4)) {
        state := StateSendShortToken
        busy := True
        received := False
      }
      is(CMDSendData.asBits.resize(4)) {
        state := StateSendData
        busy := True
        received := False
      }
      is(CMDBusReset.asBits.resize(4)) {
        state := StateSendReset
        busy := True
        received := False
      }
      default {
        cmd_start := False
        report := True
        received := False
      }
    }
  }
```

В состоянии **StateWaitCMDorSYNC** основной КА может принимать входные пакет если обнаружена активность на шине USB:

```
  when(bus_activity) { // Activity on the bus ?
    state := StateReceive
    busy := True
    received := False
  }
```

Или выполнять процедуру «KeepAlive» если она разрешена битом **keepalive_enable** (бит регистра управления) и на шине USB нет ошибки:

```
  when(keepalive_enable && !(bus_error)) { // Keepalive enabled and not Error state ?
    T2 := T2 + 1

    when(T2 === USBLowSpeedKeepAliveClocks) {
      state := StateKeepAlive
      busy := True
    }
  }

} // end of is(StateWaitCMDorSYNC)
```

В состоянии **StateSendLongToken** основной КА подает необходимые данные на вход вспомогательного автомата **USBSendLongToken** представленного переменной **send_long_token**, коммутирует шину USB соединяя сигналы **send_long_token.io.usb** и **io.usb**, подает сигнал активации **send_long_token.io.valid** и ждет сигнала готовности **send_long_token.io.ready**. По готовности, основной КА возвращается в состояние **StateWaitCMDorSYNC** подымая флаг **report** для генерации программного прерывания:

Листинг 8.21. Класс Apb3USB10Ctrl: обработка состояния StateSendLongToken.

```
is(StateSendLongToken) { // Connected, send Token
  send_long_token.io.pid := cmd_pid
  send_long_token.io.addr := cmd_addr
  send_long_token.io.endp := cmd_endp
  send_long_token.io.valid := True
  send_long_token.io.usb <> io.usb
  when(send_long_token.io.ready) {
    state := StateWaitCMDorSYNC
    report := True
    cmd_start := False
    busy := False
    T2 := 0
  }
}
```

Аналогичное поведение у основного КА для всех остальных состояний ассоциированных с активацией вспомогательного автомата.

Листинг 8.22. Класс Apb3USB10Ctrl: обработка остальных состояний для вспомогательных КА.

```
is(StateSendShortToken) { // Connected, send Short Token
  send_short_token.io.pid := cmd_pid
  send_short_token.io.valid := True
  send_short_token.io.usb <> io.usb
  when(send_short_token.io.ready) {
    state := StateWaitCMDorSYNC
    report := True
    cmd_start := False
    busy := False
    T2 := 0
  }
}

is(StateSendData) { // send DATA packet
  send_data.io.pid := cmd_pid
  send_data.io.data := send_data_high ## send_data_low
  send_data.io.len := cmd_len // in bits - 1
  send_data.io.valid := True
  send_data.io.usb <> io.usb
  when(send_data.io.ready) {
    state := StateWaitCMDorSYNC
    report := True
    cmd_start := False
    busy := False
    T2 := 0
  }
}

is(StateSendReset) { // Bus Reset condition is SE0 (D+ and D- are low) for 11ms
  send_se0.io.valid := True
  send_se0.io.usb <> io.usb
  send_se0.io.len := reset_delay_bits
  when(send_se0.io.ready) {
    state := StateWaitCMDorSYNC
    report := True
    cmd_start := False
    busy := False
    T2 := 0
  }
}

is(StateKeepAlive) { // KeepAlive (Low-Speed only) is SE0 for just two bit intervals
  send_se0.io.valid := True
  send_se0.io.usb <> io.usb
  send_se0.io.len := 2
  when(send_se0.io.ready) {
    state := StateWaitCMDorSYNC
    cmd_start := False
    busy := False
```

```
        T2 := 0
      }
    }

    is(StateReceive) { // Receive data piece
      receiver.io.valid := True
      receiver.io.usb <> io.usb
      when(receiver.io.ready) {
        state := StateWaitCMDorSYNC
        received := True
        busy := False
        report := True
        received_pid := receiver.io.packet(7 downto 0)
        received_data_low := receiver.io.packet(39 downto 8)
        received_data_high := receiver.io.packet(71 downto 40)
        received_bits := receiver.io.bits_recv.asBits.resized
        received_crc16 := receiver.io.received_crc16 //receiver.io.packet(87 downto 72)
        calculated_crc16 := receiver.io.calculated_crc16
        crc16_ok := receiver.io.received_crc16 === receiver.io.calculated_crc16
      }
    }

  } // switch(state)
```

Для состояния **StateReceive** после получения сигнала готовности **receiver.io.ready** производится копирование принятых данных, полученного и рассчитанного кодов CRC16, а так же флага **crc16_ok** во внутренние буферы, из которого эти данные попадут в программно доступные регистры (см. трансляцию сигналов выше).

И, собственно всё! На этом заканчивается реализация основного автомата и всего USB хост-контроллера. Далее мы рассмотрим как этот код подключить к существующей синтезируемой СнК и напишем простейший драйвер к нему.

5.3. Интеграция кода хост-контроллера в СнК и сборка проекта

Синтезируемая СнК «KarnixSoC», с которой я работаю (как и вычислительное ядро VexRiscv на базе которого она построена) полностью написана на SpinalHDL и является частью репозитория проекта SpinalHDL, пока что только сопровождаемой мной неофициальной ветки с одноименным названием. Клонировать репозиторий с поддержкой «KarnixSoC» можно с Github-а следующей командой:

```
$ git clone https://github.com/pointcheck/KarnixSOC.git
```

В внутри дерева каталогов этого репозитория весь код на Scala/SpinalHDL располагается в подкаталоге **./src/main/scala**. В этом подкаталоге я завел отдельный подкаталог **mylib** для своих наработок. Весь код USB 1.0 хост-контроллера описанный в главе 5.2 собран в один файл **Apb3USB10.scala** внутри **mylib**, за исключением интерфейсного класса **USBInterface** код которого вынесен в отдельный файл **USBInterface.scala**.

Листинг 8.23. Перечень файлов аппаратуры входящих в СнК «KarnixSoC».

```
rz@devbox:~$ cd KarnixSOC/

rz@devbox:~/KarnixSOC$ ls -l src/main/scala/mylib/
total 156
-rw-rw-r-- 1 rz rz  1255 Oct  7 20:47 Apb3MacEthCtrl.scala
-rw-rw-r-- 1 rz rz  1216 Oct  7 20:47 Apb3Timer.scala
-rw-rw-r-- 1 rz rz 28434 Oct  7 20:47 Apb3USB10.scala
-rw-rw-r-- 1 rz rz   911 Oct  7 20:47 Apb3WatchDog.scala
-rw-rw-r-- 1 rz rz 18846 Oct  7 20:47 Axi4SharedToQSPI.scala
-rw-rw-r-- 1 rz rz  4694 Oct  7 20:47 Axi4SharedToSRAM.scala
-rw-rw-r-- 1 rz rz 15873 Oct  7 20:47 CGA4HDMI.scala
-rw-rw-r-- 1 rz rz  4121 Oct  7 20:47 HDMI.scala
-rw-rw-r-- 1 rz rz 18688 Oct  7 20:47 HUB.scala
-rw-rw-r-- 1 rz rz  6874 Oct  7 20:47 KarnixTestHDMI.scala
-rw-rw-r-- 1 rz rz   952 Oct  7 20:47 MachineTimer.scala
-rw-rw-r-- 1 rz rz  6301 Oct  7 20:47 MicroI2C.scala
-rw-rw-r-- 1 rz rz  1932 Oct  7 20:47 MicroPLIC.scala
-rw-rw-r-- 1 rz rz  1034 Oct  7 20:47 PWM.scala
-rw-rw-r-- 1 rz rz  3231 Oct  7 20:47 Sram.scala
-rw-rw-r-- 1 rz rz   275 Oct  7 20:47 USBInterface.scala
```

Файл **Apb3USB10.scala** имеет зависимости только от стандартного набора библиотек SpinalHDL и содержит менее 900 строк кода:

```
rz@devbox:~/KarnixSOC$ wc src/main/scala/mylib/Apb3USB10.scala
  882  3278 28434 src/main/scala/mylib/Apb3USB10.scala
```

Файл **USBInterface.scala** содержит всего несколько уже знакомых строк:

Листинг 8.24. Код файла USBInterface.scala.

```
rz@devbox:~/KarnixSOC$ cat src/main/scala/mylib/USBInterface.scala

package mylib

import spinal.core._
import spinal.lib._
import spinal.lib.io.TriState

case class USBInterface() extends Bundle with IMasterSlave{
  val dm = inout(Analog(Bool()))
  val dp = inout(Analog(Bool()))
```

```
    override def asMaster(): Unit = {
      inout(dm, dp)
    }
  }
```

Выделение кода интерфейсного класса **USBInterface** в отдельный файл сделано для того, чтобы этот интерфейсный класс можно было импортировать (подключать в помощью оператора **import**) и использовать в других местах, в том числе в главном файле описывающем СнК «KarnixSoC» расположенном в другом месте:

```
rz@devbox:~/KarnixSOC$ ls -l src/main/scala/vexriscv/demo/KarnixSOC.scala
-rw-rw-r-- 1 rz rz 33226 Oct  7 20:47 src/main/scala/vexriscv/demo/KarnixSOC.scala

rz@devbox:~/KarnixSOC$ wc src/main/scala/vexriscv/demo/KarnixSOC.scala
  968  3020 33226 src/main/scala/vexriscv/demo/KarnixSOC.scala
```

Для того, чтобы интегрировать компонент USB хост-контроллера в СнК и протестировать его, необходимо выполнить следующие действия:

- Физически подключить USB разъем типа A к плате и соединить сигналы D+ и D- со свободными выводами микросхемы ПЛИС.
- Добавить сигналы D+ и D- шины USB в файл конфигурации ПЛИС и ассоциировать их с физическими выводами микросхемы.
- Добавить описание шины USB в интерфейс класса **KarnixSOC** описывающего структуру СнК и его внешние сигналы.
- Внутри СнК создать сигнал тактирования **usb_clk** и сформировать его с помощью PLL (либо методом деления основной частоты — это тоже работает).
- Создать объект класса **Apb3USB10Ctrl** и, с одной стороны, подключить его к шине APB3, а с другой — к интерфейсу шины USB и сигналу тактирования.

Далее рассмотрим детально каждый из этих шагов, после чего проведем сборку (синтез) СнК.

5.3.1. Подключение разъема USB к плате «Карно»

На плате «Карно» мной были выбраны два свободных вывода на разъеме GPIO для сигналов D+ и D- шины USB. Так получилось, что это оказались выводы GPIO_22 (D-) и GPIO_23 (D+). Для платы «Карно» на скорую руку была изготовлена макетная плата расширения содержащая разъем USB type A. К выводам 2 и 3 USB разъема были подпаяны проводники из МГТФ соединяющие их с линиями GPIO_22 и GPIO_23 соответственно. Вывод 1 соединен с линией питания +5 В, вывод 5 — с «землей» (GND). Рядом в разъемом на макетной плате были запаяны два резистора по 15К подтягивающие сигналы D+ и D- к «земле» в соответствии со схемой приведенной на рис 4. Тут же был установлен один развязывающий конденсатор 4.7 мкФ в линию питания +5 В (VBUS).

Вся эта конструкция приведена на фотографии рис. 14. На фото видны два проводника, белый и желтый, выходящие за пределы платы. Это отводы от сигналов D+ и D- предназначенные для подключения приборов (анализатора сигналов).

Позже выяснилось, что резисторы подтяжек не требуются, так как внутри микросхемы ПЛИС Lattice ECP5 по умолчанию все выводы имеют «слабую» подтяжку к «земле» и шина USB в режиме «Low Speed» замечательно работает без них. Получается, что для того, чтобы начать работать с USB устройствами в ПЛИС этой серии достаточно припаять четыре проводника — два сигнальных, +5 В и «земли».

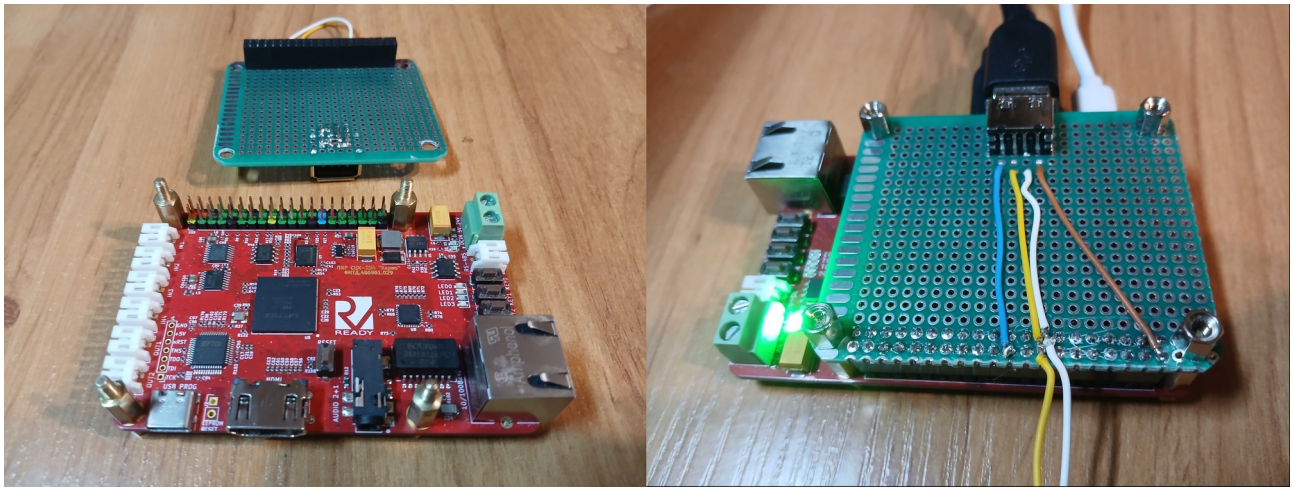

Рис. 14. Плата «Карно» и макетная плата с USB разъемом: слева — раздельно, справа — макетная плата установлена на HEX-стойки и подключена к разъему GPIO.

5.3.2. Модификация файла конфигурации ПЛИС

Для платы «Карно», содержащей микросхему ПЛИС Lattice ECP5-25F исполненной в корпусе caBGA256, внутри репозитория создан отдельный подкаталог **./scripts/KarnixSOC/ECP5-25F_karnix_board/** содержащий файл конфигурации **karnix_cabga256.lpf** для этой ПЛИС и файл сборщика **Makefile**. В этом же подкаталоге ведется сборка, в него кладутся временный и результирующие бинарные файлы.

Отредактируем конфигурационный файл **karnix_cabga256.lpf** добавив в самый конец следующие строки:

```
LOCATE COMP "io_usb1_dm" SITE "R4";     # USB1_DM/GPIO_22
IOBUF PORT "io_usb1_dm" IO_TYPE=LVCMOS33;
LOCATE COMP "io_usb1_dp" SITE "T3";     # USB1_DP/GPIO_23
IOBUF PORT "io_usb1_dp" IO_TYPE=LVCMOS33;
```

Эти строки привяжут внутренние сигналы **io.usb1.dm** и **io.usb1.dp** к внешним выводам **R4** и **T3** микросхемы ПЛИС, которые в свою очередь выведены на плате «Карно» на разъем GPIO, и переведут эти выводы ПЛИС в режим LVCMOS33.

5.3.3. Добавление интерфейса шины USB в СнК «KarnixSoC»

Как уже было отмечено выше, все описание СнК «KarnixSoC» находится в одном файле **src/main/scala/vexriscv/demo/KarnixSOC.scala**. Отредактируем это файл следующим образом:

Сначала добавим в заголовок файла подключение новых программных модулей (с точки зрения компилятора Scala весь наш код на SpinalHDL это программа):

```
import mylib.{USBInterface, Apb3USB10Ctrl}
```

Теперь найдем в коде СнК описание класса **KarnixSOC** и в структуру **io** добавим две новые переменные: одну представляющую шину USB, вторую — для сигнала тактирования этой шины. Добавляемый код (выделен жирным) будет выглядеть так:

Листинг 8.25. Код файла KarnixSOC.scala.

```
class KarnixSOC(val config: KarnixSOCConfig) extends Component{
  ...
  val io = new Bundle {
    //Clocks / reset
    val asyncReset = in Bool()
    val mainClk = in Bool()

    //Peripherals IO
    val gpioA = master(TriStateArray(32 bits))
    val gpioB = master(TriStateArray(16 bits))
    ...

    val usb1 = master(USBInterface())
    val usb_clk = in Bool()

  }
...
```

5.3.4. Добавление компонента хост-контролера в СнК

В теле класса **KarnixSoC** добавим переменную **usb1Ctrl** типа **Apb3USB10Ctrl** описывающую компонент хост-контроллера. Для этого найдем в теле класса секцию с описанием моста **Axi4SharedToApb3Bridge** и добавим следующий код после него. Также добавим привязку адресного пространства в декодер адресов шины APB3.

```
class KarnixSOC(val config: KarnixSOCConfig) extends Component{
 …
  val io = new Bundle {

  }

  val axi = new ClockingArea(axiClockDomain) {
  ...
    val apbBridge = Axi4SharedToApb3Bridge(
      addressWidth = 20,
      dataWidth    = 32,
      idWidth      = 4
    )

    val usb1Ctrl = new Apb3USB10Ctrl(usbFrequency)
    io.usb1 <> usb1Ctrl.io.usb
    usb1Ctrl.io.usb_clk := io.usb_clk
    plic.setIRQ(usb1Ctrl.io.interrupt, 10)

    …
    val apbDecoder = Apb3Decoder(
      master = apbBridge.io.apb,
      slaves = List(
        ...
        qspi0.io.apb -> (0xC2000, 4 kB),

        usb1Ctrl.io.apb -> (0xD1000, 4 kB) // USB 1.0 implemented in SpinalHDL
      )
    )
  }
```

В добавляемом выше коде происходит следующее:

1. Создается объект класса **Apb3USB10Ctrl** содержащий компонент хост-контроллера на который ссылается переменная **usb1Ctrl**.
2. С помощью оператора <> производится коммутация сигналов шины USB между компонентом хост-контроллера **usb1Ctrl.io.usb** и внешним интерфейсом **io.usb1** текущего класса **KarnixSoC**.
3. Производится подача тактового сигнала с внешнего интерфейса **io.usb_clk** на тактовый сигнал компонента **usb1Ctrl.io.usb_clk**.
4. Производится привязка сигнала линии прерываний от компонента хост-контроллера на 10-й канал контроллера прерываний.
5. Привязка шины APB3 хост-контроллера **usb1Ctrl.io.apb** к шине APB3 моста **apbBridge.io.apb**, при этом задается адрес **0xD1000** представляющий собой смещение относительно начала адресного пространства выделенного под мост APB3.

В рамках **KarnixSoC** под устройства на шине APB3 выделено адресное пространство емкостью 1МБ начиная с 0xF0000000. Таким образом, базовый адрес области регистров USB хост-контроллера вычисляется как: **0xF0000000 + 0xD1000 = 0xF00D1000**. Начиная с этого адреса будут располагаться программно доступные регистры управления хост-контроллером.

5.3.5. Вывод шины USB наружу

Выше мы уже описали комплексный сигнал **io.usb** в классе **KarnixSoC**, инстанциировали и подключили к нему наш хост-контроллер. Осталось соединить этот комплексный сигнал с внешним миром. Для этого найдем объявление класса **KarnixSOCTopLevel** и аналогичным образом добавим в структуру **io** описание сигнала **usb** определяемого интерфейсным классом **USBInterface**:

```
case class KarnixSOCTopLevel() extends Component{
    val io = new Bundle {
        val clk25 = in Bool()
        val uart_debug_txd = out Bool() // mapped to uart_debug_txd
        val uart_debug_rxd = in Bool() // mapped to uart_debug_rxd
        …

        val usb1 = master(USBInterface())
    }
```

Класс **KarnixSOCTopLevel** содержит переменную **karnix_soc** типа **KarnixSoC** которая в свою очередь содержит весь наш СнК. Добавим коммутацию двух комплексных сигналов **karnix_soc.io.usb1** и **io.usb1** одной строкой кода:

```
karnix_soc.io.usb1 <> io.usb1
```

Это свяжет шины USB внутри **karnix_soc** с внешними выводами микросхемы ПЛИС. На этом действия по интеграции компонента USB хост-контроллера в СнК закончены. Осталось разрешить вопрос с тактовым сигналом **usb_clk**.

5.3.6. Формирование тактового сигнала usb_clk

У используемой в плате «Карно» микросхемы ПЛИС Lattice ECP5-25F имеется на борту два встроенных PLL и так получилось, что оба из них уже заняты: один под формирование системной тактовой частоты ядра (60 МГц) от входного генератора 25 МГц. Второй PLL задействован под CGA видеоадаптер, этот PLL формирует частоту 250 МГц для блока TMDS, что формирует сигнал на HDMI разъем. Встает вопрос — откуда взять 12 МГц

для USB ? Мной были перепробованы различные варианты (они, кстати, представлены в файле **KarnixSoC.scala** в закомментированном виде) и пришел к выводу, что самый простой и более-менее рабочий вариант - это получить частоту **usb_clk** путем деления частоты **karnix_soc.io.pixclk_x10** используемой в блоке CGA видеоадаптера для формирования TMDS сигнала.

Для описания счетчика делителя частоты нам потребуется завести еще один тактовый домен **pixclk_x10_ClockDomain** и внутри него выполнить счетчик **usb_clk_div** тактируемый от **karnix_soc.io.pixclk_x10**. Данный счетчик будет непрерывно считать от 0 до 20 тактов и инвертировать сигнал **usb_clk** в точках 10 и 20, то есть будет происходить деление частоты 250 МГц на 21, результирующая частота на сигнальной линии **usb_clk** будет близкой к **11.9 МГц**. Практика показала, что этого вполне достаточно для нормальной работы «Low Speed» USB устройства.

Код делителя частоты для формирования **usb_clk**:

Листинг 8.26. Код файла KarnixSOC.scala: имплементация сигнала usb_clk.

```
val pixclk_x10_ClockDomain = ClockDomain(
  clock = karnix_soc.io.pixclk_x10,
  config = ClockDomainConfig(resetKind = BOOT),
  frequency = FixedFrequency(250.0 MHz)
)

val pixclk_x10_area = new ClockingArea(pixclk_x10_ClockDomain) {

  val usb_clk_div = Reg(UInt(5 bits)) init(0)
  val usb_clk = Reg(Bool()) init(False)

  usb_clk_div := usb_clk_div + 1

  when(usb_clk_div === 10) {
    usb_clk := True
  }

  when(usb_clk_div === 20) {
    usb_clk := False
    usb_clk_div := 0
  }

  var dcca_usb = DCCA()
  dcca_usb.CE := True
  dcca_usb.CLKI := usb_clk
  karnix_soc.io.usb_clk := dcca_usb.CLKO
}
```

В приведенном выше коде результирующий сигнал **usb_clk** прогоняется через встроенный в ПЛИС блок **DCCA()**, а его выход уже подключается к **karnix_soc.io.usb_clk** для тактирования аппаратуры USB хост-контроллера. Это необходимо для того, чтобы синтезатор Yosys понимал, что сформированный счетчиком сигнал **usb_clk** содержит «синтетический» тактовый сигнал который нужно трассировать используя специально выделенные внутри ПЛИС линии тактовых сигналов. Если этого не сделать, то в некоторых (и очень частых) случаях на линии тактового сигала будут формироваться задержки или джиттер, что плохо сказывается на передачи данных по шине USB.

5.3.7. Сборка проекта СнК «KarnixSoC»

Выполнение сборки всего проекта СнК «KarnixSoC» осуществляется из подкаталога **./scripts/KarnixSOC/ECP5-25F_karnix_board/**, в нём содержится **Makefile** описывающий алгоритм сборки. Этот алгоритм состоит из следующих шагов:

1. Компиляция C/C++ кода который будет использоваться в качестве программы начального старта для синтезируемого процессора VexRiscV (ROM). По умолчанию собирается примитивный загрузчик **karnix_bootloader**, его описание дано в файле ./src/main/c/karnix/karnix_bootloader/README.md.
2. Подготовка других бинарных файлов для инициализации синтезируемой аппаратуры. Например, формируется файл с растровыми шрифтами для знакогенератора CGA видеоадаптера.
3. Компиляция SpinalHDL программы и всех используемых модулей с помощью компилятора Scala и утилиты **sbt**.
4. Исполнение полученного байт-кода Java машиной, в результате чего генерируется файл **./KarnixSOCTopLevel.v** на языке Verilog с описанием всей аппаратуры СнК.
5. Verilog файл подается на вход синтезатору во главе с утилитой Yosys. Синтезатор производит синтез цифровой схемы (нетлиста), её оптимизацию, трассировку и размещение в ПЛИС. На выходе получается бинарный файл (битстрим) готовый к загрузке в ПЛИС.

Более подробно о тонкостях происходящих «под капотом» SpinalHDL в процессе сборки можно прочесть в моей статье «*Разработка цифровой аппаратуры нетрадиционным методом: Yosys, SpinalHDL, VexRiscv*», здесь же мы ограничимся только парой команд необходимых для запуска процесс сборки.

Чтобы запустить полную сборку проекта достаточно выполнить команду **make generate compile**. Цель **generate** занимается всем тем, что связано со Scala и SpinalHDL. Цель **compile** запускает синтезатор Yosys. Например:

Листинг 8.27. Фрагмент вывода на консоль при сборки проекта.

```
rz@devbox:~/KarnixSOC/scripts/KarnixSOC/ECP5-25F_karnix_board$ make generate compile

(cd ../../..; sbt "runMain vexriscv.demo.KarnixSOCVerilog")
[info] welcome to sbt 1.6.0 (Ubuntu Java 11.0.9.1)
[info] loading settings for project karnixsoc-build from plugins.sbt ...
[info] loading project definition from /home/rz/KarnixSOC/project
[info] loading settings for project root from build.sbt ...
[info] set current project to VexRiscv (in build file:/home/rz/KarnixSOC/)
[info] compiling 1 Scala source to /home/rz/KarnixSOC/target/scala-2.12/classes ...
[warn] four deprecations (since ???); re-run with -deprecation for details
[warn] one warning found
```

В этом месте, т. е. при компиляции и исполнении SpinalHDL программы, могут появляться сообщения об ошибках: синтаксических от компилятора Scala, или логических — от библиотеки SpinalHDL. При исполнении SpinalHDL программа выдает в консоль отладочную информацию:

```
[info] running (fork) vexriscv.demo.KarnixSOCVerilog
[info] [Runtime] SpinalHDL v1.10.2a    git head : a348a60b7e8b6a455c72e1536ec3d74a2ea16935
[info] [Runtime] JVM max memory : 8294.0MiB
[info] [Runtime] Current date : 2025.10.08 21:02:51
[info] [Progress] at 0.000 : Elaborate components
[info] Apb3USB10Ctrl::usbFrequency = 11900000 Hz
[info] Apb3USB10Ctrl::USBSlowSpeedClockDiv = 7
[info] Apb3USB10Ctrl::USBLowSpeedKeepAliveClocks = 11899
```

```
[info] Apb3USB10Ctrl::USBLowSpeedErrorClocks = 9519
…
```

Выделенные жирным сообщения это наши отладочные сообщения из класса **Apb3USB10Ctrl** выведенные с помощью функции **println()**, они отображают рассчитанные значения параметров (см. главу 5.2.9).

Если возникнут ошибки, то их следует устранить, чтобы SpinalHDL программа успешно исполнилась и сформировала Verilog файл. При успешной генерации Verilog выглядит на консоли будут сообщения со статистикой следующего характера:

```
[info] [Progress] at 6.443 : Checks and transforms
[info] [Progress] at 9.224 : Generate Verilog to .
[info] [Warning] 729 signals were pruned. You can call printPruned on the backend report to
get more informations.
[info] [Done] at 11.297
[success] Total time: 36 s, completed Oct 8, 2025, 9:03:03 PM
…
```

Далее создается подкаталог **bin.** В него будут помещаться временные и результирующий бинарный файлы, а процесс сборки переходит к утилите Yosys:

```
mkdir -p bin

yosys -l yosys.log -v2 -p "synth_ecp5 -abc9 -top KarnixSOCTopLevel -json
bin/KarnixSOCTopLevel.json" ../../../KarnixSOCTopLevel.v
../../../src/main/verilog/TMDS_encoder.sv ../../../src/main/verilog/OBUFDS.sv
1. Executing Verilog-2005 frontend: ../../../KarnixSOCTopLevel.v
...
```

В процессе синтеза тоже могут возникать ошибки, но чаще всего они связаны с нехваткой каких либо ресурсов ПЛИС. Информация о потребляемых ресурсах также выдается на консоль:

```
Info: Device utilisation:
Info:              TRELLIS_IO:     121/    197    61%
Info:                    DCCA:       6/     56    10%
Info:                  DP16KD:      55/     56    98%
Info:              MULT18X18D:       9/     28    32%
Info:                  ALU54B:       0/     14     0%
Info:                 EHXPLLL:       2/      2   100%
Info:                 EXTREFB:       0/      1     0%
Info:                    DCUA:       0/      1     0%
Info:               PCSCLKDIV:       0/      2     0%
Info:                 IOLOGIC:       0/    128     0%
Info:                SIOLOGIC:       0/     69     0%
Info:                     GSR:       0/      1     0%
Info:                   JTAGG:       0/      1     0%
Info:                    OSCG:       0/      1     0%
Info:                   SEDGA:       0/      1     0%
Info:                     DTR:       0/      1     0%
Info:                 USRMCLK:       0/      1     0%
Info:                 CLKDIVF:       0/      4     0%
Info:               ECLKSYNCB:       0/     10     0%
Info:                 DLLDELD:       0/      8     0%
Info:                  DDRDLL:       0/      4     0%
Info:                 DQSBUFM:       0/      8     0%
Info:         TRELLIS_ECLKBUF:       0/      8     0%
Info:            ECLKBRIDGECS:       0/      2     0%
Info:                    DCSC:       0/      2     0%
Info:              TRELLIS_FF:    8417/  24288    34%
Info:            TRELLIS_COMB:   22947/  24288    94%
Info:            TRELLIS_RAMW:     697/   3036    22%
...
```

После выполнения трассировки и размещения на консоль выдается итоговая статистика по предельным частотам на различных тактовых сигналах. Если заданные

частоты превышают предельно допустимые, то синтез битстрима прекращается. В этом случае требуется провести оптимизацию кода или запустить синтез еще раз со случайным «seed» — вдруг на этот раз повезет? ;-)

```
Info: Max frequency for clock '$glbnet$io_lan_rxclk$TRELLIS_IO_IN': 131.89 MHz (PASS at
25.00 MHz)
Info: Max frequency for clock             '$glbnet$hdmi_pll_CLKOP': 379.08 MHz (PASS at
250.00 MHz)
Info: Max frequency for clock  'io_hdmi_tmds_clk_p$TRELLIS_IO_OUT': 42.65 MHz (PASS at 12.00
MHz)
Info: Max frequency for clock '$glbnet$io_lan_txclk$TRELLIS_IO_IN': 104.96 MHz (PASS at
25.00 MHz)
Info: Max frequency for clock           'io_clkusb$TRELLIS_IO_OUT': 116.02 MHz (PASS at
12.00 MHz)
Info: Max frequency for clock  '$glbnet$io_clkcore$TRELLIS_IO_OUT': 65.55 MHz (PASS at 60.00
MHz)
...
```

В итоге весь процесс сборки заканчивается сообщением вида:

```
Info: Program finished normally.
ecpbram -v -i bin/KarnixSOCTopLevel_random_25F.config -o bin/KarnixSOCTopLevel_25F.config -f
../../../KarnixSOCTopLevel_random.hexx -t
../../../src/main/c/karnix/karnix_bootloader/build/karnix_bootloader.hexx
Padding to_hexfile from 2769 words to 18432
Loaded pattern for 32 bits wide and 18432 words deep memory.
Extracted 1152 bit slices from from/to hexfile data.
ecppack --svf bin/KarnixSOCTopLevel_25F.svf  bin/KarnixSOCTopLevel_25F.config
bin/KarnixSOCTopLevel_25F-karnix_bootloader.bit
```

Это означает, что результирующий битстрим готов и находится в файле **bin/KarnixSOCTopLevel_25F-karnix_bootloader.bit** :

```
rz@devbox:~/KarnixSOC/scripts/KarnixSOC/ECP5-25F_karnix_board$ ls -l bin/
total 67204
-rw-rw-r-- 1 rz rz 34867485 Oct  8 21:06 KarnixSOCTopLevel.json
-rw-rw-r-- 1 rz rz   709859 Oct  8 21:13 KarnixSOCTopLevel_25F-karnix_bootloader.bit
-rw-rw-r-- 1 rz rz 10581041 Oct  8 21:13 KarnixSOCTopLevel_25F.config
-rw-rw-r-- 1 rz rz  1501646 Oct  8 21:13 KarnixSOCTopLevel_25F.svf
-rw-rw-r-- 1 rz rz 10564390 Oct  8 21:08 KarnixSOCTopLevel_placed.svg
-rw-rw-r-- 1 rz rz 10579662 Oct  8 21:12 KarnixSOCTopLevel_random_25F.config
```

Теперь остается только подключить плату «Карно» с USB порту рабочей машины на которой производится сборка проекта и загрузить в неё битстрим следующей командой **make prog**:

Листинг 8.28. Вывод на консоль при прошивки битстрима в ПЛИС.

```
rz@devbox:~/KarnixSOC/scripts/KarnixSOC/ECP5-25F_karnix_board$ make prog

openFPGALoader bin/KarnixSOCTopLevel_25F-karnix_bootloader.bit
write to flash
No cable or board specified: using direct ft2232 interface
Jtag frequency : requested 6.00MHz   -> real 6.00MHz
Open file DONE
Parse file b3bdffff
DONE
Enable configuration: DONE
SRAM erase: DONE
Detail:
Jedec ID          : ef
memory type       : 70
memory capacity   : 18
```

```
flash chip unknown: use basic protection detection
start addr: 00000000, end_addr: 000b0000
Erasing: [================================================] 100.00%
Done
Writing: [================================================] 100.00%
Done
Refresh: DONE
```

Однако, даже если сборка аппаратной части прошла успешно и битстрим загрузился в микросхему ПЛИС, проку в этом немного, так как у нас нет ни программы использующей новую аппаратуру, ни драйвера ей управляющего. Следующие главы будут посвящены написанию низкоуровневого драйвер и простой тестовой программы на языке Си для проверки работоспособности USB хост-контроллера.

6. Реализация драйвера для USB хост-контроллера

Начнем написание низкоуровневого драйвера для нашего USB хост-контроллера с создания программных примитивов для отправки команд и ожидания статуса, а также выполнения сброса шины. Затем перейдем к более сложным функциям работы с транзакциями: IN, OUT и SETUP. После чего реализуем функции для выполнения произвольного запроса, считывания дескрипторов и установки параметров. Далее реализуем функцию инициализации устройства как последовательность запросов. И в самом конце реализуем функцию **usb10_init()** позволяющую приложению выяснить, есть ли на шине устройство, а так же выполняющую полный цикл инициализации устройства. Весь код драйвера будет располагаться в двух файлах:

./src/main/c/karnix/karnix_soc/src/usb10.h
и
./src/main/c/karnix/karnix_soc/src/usb10.c

Создадим файл **usb10.h** и добавим в него описание структуры аппаратных регистров контроллера (листинг 9.1), определения макросов для поддерживаемых команд и токенов (листинги 9.2 и 9.3), а так же макроопределения для работы с битовыми полями регистров (листинг 9.4).

Листинг 9.1. Структура регистров хост-контроллера USB 1.0.

```
typedef struct {
        volatile uint32_t STATUS;               // Controller Status register
        volatile uint32_t COMMAND;              // Command register
        volatile uint32_t RECV_DATA_LOW;        // Received data register, low 32 bits
        volatile uint32_t RECV_DATA_HIGH;       // Received data register, high 32 bits
        volatile uint32_t SEND_DATA_LOW;        // Send data register, low 32 bits
        volatile uint32_t SEND_DATA_HIGH;       // Send data register, high 32 bits
        volatile uint32_t RX_STATUS;            // Receiver status register 1
        volatile uint32_t CONTROL;              // And configuration register
        volatile uint32_t RX_STATUS2;           // Receive status register 2
} USB10_Reg;
```

Листинг 9.2. Команды хост-контроллера USB 1.0.

```
#define USB10_CMD_NOP                   0x00    // No-operation (does nothing)
#define USB10_CMD_SEND_TOKEN            0x01    // Send token with CRC5
#define USB10_CMD_SEND_SHORT_TOKEN      0x02    // Send short token
#define USB10_CMD_SEND_DATA             0x03    // Send data packet
#define USB10_CMD_BUS_RESET             0x04    // Initiate Bus Reset state
```

Листинг 9.3. Токены поддерживаемые хост-контроллером USB 1.0.

```
#define USB10_PID_SETUP         0b00101101      // "1011 0100"  SETUP   Address for host-to-
device control transfer

#define USB10_PID_DATA0         0b11000011      // "1100 0011"  DATA0   Even-numbered data
packet

#define USB10_PID_DATA1         0b01001011      // "1101 0010"  DATA1   Odd-numbered data
packet

#define USB10_PID_IN            0b01101001      // "1001 0110"  IN      Address for device-
to-host transfer
```

```
#define USB10_PID_OUT           0b11100001      // "1000 0111"  OUT     Address for host-to-
device transfer

#define USB10_PID_ACK           0b11010010      // "0100 1011"  ACK     Data packet accepted
#define USB10_PID_NAK           0b01011010      // "0101 1010"  NAK     Data packet not
accepted; please retransmit

#define USB10_PID_STALL         0b00011110      // "0111 1010"  STALL   Transfer impossible;
do error recovery
```

Листинг 9.4. Макросы для работы с битовыми полями регистров хост-контроллера.

```
// Get and set ADDR bits of COMMAND word
#define USB10_CMD_ADDR_S        24
#define USB10_CMD_ADDR_M        0x7f
#define USB10_CMD_ADDR(X)       (((X) >> USB10_CMD_ADDR_S) & USB10_CMD_ADDR_M)
#define USB10_CMD_SET_ADDR(X)   (((X) & USB10_CMD_ADDR_M) << USB10_CMD_ADDR_S)

// Get and set ENDP bits of COMMAND word
#define USB10_CMD_ENDP_S        20
#define USB10_CMD_ENDP_M        0x0f
#define USB10_CMD_ENDP(X)       (((X) >> USB10_CMD_ENDP_S) & USB10_CMD_ENDP_M)
#define USB10_CMD_SET_ENDP(X)   (((X) & USB10_CMD_ENDP_M) << USB10_CMD_ENDP_S)

// Get and set command length bits of COMMAND word
#define USB10_CMD_LEN_S         8
#define USB10_CMD_LEN_M         0xfff
#define USB10_CMD_LEN(X)        (((X) >> USB10_CMD_LEN_S) & USB10_CMD_LEN_M)
#define USB10_CMD_SET_LEN(X)    (((X) & USB10_CMD_LEN_M) << USB10_CMD_LEN_S)

// Get and set PID bits of COMMAND word
#define USB10_CMD_PID_S         4
#define USB10_CMD_PID_M         0x0f
#define USB10_CMD_PID(X)        (((X) >> USB10_CMD_PID_S) & USB10_CMD_PID_M)
#define USB10_CMD_SET_PID(X)    (((X) & USB10_CMD_PID_M) << USB10_CMD_PID_S)

// Get and set command bits of COMMAND word
#define USB10_CMD_S             0
#define USB10_CMD_M             0x0f
#define USB10_CMD(X)            (((X) >> USB10_CMD_S) >> USB10_CMD_M)
#define USB10_CMD_SET(X)        (((X) & USB10_CMD_M) << USB10_CMD_S)
```

и т. д. Полное описание всех макроопределений см. в репозитории в файле usb10.h.

6.1. Описание программных примитивов

Для того, чтобы отправлять и получать пакеты по шине, выполнять транзакции, осуществлять процедуру инициализации или сброс шины, введем следующие программные примитивы в файл **./src/main/c/karnix/karnix_soc/src/usb10.c**.

Чтобы компилятор не разбивал запись в аппаратный 32-х битный регистр контроллера на несколько операций, что может ввести контроллер в заблуждение, а выполнял запись в регистр «атомарно», введем инлайн функцию **sub10_write_reg()** состоящую из одной ассемблерной инструкции:

```
static inline void usb10_write_reg(volatile uint32_t* reg, uint32_t val) {
        asm volatile ("sw %0, (%1)" :  : "r"(val), "r"(reg));
}
```

Для симметрии введем инлайн функцию чтения 32-х битного регистра **usb10_read_reg()**:

```
static inline uint32_t usb10_read_reg(volatile uint32_t* reg) {
	return *reg;
}
```

Функция **usb10_write_reg()** позволит нам отправлять команды контроллеру одной строкой, например:

```
// RESET

usb10_write_reg(&reg->COMMAND, USB10_CMD_START_BIT | USB10_CMD_SET(USB10_CMD_BUS_RESET));
```

Для ожидания освобождения контроллера введем инлайн функцию **usb10_wait_while_busy()**. Первый параметр этой функции указывает на базовый адрес пространства аппаратных регистров контроллера. Второй параметр задает максимальное время ожидания выраженное в циклах процессора. Функция возвращает число оставшихся циклов ожидания или 0 если время ожидания истекло.

Листинг 9.5. Функция ожидания освобождения хост-контроллера.

```
static inline int usb10_wait_while_busy(USB10_Reg* reg, int timeout) {

	int i;

	// Wait for execution to complete
	while(timeout--)
		if((usb10_read_reg(&reg->STATUS) & USB10_STATUS_BUSY_BIT) == 0)
			return timeout;

	return 0; // Fail
}
```

Аналогичным образом определим более сложный примитив **usb10_wait_cmd_complete()** — функцию для ожидания завершения выполнения команды. Данная функция сначала ожидает когда контроллер начнет выполнение команды (что может происходить не моментально после записи команды в регистр), а потом дожидается завершения выполнения. Назначение параметров и возвращаемый результат такой же как и у предыдущего примитива.

Листинг 9.6. Функция ожидания завершения выполнения команды хост-контроллером.

```
static inline int usb10_wait_cmd_complete(USB10_Reg* reg, int timeout) {

	int i;

	// Make sure command execution has begun
	for(i = 0; i < 50; i++)
		if(usb10_read_reg(&reg->STATUS) & USB10_STATUS_BUSY_BIT)
			break;

	// Wait for execution to complete
	while(timeout--)
		if((usb10_read_reg(&reg->STATUS) & USB10_STATUS_BUSY_BIT) == 0)
			return timeout; // OK

	return 0; // Fail
}
```

Для того, чтобы вести учет типа последнего переданного и принятого пакетов DATAx, заведем массив для структур состоящих из двух байтов **send** и **recv** в котором будем хранить тип (0 или 1) последнего отправленного и полученного пакетов для каждого из устройств на шине.

Листинг 9.7. Массив структур для сохранения типа последнего пакета данных.

```
struct _usb10_last_data_pid {
        uint8_t sent;
        uint8_t recv;
} usb10_last_data_pid[MAX_ADDRESSES][MAX_ENDPOINTS];
```

Значения макросов **MAX_ADDRESSES** и **MAX_ENDPOINTS** определим в файле usb10.h следующим образом:

```
#define MAX_ADDRESSES           4               // How many devices should be supported
#define MAX_ENDPOINTS           4               // How many endpoints per device
```

6.2. Функции выполнения транзакций

Теперь у нас всё готово для реализации первой процедуры отсылки команды. Реализуем набор функций для выполнения транзакций на шине USB.

Описанные ниже функции **usb10_in_request()**, **usb10_out_request()** и **usb10_setup_request()** являются необходимыми базовыми элементами для выполнения более сложных обменов описанных в главе *3.2. «Транзакции»*.

6.2.1. Функция usb10_bus_reset() для сброса шины

Для выполнения процедуры сброса шины создадим функцию **usb10_bus_reset()** которая воспользовавшись выше описанными примитивами пошлет хост-контроллеру команду **USB10_CMD_BUS_RESET** и подождет пока устройства на шине выполнят внутренний цикл перезапуска:

Листинг 9.7. Функция выполнения процедуры сброса шины.

```
#define USB10_DEVICE_RESET_STR  "usb10_device_reset"

int usb10_bus_reset(USB10_Reg* reg, int wait_us)
{

        if(usb10_read_reg(&reg->STATUS) & USB10_STATUS_ERROR_BIT) {
                usb10_printf("%s: device not connected!\r\n", USB10_DEVICE_RESET_STR);
                return -1;
        }

        // RESET

        usb10_write_reg(&reg->COMMAND, USB10_CMD_START_BIT |
                        USB10_CMD_SET(USB10_CMD_BUS_RESET));

        if(!usb10_wait_cmd_complete(reg, 600000)) { // ~30ms timeout
                usb10_printf("%s: hung after %s packet!\r\n", USB10_DEVICE_RESET_STR,
"RESET");
                return -2;
        }

        for(int addr = 0; addr < MAX_ADDRESSES; addr++)
                for(int endp = 0; endp < MAX_ENDPOINTS; endp++) {
```

```
                        usb10_last_data_pid[addr][endp].sent = USB10_PID_DATA1;
                        usb10_last_data_pid[addr][endp].recv = USB10_PID_DATA0;
                }

        if(wait_us)
                delay_us(wait_us); // Usually wait for 20 ms for device to settle

        return 0;
}
```

В случае успешного исполнения команды хост-контроллером функция **usb10_bus_reset()** производит инициализацию массива **usb10_last_data_pid** начальными значениями и возвращает нулевой код ошибки. Если в процессе работы с хост-контроллером возникли ошибки, то данная функция возвращает отрицательный код показывающий на каком шаге что-то пошло не так.

6.2.2. Функция usb10_in_request() для получения данных от устройства

Следующая функция **usb10_in_request()** отправляет по шине токен IN с указанным адресом устройства **address** и номером конечной точки **endpoint**, после чего принимает один и более пакет данных DATAx подтверждая прием каждого. Эта функция выполняет целую транзакцию IN получения данных от устройства. Как видно из листинга 9.8 код данной функции достаточно объемный, попробуем разобраться в нем. Функция **usb10_in_request()** действует по следующему алгоритму:

1. Ожидает освобождения хост-контроллера от предыдущей задачи (или приема данных).
2. Вычисляет число пакетов данных которое ожидает пользователь указав входной параметр **response_size**. Если пользователь указал **0**, то принимается минимум один пакет, а дальше — как получится (принимается столько, сколько есть у устройства для передачи).
3. Хост-контроллеру передается команда **USB10_CMD_SEND_LONG_TOKEN** с указанием типа токена **USB10_PID_IN**, вместе с командой передаются значения **address** и **endpoint**.
4. Ожидается прием пакета от устройства путем мониторинга флага **USB10_STATUS_RECEIVED_BIT**.
5. Проверяется какой пакет принят: STALL, NAK или DATAx. В случае STALL, NAK или возникновения таймаута процесс завершается с соответствующим код ошибки.
6. Если принят пакет с данными, то проверяется бит валидности контрольной суммы **USB10_STATUS_CRC16_OK_BIT**. Если зафиксирована ошибка CRC16, то производится повторная попытка приема, т. е. переход на шаг 3 данного алгоритма с уменьшением числа попыток.
7. Если ошибка приема не зафиксирована (бит валидности установлен), то устройству высылается подтверждение ACK путем отправки хост-контроллеру команды **USB10_CMD_SEND_SHORT_TOKEN**.
8. После завершения команды отсылки токена ACK производится сверка PID полученного пакета с тем, что зафиксировано в массиве **usb10_last_data_pid**. Если эти значения совпадают, что получен дубликат пакета данных, он выбрасывается и цикл приема повторяется.
9. Если PID полученного пакета и зафиксированного в **usb10_last_data_pid** не совпадают, то считается то принят новый пакет с данными. Принятые данные

перекладываются в выходной буфер пользователя на который указывает **response_data**, значение **usb10_last_data_pid** обновляется.

10. Проверяется размер полезной нагрузки принятого пакета. Если он меньше максимального возможного для пакета, то считается что получен «хвост» и больше у устройства данных для передачи нет. Процесс завершается с положительным кодом ошибки (успех) равным суммарной длине принятых блоков.
11. Проверяется максимально допустимая длина принимаемого блока данных заданная пользователем в **response_size**. Если она достигнута, то также происходит завершение процедуры с положительным кодом ошибки равным этой длине.
12. В противном случае цикл приема повторяется путем перехода на шаг 3.

Листинг 9.8. Функция выполнения транзакции IN для получения данных от устройства.

```
#define USB10_IN_REQUEST_STR    "usb10_in_request"

int usb10_in_request(USB10_Reg* reg, uint8_t address, uint8_t endpoint,
        uint8_t* response_data, uint32_t response_size)
{
        if(address >= MAX_ADDRESSES || endpoint >= MAX_ENDPOINTS)
                return -999;

        // Wait till bus is free
        if(!usb10_wait_while_busy(reg, 10000)) { // ~1ms timeout
                usb10_printf("%s: bus is busy for too long!\r\n",
USB10_IN_REQUEST_STR);
                return -1;
        }

        csr_clear(mstatus, MSTATUS_MIE); // Disable Machine interrupts during I/O

        int ret = 0;
        int bytes_received = 0;

        int num_of_data_packets = response_size % USB10_LOW_SPEED_DATA_SIZE ?
                response_size / USB10_LOW_SPEED_DATA_SIZE + 1 : response_size /
USB10_LOW_SPEED_DATA_SIZE;

        if(response_size == 0)
                num_of_data_packets = 1; // read at least one data packet

        for(int i = 0; i < num_of_data_packets; i++) {

                // Request part of response data

                uint32_t rx_status;
                int timeout;
                int retry = 8;

                int packet_size_bits = (i + 1) * USB10_LOW_SPEED_DATA_SIZE  <=
response_size ?
                                (USB10_LOW_SPEED_DATA_SIZE * 8 + 8 + 16) : // PID8 +
DATA*8 + CRC16
                                (response_size % USB10_LOW_SPEED_DATA_SIZE) * 8 + 8 +
16;

                again_data:

                usb10_write_reg(&reg->COMMAND, USB10_CMD_START_BIT |
                                USB10_CMD_SET_PID(USB10_PID_IN) |
                                USB10_CMD_SET_ADDR(address) |
                                USB10_CMD_SET_ENDP(endpoint) |
                                USB10_CMD_SET(USB10_CMD_SEND_LONG_TOKEN));

                timeout = 25000;
                while(timeout--)
                        if(usb10_read_reg(&reg->STATUS) & USB10_STATUS_RECEIVED_BIT)
                                goto rcvd2;

                {
```

```
                        usb10_printf("%s: no response after %s(%d:%d)\r\n",
USB10_IN_REQUEST_STR, "IN",
                                address, endpoint);

                        if(retry--)
                                goto again_data;

                        ret = -6;
                        goto fail;
                }

                rcvd2:

                rx_status = usb10_read_reg(&reg->RX_STATUS); // read once, use many
times

                int received_pid = USB10_RX_STATUS_PID(rx_status);

                if(received_pid == USB10_PID_NAK) {
                        ret = -8;
                        goto fail;
                }

                if(received_pid == USB10_PID_STALL) { // STALL indicated fatal error
                        ret = -12;
                        goto fail;
                }

                // Note: Some short token packets missing CRC16 which also considered
as fail

                #if(1)
                if(!(usb10_read_reg(&reg->STATUS) & USB10_STATUS_CRC16_OK_BIT)) {
                        usb10_printf("%s: CRC16 error (%d), STATUS = %08X, RX_STATUS
= %08X, RX_STATUS2 = %08X, DATA = %08X:%08X\r\n",
                                USB10_IN_REQUEST_STR, i, usb10_read_reg(&reg-
>STATUS), rx_status,
                                usb10_read_reg(&reg->RX_STATUS2),
                                usb10_read_reg(&reg->RECV_DATA_HIGH),
usb10_read_reg(&reg->RECV_DATA_LOW)
                        );

                        delay_us(20);

                        if(retry--)
                                goto again_data;

                        ret = -10;
                        goto fail;
                }
                #endif

                // Send ACK for received packet

                usb10_write_reg(&reg->COMMAND, USB10_CMD_START_BIT |
                                USB10_CMD_SET_PID(USB10_PID_ACK) |
                                USB10_CMD_SET(USB10_CMD_SEND_SHORT_TOKEN));

                if(!usb10_wait_cmd_complete(reg, 20000)) { // 2ms timeout
                        usb10_printf("%s: hung after %s packet!\r\n",
USB10_IN_REQUEST_STR, "ACK");
                        ret = -7;
                        goto fail;
                }

                // Now analyze what we've got here

                if(received_pid != USB10_PID_DATA0 && received_pid !=
USB10_PID_DATA1) {
                        usb10_printf("%s: expected %s packet instead of 0x%02X,
RX_STATUS = %08X\r\n", USB10_IN_REQUEST_STR,
                                "DATA", received_pid, rx_status);

                        if(retry--)
                                goto again_data;

                        ret = -9;
```

```
                                goto fail;
                        }

                        if(received_pid == usb10_last_data_pid[address][endpoint].recv) {

                                usb10_printf("%s: received %d dupe data, RX_STATUS = %08X,
        DATA = %08X:%08X\r\n",
                                        USB10_IN_REQUEST_STR, i, rx_status,
                                        usb10_read_reg(&reg->RECV_DATA_HIGH),
        usb10_read_reg(&reg->RECV_DATA_LOW)
                                );

                                if(retry--)
                                        goto again_data;

                                ret = -10;
                                goto fail;
                        }

                        usb10_printf("%s: packet received (%d), RX_STATUS = 0x%08X, DATA =
        %08X:%08X\r\n",
                                USB10_IN_REQUEST_STR, i, rx_status,
                                usb10_read_reg(&reg->RECV_DATA_HIGH), usb10_read_reg(&reg-
        >RECV_DATA_LOW)
                        );

                        usb10_last_data_pid[address][endpoint].recv = received_pid; //
        Remember last received DATA PID

                        if(response_size && response_data) { // collect data if response
        expected
                                *(uint32_t*)(response_data + i * 8 + 0) =
        usb10_read_reg(&reg->RECV_DATA_LOW);
                                *(uint32_t*)(response_data + i * 8 + 4) =
        usb10_read_reg(&reg->RECV_DATA_HIGH);
                        }

                        bytes_received += (USB10_RX_STATUS_LEN(rx_status) - 8 - 16) / 8;

                        #if(1) // Enable this if packet size matching is necessary
                        if(USB10_RX_STATUS_LEN(rx_status) != packet_size_bits) {
                                usb10_printf("%s: received bogus data packet (%d) size: %d
        bits != %d, DATA = %08X:%08X\r\n",
                                        USB10_IN_REQUEST_STR, i,
        USB10_RX_STATUS_LEN(rx_status), packet_size_bits,
                                        usb10_read_reg(&reg->RECV_DATA_HIGH),
        usb10_read_reg(&reg->RECV_DATA_LOW)
                                );

                                break;
                        }
                        #endif

                }


                usb10_printf("%s: RX DATA: ", USB10_IN_REQUEST_STR);
                usb10_printf("%s: RX %d bytes of DATA: ", bytes_received, USB10_IN_REQUEST_STR);

                for(int i = 0; i < bytes_received; i++)
                        usb10_printf("%02X ", response_data[i]);

                usb10_printf("\r\n", 0);

                ret = bytes_received;

                fail:

                csr_set(mstatus, MSTATUS_MIE); // Enable Machine interrupts

                return ret;
        }
```

Здесь следует сделать ряд замечаний по реализации проверок при реализации транзакции. Дело в том, что хосту отводится очень мало времени на реакцию на полученный

пакет (менее 12 мкс). Если драйвер будет выполнять все необходимые проверки после получения пакета, то он не сможет вовремя отреагировать (вовремя отослать ACK), что приведет к очень нежелательному эффекту — устройство будет воспринимать это как ошибку передачи и будет постоянно повторять посылку одного и того же пакета (либо выдаст STALL). Чтобы избежать такого эффекта, в драйвере, в процессе обмена во-первых запрещается обработка прерываний, и во-вторых делается минимум проверок, а код функции устроен так, чтобы исполнялся как можно быстрее. Это касается всех дальнейших реализаций транзакций.

6.2.3. Функция usb10_out_request() для отправки данных в устройство

Следующая функция **usb10_out_request()** отправляет по шине токен OUT с указанным адресом устройства **address** и номерам конечной точки **endpoint**, после чего отправляет один и более пакет данных DATAx, каждый раз дожидаясь подтверждения. На передаваемый блок данных ссылается указатель **request_data**, длина задается переменной **request_size**. Эта функция выполняет транзакцию OUT передачи данных в устройство.

Функция **usb10_out_request()** может передать пакет DATAx с блоком полезной нагрузки размер которой менее максимально допустимого (передача «хвоста») или с вообще может быть пустым (без данных). Последнее часто используется для сигнализации конца обмена. Реализация функции приведена в листинге 9.9. Алгоритм действия функции **usb10_out_request()** следующий:

1. Ожидается освобождение хост-контроллера от выполнения предыдущей команды или приема даных.
2. Вычисляется количество пакетов DATAx необходимых для отправки всего блока данных. Если размер передаваемого блока **request_size** равен нулю, то будет передан один пакет DATAx с пустым блоком полезной нагрузки.
3. На каждый передаваемой пакет DATAx вычисляется значение **packet_size_bits** задающее размер пакета в битах и **packet_pid** (PID пакета) исходя из значения сохраненного в массиве **usb10_last_data_pid**.
4. С помощью команды **USB10_CMD_SEND_LONG_TOKEN** хост-контроллеру сначала по шине передается токен OUT. Вместе с токеном передается адрес устройства **address** и номер адресуемой конечной точки **endpoint**.
5. Производится ожидание завершения исполнения команды отправки токена.
6. С помощью команды **USB10_CMD_SEND_DATA** хост-контроллеру передается пакет с данными DATAx. Вместе с командой передается тип пакета **packet_pid** и его длина в битах **packet_size_bits**.
7. Сразу после отправки пакета с данными драйвер переходит к опросу флага **USB10_STATUS_RECEIVED_BIT** регистра статуса хост-контроллера. Установка этого флага сигнализирует о получении подтверждающего пакета от устройства.
8. В случае если подтверждающий пакет не получен по истечению таймаута, производится повторная попытка отправить данные, то есть переход на шаг 4.
9. Анализируется тип полученного подтверждающего пакета. Если тип подтверждающего пакета не ACK, то происходит завершение и выход из функции с отрицательным кодом ошибки отражающей шаг на котором она произошла.
10. Если же ACK получен, то производится сохранение номера только что отправленного пакета **packet_pid** в массив **usb10_last_data_pid** и управление передается на начало цикла для отправки следующего пакета, т. е. на шаг 3.
11. После успешной отправки всех пакетов функция **usb10_out_request()** завершается с кодом **0** (успех).

Листинг 9.9. Функция выполнения транзакции OUT для отправки данных на устройство.

```
#define USB10_OUT_REQUEST_STR   "usb10_out_request"

int usb10_out_request(USB10_Reg* reg, uint8_t address, uint8_t endpoint,
        uint8_t* request_data, uint32_t request_size)
{
        /* NOTE: request_data buffer should be at least 8 bytes long, even if request_size
is zero !!! */

        if(address >= MAX_ADDRESSES || endpoint >= MAX_ENDPOINTS)
                return -999;

        // Wait till bus is free
        if(!usb10_wait_while_busy(reg, 10000)) { // ~1ms timeout
                usb10_printf("%s: bus is busy for too long!\r\n", USB10_OUT_REQUEST_STR);
                return -1;
        }

        csr_clear(mstatus, MSTATUS_MIE); // Disable Machine interrupts during I/O

        int ret = 0;

        int num_of_data_packets = request_size % USB10_LOW_SPEED_DATA_SIZE ?
                request_size / USB10_LOW_SPEED_DATA_SIZE + 1 : request_size /
USB10_LOW_SPEED_DATA_SIZE;

        if(request_size == 0)
                num_of_data_packets = 1; // send at least one data packet

        for(int i = 0; i < num_of_data_packets; i++) {

                // Send part of request data

                uint32_t rx_status;
                int timeout;
                int retry = 8;

                int packet_size_bits = (i + 1) * USB10_LOW_SPEED_DATA_SIZE  <=
request_size ?
                                (USB10_LOW_SPEED_DATA_SIZE * 8 - 1) : // DATA*8 - 1
                                (request_size % USB10_LOW_SPEED_DATA_SIZE) * 8 - 1;

                // Calculate new DATA PID depending on last used/received
                int data_pid = (usb10_last_data_pid[address][endpoint].sent ==
USB10_PID_DATA0) ?
                                USB10_PID_DATA1 : USB10_PID_DATA0;
                again_out:

                // Prepare data to send, if such were provided
                if(request_size && request_data) {
                        usb10_write_reg(&reg->SEND_DATA_LOW, *(uint32_t*)(request_data +
0));
                        usb10_write_reg(&reg->SEND_DATA_HIGH, *(uint32_t*)(request_data +
4));
                }

                // Send OUT token first

                usb10_write_reg(&reg->COMMAND, USB10_CMD_START_BIT |
                                USB10_CMD_SET_PID(USB10_PID_OUT) |
                                USB10_CMD_SET_ADDR(address) |
                                USB10_CMD_SET_ENDP(endpoint) |
                                USB10_CMD_SET(USB10_CMD_SEND_LONG_TOKEN));

                if(!usb10_wait_cmd_complete(reg, 20000)) { // 2ms timeout
                        usb10_printf("%s: hung after %s packet!\r\n", USB10_OUT_REQUEST_STR,
"OUT");
                        ret = -2;
                        goto fail;
                }

                // Send DATA packet
                usb10_write_reg(&reg->COMMAND, USB10_CMD_START_BIT |
```

```
                                USB10_CMD_SET_LEN(packet_size_bits) |
                                USB10_CMD_SET_PID(data_pid) |
                                USB10_CMD_SET(USB10_CMD_SEND_DATA));

                if(!usb10_wait_cmd_complete(reg, 5000)) { // ~0.5ms timeout
                        usb10_printf("%s: hung after %s packet!\r\n", USB10_OUT_REQUEST_STR,
"DATA");
                        ret = -3;
                        goto fail;
                }

                timeout = 2500;
                while(timeout--)
                        if(usb10_read_reg(&reg->STATUS) & USB10_STATUS_RECEIVED_BIT)
                                goto recv1;

                {
                        usb10_printf("%s: no response after %s(%d:%d)\r\n",
USB10_OUT_REQUEST_STR, "DATA",
                                address, endpoint);

                        if(retry--)
                                goto again_out;

                        ret = -6;
                        goto fail;
                }

                recv1:

                rx_status = usb10_read_reg(&reg->RX_STATUS); // read once, use many times

                // Now analyze what we've got here

                int received_pid = USB10_RX_STATUS_PID(rx_status);

                if(received_pid == USB10_PID_NAK) {
                        ret = -8;
                        goto fail;
                }

                if(received_pid != USB10_PID_ACK) {
                        usb10_printf("%s: expected %s packet instead of 0x%02X, RX_STATUS =
%08X\r\n",
                                USB10_OUT_REQUEST_STR, "ACK",
                                USB10_RX_STATUS_PID(rx_status), rx_status);

                        if(received_pid == USB10_PID_STALL) {
                                ret = -9;
                                goto fail;
                        }

                        if(retry--)
                                goto again_out;

                        ret = -9;
                        goto fail;
                }

                usb10_last_data_pid[address][endpoint].sent = data_pid;

                usb10_printf("%s: packet sent (%d), RX_STATUS = 0x%08X, DATA = %08X:
%08X\r\n",
                        USB10_OUT_REQUEST_STR, i, rx_status,
                        usb10_read_reg(&reg->SEND_DATA_HIGH), usb10_read_reg(&reg-
>SEND_DATA_LOW)
                );

        }

        fail:

        csr_set(mstatus, MSTATUS_MIE); // Enable Machine interrupts

        return ret;
}
```

6.2.4. Функция usb10_setup_request() для отправки запроса и получения ответа

Функция **usb10_setup_request()** является своего рода комбинацией первых двух, она служит для отправки по шине токена SETUP содержащего пакет с данными запроса, с последующим ожиданием и получением ответа на этот запрос. Вместе с токеном SETUP передаются адрес устройства **address** и номер конечной точки **endpoint**. Обычно запросы типа SETUP отправляются на дефолтную конечную точку с номером **0** и являются частью конфигурационного процесса. Согласно спецификации, отправка SETUP всегда сбрасывает счетчик пакетов на устройстве, а значит функция устанавливает начальные значения в массиве **usb10_last_data_pid** для данного устройства и точки доступа.

Функция **usb10_setup_request()** принимает в качестве параметров два указателя: **request_data** — указывает на буфер содержащий код запроса (согласно спецификации это всегда 8 байт), и **response_data** — указывает на буфер в который требуется поместить принятый ответ. Максимальный размер ответа задается параметром **response_size**. Ответ может содержать меньшее число байт, но не может быть больше **response_size**. Функция **usb10_setup_request()** реализует следующий алгоритм:

1. Ожидается освобождение хост-контроллера от выполнения предыдущей команды или приема данных.
2. Устанавливаются значения по умолчанию для типа пакета в массиве **usb10_last_data_pid** в обе стороны - на прием и на передачу.
3. С помощью команды хост-контроллера **USB10_CMD_SEND_LONG_TOKEN** отправляется токен SETUP содержащий адрес устройства **address** и номер конечной точки **endpoint**.
4. Ожидается завершение исполнение команды хост-контроллером. Если произошел таймаут, то выполнение функции завершается с отрицательным кодом.
5. С помощью команды **USB10_CMD_SEND_DATA** отправляется пакет с данными **DATA0** содержащий 8 байт данных запроса.
6. Ожидается прием подтверждающего пакета путем опроса флага **USB10_STATUS_RECEIVED_BIT** регистра статуса хост-контроллера. Если произошел таймаут, то повторяется попытка отправить токен и пакет данных запроса еще раз, т. е. переход к шагу 3. Если попытки исчерпаны (всего 8 попыток), то функция завершается с отрицательным кодом.
7. Анализируется тип принятого пакета. Если это не ACK, то предполагается, что на приемной стороне пакет принялся со сбоем, поэтому производится следующая попытка отправить токен и пакет запроса, т. е. переход к шагу 3. Если попытки исчерпаны, то функция завершается с отрицательным кодом.
8. Если получен подтверждающий пакет ACK, то исполнение функции переходит в стадию приема ответа.
9. Вычисляется число пакетов ответа **num_of_data_packets** которые необходимо будет принять, а также максимальный размер принимаемого пакета в битах **packet_size_bits**.
10. С помощью команды **USB10_CMD_SEND_LONG_TOKEN** отправляется токен IN содержащий все тот же адрес устройства **address** и тот же номер конечной точки **endpoint**.
11. Ожидается получение пакета путем мониторинга состояния флага **USB10_STATUS_RECEIVED_BIT** регистра статуса хост-контроллера.
12. Анализируется тип полученного пакета. Если это пакет типа STALL, то на устройстве произошла ошибка требующая выполнения полного сброса. Этот состояние индицируется соответствующим кодом ошибки с отрицательным номером.

13. Проверяется бит **USB10_STATUS_CRC16_OK_BIT** регистра статуса индицирующего валидность кода CRC16 принятого пакета.
14. Если этот бит установлен (принят валидный пакет с данными), то с помощью команды **USB10_CMD_SEND_SHORT_TOKEN** моментально отправляется подтверждающий пакет ACK.
15. Если произошла ошибка CRC16, то производится повторная попытка отправить токен IN для получения данных ответа, т. е. переход на шаг 9.
16. Если пакет с данными принят корректно, то анализируется его тип (**DATA0** или **DATA1**) и сравнивается со значением занесенным в массив **usb10_last_data_pid**. Если они совпадают, то предполагается что получен дубликат. В этом случае принятые данные игнорируются и производится повторный цикл приема, т. е. переход на шаг 9.
17. Если PID принятого пакета не совпадает с тем, что занесено в **usb10_last_data_pid**, то считается что принят новый пакет с данными ответа, данные переносятся в буфер приема **response_data**, а PID полученного пакета сохраняется в **usb10_last_data_pid**.
18. Процесс повторяется (переход на шаг 9) пока не будет получено указанное число пакетов, либо не будет принят пакет с размером блока полезных данных меньше максимально возможного (или нулевого размера). Это рассматривается как конец обмена.
19. Функция завершается с положительным кодом возврата равным числу принятых байт ответа.

Листинг 9.10. Функция выполнения транзакции SETUP для отправки запроса и получения ответа.

```
#define USB10_SETUP_REQUEST_STR "usb10_setup_request"

int usb10_setup_request(USB10_Reg* reg, uint8_t address, uint8_t endpoint,
        uint8_t *request_data, uint8_t* response_data, uint32_t response_size)
{

        if(address >= MAX_ADDRESSES)
                return -999;

        // Wait till bus is free
        if(!usb10_wait_while_busy(reg, 10000)) { // ~1ms timeout
                usb10_printf("%s: bus is busy for too long!\r\n", USB10_SETUP_REQUEST_STR);
                return -1;
        }

        usb10_printf("%s: request %02X %02X to addr = %d\r\n", USB10_SETUP_REQUEST_STR,
                request_data[0], request_data[1], address);

        csr_clear(mstatus, MSTATUS_MIE); // Disable Machine interrupts during I/O

        uint32_t rx_status;
        int timeout;
        int retry = 3;
        int ret = 0;
        int bytes_received = 0;

        usb10_last_data_pid[address][endpoint].sent = USB10_PID_DATA1;
        usb10_last_data_pid[address][endpoint].recv = USB10_PID_DATA0;

        again_setup:

        // Prepare DATA to be sent
        usb10_write_reg(&reg->SEND_DATA_LOW, *(uint32_t*)(request_data + 0));
        usb10_write_reg(&reg->SEND_DATA_HIGH, *(uint32_t*)(request_data + 4));

        int data_pid = (usb10_last_data_pid[address][0].sent == USB10_PID_DATA0) ?
                USB10_PID_DATA1 : USB10_PID_DATA0;

        // SETUP I/O with 0:0
        usb10_write_reg(&reg->COMMAND, USB10_CMD_START_BIT |
```

```
                        USB10_CMD_SET_PID(USB10_PID_SETUP) |
                        USB10_CMD_SET_ADDR(address) |
                        USB10_CMD_SET_ENDP(endpoint) |
                        USB10_CMD_SET(USB10_CMD_SEND_LONG_TOKEN));


        if(!usb10_wait_cmd_complete(reg, 20000)) { // ~2ms timeout
                usb10_printf("%s: hung after %s packet!\r\n", USB10_SETUP_REQUEST_STR,
"SETUP");
                ret = -2;
                goto fail;
        }

        usb10_write_reg(&reg->COMMAND, USB10_CMD_START_BIT |
                        USB10_CMD_SET_LEN(8*8-1) |
                        USB10_CMD_SET_PID(data_pid) |
                        USB10_CMD_SET(USB10_CMD_SEND_DATA));

        if(!usb10_wait_cmd_complete(reg, 5000)) { // ~0.5ms timeout
                usb10_printf("%s: hung after %s packet!\r\n", USB10_SETUP_REQUEST_STR,
"DATA");
                ret = -3;
                goto fail;
        }

        timeout = 2500;
        while(timeout--)
                if(usb10_read_reg(&reg->STATUS) & USB10_STATUS_RECEIVED_BIT)
                        goto rcvd1;

        {
                usb10_printf("%s: no response after %s(%d:%d)\r\n", USB10_SETUP_REQUEST_STR,
                        "DATA", address, 0);

                if(--retry)
                        goto again_setup;

                ret = -4;
                goto fail;
        }

        rcvd1:


        usb10_last_data_pid[address][0].sent = data_pid; // remember last used DATA PID

        rx_status = usb10_read_reg(&reg->RX_STATUS); // read once, use many times

        if(USB10_RX_STATUS_PID(rx_status) != USB10_PID_ACK) {
                usb10_printf("%s: expected %s packet instead of 0x%02X, RX_STATUS =
%08X\r\n", USB10_SETUP_REQUEST_STR, "ACK",
                        USB10_RX_STATUS_PID(rx_status), rx_status);

                if(--retry)
                        goto again_setup;

                ret = -5;
                goto fail;
        }

        //delay_us(10); // Let device process request

        int num_of_data_packets = response_size % USB10_LOW_SPEED_DATA_SIZE ?
                response_size / USB10_LOW_SPEED_DATA_SIZE + 1 : response_size /
USB10_LOW_SPEED_DATA_SIZE;

        for(int i = 0; i < num_of_data_packets; i++) {

                // Request part of response data

                retry = 8;

                int packet_size_bits = (i + 1) * USB10_LOW_SPEED_DATA_SIZE <=
response_size ?
                                (USB10_LOW_SPEED_DATA_SIZE * 8 + 8 + 16) : // PID8 + DATA*8
+ CRC16
                                (response_size % USB10_LOW_SPEED_DATA_SIZE) * 8 + 8 + 16;
```

```
                again_data:

                usb10_write_reg(&reg->COMMAND, USB10_CMD_START_BIT |
                                USB10_CMD_SET_PID(USB10_PID_IN) |
                                USB10_CMD_SET_ADDR(address) |
                                USB10_CMD_SET_ENDP(0) |
                                USB10_CMD_SET(USB10_CMD_SEND_LONG_TOKEN));

                timeout = 2500;
                while(timeout--)
                        if(usb10_read_reg(&reg->STATUS) & USB10_STATUS_RECEIVED_BIT)
                                goto rcvd2;

                {
                        usb10_printf("%s: no response after %s(%d:%d)\r\n",
USB10_SETUP_REQUEST_STR,
                                "IN", address, 0);

                        if(retry--)
                                goto again_data;

                        ret = -6;
                        goto fail;
                }

                rcvd2:

                rx_status = usb10_read_reg(&reg->RX_STATUS); // read once, use many times

                int received_pid = USB10_RX_STATUS_PID(rx_status);

                if(received_pid == USB10_PID_STALL) { // STALL indicates fatal error
                        ret = -12;
                        goto fail;
                }

                // Note: NAK packets missing CRC16 which also considered as fail

                #if(1)
                if(!(usb10_read_reg(&reg->STATUS) & USB10_STATUS_CRC16_OK_BIT)) {
                        usb10_printf("%s: CRC16 error (%d), STATUS = %08X, RX_STATUS = %08X,
RX_STATUS2 = %08X, DATA = %08X:%08X\r\n",
                                USB10_SETUP_REQUEST_STR, i, usb10_read_reg(&reg->STATUS),
rx_status,
                                usb10_read_reg(&reg->RX_STATUS2), usb10_read_reg(&reg-
>RECV_DATA_HIGH),
                                usb10_read_reg(&reg->RECV_DATA_LOW)
                        );


                        delay_us(20);

                        if(retry--)
                                goto again_data;

                        ret = -10;
                        goto fail;
                }
                #endif

                // Send ACK for received packet

                usb10_write_reg(&reg->COMMAND, USB10_CMD_START_BIT |
                                USB10_CMD_SET_PID(USB10_PID_ACK) |
                                USB10_CMD_SET(USB10_CMD_SEND_SHORT_TOKEN));

                if(!usb10_wait_cmd_complete(reg, 20000)) { // 2ms timeout
                        usb10_printf("%s: hung after %s packet!\r\n",
USB10_SETUP_REQUEST_STR, "ACK");
                        ret = -7;
                        goto fail;
                }

                // Now analyze what we've got here

                if(received_pid != USB10_PID_DATA0 && received_pid != USB10_PID_DATA1) {
```

```
                        usb10_printf("%s: expected %s packet instead of 0x%02X, RX_STATUS =
%08X\r\n", USB10_SETUP_REQUEST_STR, "DATA",
                                USB10_RX_STATUS_PID(rx_status), rx_status);

                        if(retry--)
                                goto again_data;

                        ret = -8;
                        goto fail;
                }

                if(received_pid == usb10_last_data_pid[address][0].recv) {

                        usb10_printf("%s: received %d dupe data, RX_STATUS = %08X, DATA =
%08X:%08X\r\n",
                                USB10_SETUP_REQUEST_STR, i, rx_status,
                                usb10_read_reg(&reg->RECV_DATA_HIGH),
usb10_read_reg(&reg→RECV_DATA_LOW)
                        );

                        if(retry--)
                                goto again_data;

                        ret = -9;
                        goto fail;
                }

                usb10_printf("%s: packet received (%d), RX_STATUS = 0x%08X, DATA = %08X:
%08X\r\n",
                        USB10_SETUP_REQUEST_STR, i, rx_status,
                        usb10_read_reg(&reg->RECV_DATA_HIGH), usb10_read_reg(&reg-
>RECV_DATA_LOW)
                );

                usb10_last_data_pid[address][0].recv = received_pid; // Remember last
received DATA PID

                if(response_size && response_data) { // collect data if response expected
                        *(uint32_t*)(response_data + i * 8 + 0) = usb10_read_reg(&reg-
>RECV_DATA_LOW);
                        *(uint32_t*)(response_data + i * 8 + 4) = usb10_read_reg(&reg-
>RECV_DATA_HIGH);
                }

                bytes_received += (USB10_RX_STATUS_LEN(rx_status) - 8 - 16) / 8;

                #if(1) // Enable this if packet size matching is necessary
                if(USB10_RX_STATUS_LEN(rx_status) != packet_size_bits) {
                        usb10_printf("%s: received bogus data packet (%d) size: %d bits !=
%d, DATA = %08X:%08X\r\n",
                                USB10_SETUP_REQUEST_STR, i, USB10_RX_STATUS_LEN(rx_status),
packet_size_bits,
                                usb10_read_reg(&reg->RECV_DATA_HIGH), usb10_read_reg(&reg-
>RECV_DATA_LOW)
                        );

                        break;
                }
                #endif

        }


        usb10_printf("%s: RX %d bytes of DATA: ", bytes_received, USB10_SETUP_REQUEST_STR);

        for(int i = 0; i < bytes_received; i++)
                usb10_printf("%02X ", response_data[i]);

        usb10_printf("\r\n", 0);

        ret = bytes_received;

        fail:

        csr_set(mstatus, MSTATUS_MIE); // Enable Machine interrupts

        return ret;
```

```
}
```

6.3. Функции управления устройством

С помощью описанных выше базовых блоков несложно реализовать набор функций для взаимодействия с устройством: считывание конфигурации или установки параметров. Для реализации примитивного драйвера нам потребуется всего две таких функции: **usb10_get_device_descriptor()** для запроса структуры «Device Descriptor», и функция **usb10_set_value()** для установки произвольного параметра в заданное значение.

Обе эти функции используют структуру **USB10_SetupRequest** которая описывает произвольный запрос к устройству. Зададим определение этой структуры в файле **usb10.h** следующим образом:

Листинг 9.11. Структура стандартного запроса USB10_SetupRequest

```
typedef struct {
        uint8_t bmRequestType;  // D7: 0 - host->device, D6-5: 00 - standard, 01 - class,
D4-0 - recipient: 0 - Device, 1 - Interface
        uint8_t bRequest;
        uint16_t wValue;        // Extra parameter to the request
        uint16_t bIndex;        // LANGID, Endpoint number, Config number, Interface number
        uint16_t wLength;
} USB10_SetupRequest;
```

Данная структура имеет размер ровно 8 байт и помещается в один пакет DATAx для USB 1.0. Структура запроса **USB10_SetupRequest** всегда отправляется в устройство посредством исполнения транзакции SETUP, то есть с помощью описанной выше функции **usb10_setup_request()**. Запросы могут быть как с ответом, так и без него. Если на запрос не предполагается получение ответа, то в параметр response_data передается указатель **NULL**, а параметр размера ответа **response_size** устанавливается в **0**.

6.3.1. Функции usb10_get_device_descriptor() для считывания структуры «Device Descriptor»

Функция **usb10_get_device_descriptor()** достаточно проста в реализации, её код на языке Си приведен в листинге 9.12. Весь алгоритм сводится к заполнению полей структуры запроса **USB10_SetupRequest**, отправки этого запроса и получению ответа с помощью **usb10_setup_request()**, и отправке признака завершения обмена — пустого пакета с токеном OUT с помощью **usb10_out_request()**.

Более подробно формат обмена при запросе конфигурационной структуры обсуждался в главе *3.7.1. «Первый сброс и запрос структуры «Device Descriptor»*.

Листинг 9.12. Функции usb10_get_device_descriptor() для считывания структуры «Device Descriptor».

```
#define USB10_GET_DEVICE_DESCR_STR      "usb10_get_device_descr"

int usb10_get_device_descriptor(USB10_Reg* reg, uint8_t address, uint8_t endpoint, uint8_t
descr_type, uint8_t descr_item,
        uint16_t resp_len, void* descr_resp)
{
        int ret = 0;
        int bytes_read = 0;

        USB10_SetupRequest request;
```

```
        request.bmRequestType = 0x80; // device->host, standard, device
        request.bRequest = 0x06; // GET_DESCRIPTOR
        request.wValue = (descr_type << 8) | descr_item;
        request.bIndex = 0; // LANGID
        request.wLength = resp_len;

        usb10_printf("%s: %s\r\n", USB10_GET_DEVICE_DESCR_STR, "begin");

        // Begin of transaction
        if((ret = usb10_setup_request(USB1, address, endpoint, (uint8_t*)&request,
(uint8_t*)descr_resp, resp_len)) < 0) {
                usb10_printf("%s(%d): setup ret = %d\r\n", USB10_GET_DEVICE_DESCR_STR,
descr_type, ret);
                return ret;
        }

        bytes_read = ret;

        // End of transaction
        if((ret = usb10_out_request(USB1, address, endpoint, NULL, 0)) < 0) {
                usb10_printf("%s: out ret = %d\r\n", USB10_GET_DEVICE_DESCR_STR, ret);
                return ret;
        }
        usb10_printf("%s: %s\r\n", USB10_GET_DEVICE_DESCR_STR, "ok");

        return ret < 0 ? ret : bytes_read;
}
```

6.3.2. Функции usb10_set_value() для установки значения параметра

Функция **usb10_set_value()** еще проще. Она заполняет структуру стандартного запроса **USB10_SetupRequest**, в том числе помещает в неё значение устанавливаемого параметра, и выполняет запрос с помощью **usb10_setup_request()**. Реализация данной функции на языке Си приведен в листинге 9.13. Завершение обмена осуществляется отправкой пустого пакета с токеном IN с помощью **usb10_in_request()**.

Функция **usb10_set_value()** реализует зарос типа SET_VALUE который активно используется в процесс настройки устройства. Более детально формат обмена данного запроса обсуждался в главе *3.7.2. «Второй сброс и запрос присвоения адреса «Device SET_ADDRESS Request»*.

Листинг 9.13. Функции usb10_set_value() для установки значения параметра.

```
#define USB10_SET_VALUE_STR     "usb10_set_value"

int usb10_set_value(USB10_Reg* reg, uint8_t address, uint8_t endpoint, uint8_t bmRequestType,
uint8_t bRequest,
        uint16_t wValue, uint8_t bIndex, uint16_t wLength)
{
        int ret = 0;

        USB10_SetupRequest request;

        request.bmRequestType = bmRequestType;
        request.bRequest = bRequest;
        request.wValue = wValue;
        request.bIndex = bIndex;
        request.wLength = wLength;

        usb10_printf("%s: %s\r\n", USB10_SET_VALUE_STR, "begin");

        // Begin of transaction
        if((ret = usb10_setup_request(USB1, address, endpoint, (uint8_t*)&request, NULL, 0)) < 0) {
                usb10_printf("%s: setup ret = %d\r\n", USB10_SET_VALUE_STR, ret);
                return ret;
        }

        // End of transaction
        for(int i = 0; i < 5; i++) {
                ret = usb10_in_request(USB1, address, endpoint, NULL, 0);
```

```
                if(ret > 0)
                        break;

                if(ret == -8)
                        continue; // NAK, try again

                usb10_printf("%s: in ret = %d\r\n", USB10_SET_VALUE_STR, ret);
                return ret;
        }

        usb10_printf("%s: %s\r\n", USB10_SET_VALUE_STR, "ok");

        return ret;
}
```

6.4. Функция usb10_init() для детектирования и инициализации устройства

Процедура инициализации устройства достаточно подробно разобрана в главе *3.6. «Процедура инициализация USB устройства»* и, по сути, она состоит из последовательности вызовов описанных выше функций. Но перед там как мы приступим к её реализации, нам потребуется завести несколько структур данных для сохранения получаемых от устройства конфигурационных структур, таких как «Device Descriptor», «Configuration Descriptor», «Interface Descriptor» и «Endpoint Descriptor».

Сначала опишем их в заголовочном файле **usb10.h**, сразу после **USB10_SetupRequest**, следующим образом:

Листинг 9.14. Конфигурационные структуры USB устройств

```
typedef struct {
        uint8_t bLength;                // 1 Length of this descriptor = 18 bytes
        uint8_t bDescriptorType;        // 1 Descriptor type = DEVICE (01h)
        uint16_t bcdUSB;                // 2 USB specification version (BCD)
        uint8_t bDeviceClass;           // 1 Device class
        uint8_t bDeviceSubClass;        // 1 Device subclass
        uint8_t bDeviceProtocol;        // 1 Device Protocol
        uint8_t bMaxPacketSize0;        // 1 Max Packet size for endpoint 0
        uint16_t idVendor;              // 2 Vendor ID (or VID, assigned by USB-IF)
        uint16_t idProduct;             // 2 Product ID (or PID, assigned by the manufacturer)
        uint16_t bcdDevice;             // 2 Device release number (BCD)
        uint8_t iManufacturer;          // 1 Index of manufacturer string
        uint8_t iProduct;               // 1 Index of product string
        uint8_t iSerialNumber;          // 1 Index of serial number string
        uint8_t bNumConfigurations;     // 1 Number of configurations supported
} USB10_DeviceDescriptor;

typedef struct {
        uint8_t bLength;                // 1 Length of this descriptor = 9 bytes
        uint8_t bDescriptorType;        // 1 Descriptor type = CONFIGURATION (02h)
        uint16_t wTotalLength;          // 2 Total length including interface and endpoint
descriptors
        uint8_t bNumInterfaces;         // 1 Number of interfaces in this configuration
        uint8_t bConfigurationValue;    // 1 Configuration value used by SET_CONFIGURATION to select
this configuration
        uint8_t iConfiguration;         // 1 Index of string that describes this configuration
        uint8_t bmAttributes;           // 1 Bit 7: Reserved (set to 1), Bit 6: Self-powered, Bit 5:
Remote wakeup
        uint8_t bMaxPower;              // 1 Maximum power required for this configuration (in 2 mA
units)
} USB10_ConfigurationDescriptor;


typedef struct {
        uint8_t bLength;                // 1 Length of this descriptor = 9 bytes
        uint8_t bDescriptorType;        // 1 Descriptor type = INTERFACE (04h)
```

```
        uint8_t bInterfaceNumber;       // 1 Zero based index of this interface
        uint8_t bAlternateSetting;      // 1 Alternate setting value
        uint8_t bNumEndpoints;          // 1 Number of endpoints used by this interface (not
including EP0)
        uint8_t bInterfaceClass;        // 1 Interface class
        uint8_t bInterfaceSubclass;     // 1 Interface subclass
        uint8_t bInterfaceProtocol;     // 1 Interface protocol
        uint8_t iInterface;             // 1 Index to string describing this interface
} USB10_InterfaceDescriptor;

typedef struct {
        uint8_t bLength;                // 1 Length of this descriptor = 7 bytes
        uint8_t bDescriptorType;        // 1 Descriptor type = ENDPOINT (05h)
        uint8_t bEndpointAddress;       // 1
              // Bit 3...0: The endpoint number
              // Bit 6...4: Reserved, reset to zero
              // Bit 7: Direction. Ignored for Control
              //        0 = OUT endpoint
              //        1 = IN endpoint
        uint8_t bmAttributes;           // 1
              // Bits 1..0: Transfer Type
              //        00 = Control
              //        01 = Isochronous
              //        10 = Bulk
              //        11 = Interrupt
              // If not an isochronous endpoint, bits 5...2 are reserved and must be
              // set to zero. If isochronous, they are defined as follows:
              // Bits 3..2: Synchronization Type
              //        00 = No Synchronization
              //        01 = Asynchronous
              //        10 = Adaptive
              //        11 = Synchronous
              // Bits 5..4: Usage Type
              //        00 = Data endpoint
              //        01 = Feedback endpoint
              //        10 = Implicit feedback Data endpoint
              //        11 = Reserved
        uint16_t wMaxPacketSize;        // 2 Maximum packet size for this endpoint
        uint8_t bInterval;              // 1 Polling interval in milliseconds for interrupt
endpoints
                // (1 for isochronous endpoints, ignored for control or bulk)
} USB10_EndpointDescriptor;
```

Так как наш драйвер очень минималистичный и рассчитан на работу только с одним USB устройством на шине, то эти структуры будем хранить в области глобальных данных. Создадим для каждой из структур по одной глобальной переменной. Поместим их описание в заголовке файла **usb10.c** и тут же проинициализируем нулевыми значениями:

```
uint8_t usb10_device_address = 0;
USB10_DeviceDescriptor usb10_device_descr = {0};
USB10_ConfigurationDescriptor usb10_config_descr = {0};
USB10_InterfaceDescriptor usb10_interface_descr = {0};
USB10_EndpointDescriptor usb10_endpoint_descr = {0};
```

Добавим объявления этих переменных в файл **usb10.h:**

```
extern uint8_t usb10_device_address;    // last used device address
extern USB10_DeviceDescriptor usb10_device_descr;
extern USB10_ConfigurationDescriptor usb10_config_descr;
extern USB10_InterfaceDescriptor usb10_interface_descr;
extern USB10_EndpointDescriptor usb10_endpoint_descr;
```

Аналогичным образом добавим глобальную переменную **usb10_device_address** которая будет содержать текущий адрес присвоенный подключенному к USB шине устройство. Каждый раз, когда устройство будет отключаться и подключаться заново, будем увеличивать значение этой переменной на 1.

Теперь у нас все готово для реализации функции **usb10_init()**. Функция получает на вход указатель **reg** ссылающийся на область памяти регистров хост-контроллера, указатель **new_device_address** на переменную в которую будет помещен адрес присвоенный вновь подключенному USB устройству, и четыре указателя на указатели (**device_resp**, **config_resp**, **interface_resp** и **endpoint_resp**) ссылающиеся на блоки памяти конфигурационных структур, которые будут заполнены различными данными полученными от устройства в процессе инициализации.

Функция **usb10_init()** возвращает 0 если в процессе работы ей удалось обнаружить и полностью проинициализировать новое устройство. Также она изменяет глобальную переменную **usb10_device_address**, запоминая в ней адрес USB устройства. Если в процесс инициализации устройства происходит ошибка (или устройство отсутствует на шине), то данная переменная сбрасывается в 0. Далее, в коде приложения, мы сможем использовать данную переменную как флаг: если она равна нулю, то нужно вызвать **usb10_init()**, иначе — можно работать с устройством (опрашивать или передавать ему команды).

Функция **usb10_init()** придерживается следующего алгоритма:

1. Выполняет первый сброс шины путем вызова **usb10_bus_reset()**.
2. С помощью **usb10_get_device_descriptor()** делается попытка считать с устройства 0 сокращенную версию (18 байт) конфигурационной структуру «Device Description». Если попытка завершилась неудачно, то предполагается что на шине нет нового подключенного устройства и работа функции завершается с отрицательным кодом.
3. Если чтение структуру «Device Description» с нулевого устройства прошло успешно, то это признак того, что на шине имеется **неинициализированное** устройство и процесс продолжается.
4. Выполняет второй сброс шины путем вызова **usb10_bus_reset()**.
5. С помощью **usb10_set_value()** устройству присваивается новый адрес равный **usb10_device_address + 1**.
6. С помощью **usb10_get_device_descriptor()** повторно запрашивается структура «Device Descriptor», но уже от устройства с новым адресом. Структура читается полностью (255 байт).
7. Производится разбор считанной структуры, выделяются и копируются вложенные структуры.
8. С помощью **usb10_set_value()** устанавливается номер активной конфигурации **1**.
9. Заполняются указатели **new_device_address**, **device_resp**, **config_resp**, **interface_resp** и **endpoint_resp**, и выполнение завершается с нулевым кодом.

Как уже неоднократно отмечалось выше по тексту, данный алгоритм инициализации выполняет минимально достаточные действия для того, чтобы можно было взаимодействовать с устройством на программном уровне. Далее мы рассмотрим как опрашивать устройство (получать с него данные) или посылать ему команды из приложения.

Листинг 9.15. Функции usb10_init() для детектирования и инициализации устройства.

```
#define usb10_init_STR  "usb10_init"

int usb10_init(USB10_Reg* reg, uint8_t* new_device_address,
        USB10_DeviceDescriptorUnion **device_resp,
        USB10_ConfigurationDescriptorUnion **config_resp,
        USB10_InterfaceDescriptorUnion **interface_resp,
        USB10_EndpointDescriptorUnion **endpoint_resp)
{
        /* NOTE: descr_resp, config_resp, interface_resp and endpoint_resp can be NULL
pointers,
           so do not use these here, use global data structures instead! */

        int ret = 0;

        usb10_printf("\r%s: Scanning for devices...\r\n", usb10_init_STR);

        if((ret = usb10_bus_reset(USB1, 12000)) < 0)
                goto usb10_error;

        if((ret = usb10_get_device_descriptor(USB1, USB10_UNNUMBERED_DEVICE, USB10_EP0,
USB10_DESCR_TYPE_DEVICE,
                                        0, 18, &usb10_device_descr)) < 0)
                goto usb10_error;

        //usb10_printf("\r%s: Device detected: VID/PID = 0x%04X/0x%04X, "
        printf("\r%s: Device detected: VID/PID = 0x%04X/0x%04X, "
                "class/subclass = 0x%02X/0x%02X, bcdUSB = 0x%04X\r\n",
                usb10_init_STR,
                usb10_device_descr.device.idVendor,
                usb10_device_descr.device.idProduct,
                usb10_device_descr.device.bDeviceClass,
                usb10_device_descr.device.bDeviceSubClass,
                usb10_device_descr.device.bcdUSB);

        if((ret = usb10_bus_reset(USB1, 12000)) < 0)
                goto usb10_error;
        int device_address = (usb10_device_address + 1) % MAX_ADDRESSES;

        // Assign new device address to device_address
        if((ret = usb10_set_value(USB1, USB10_UNNUMBERED_DEVICE, USB10_EP0, 0x00,
USB10_REQ_SET_ADDRESS,
                                device_address, 0, 0)) < 0)
                goto usb10_error;

        // Request as many as possible description data

        uint8_t bulk_config_data[256];

        if((ret = usb10_get_device_descriptor(USB1, device_address, USB10_EP0,
USB10_DESCR_TYPE_CONFIG,
                                        0, 255, bulk_config_data)) < 0)
                goto usb10_error;

        usb10_printf("%s: total config size = %d\r\n", usb10_init_STR, ret);

        uint8_t *p = bulk_config_data;
        int bytes_parsed = 0;

        // Mark old descriptors as obsolete

        *(uint8_t*)&usb10_config_descr = 0;
        *(uint8_t*)&usb10_interface_descr = 0;
        *(uint8_t*)&usb10_endpoint_descr = 0;

        while(bytes_parsed < ret) {

                usb10_printf("%s: descriptor type = %d, size = %d\r\n", usb10_init_STR,
p[1], p[0]);

                // Save first ocurance as default descriptor of its type

                switch(p[1]) {
                        case 0x02:      // Config
                                if(*(uint8_t*)&usb10_config_descr == 0)
```

```
                                        memcpy(&usb10_config_descr, p,
sizeof(usb10_config_descr));
                                break;
                        case 0x04:      //Interface
                                if(*(uint8_t*)&usb10_interface_descr == 0)
                                        memcpy(&usb10_interface_descr, p,
sizeof(usb10_interface_descr));
                                break;
                        case 0x05:      // Endpoint
                                if(*(uint8_t*)&usb10_endpoint_descr == 0)
                                        memcpy(&usb10_endpoint_descr, p,
sizeof(usb10_endpoint_descr));
                                break;
                }

                bytes_parsed += p[0];
                p += p[0];
        }

        // Activate 1st configuration
        if((ret = usb10_set_value(USB1, device_address, USB10_EP0, 0x00,
USB10_REQ_SET_CONFIG, 1, 0, 0)) < 0)
                goto usb10_error;

        usb10_device_address = device_address;

        //usb10_printf("%s: Device addr = %d, Config: bLength = %d "
        printf("%s: Address = %d, Config wTotal = %d, "
                "Interface Class/Subclass/Proto = %d/%d/%d, "
                "EPAddress = 0x%02X, Interval = %d ms, MaxPacket = %d\r\n",
                usb10_init_STR,
                usb10_device_address,
                usb10_config_descr.config.wTotalLength,
                usb10_interface_descr.iface.bInterfaceClass,
                usb10_interface_descr.iface.bInterfaceSubclass,
                usb10_interface_descr.iface.bInterfaceProtocol,
                usb10_endpoint_descr.endp.bEndpointAddress,
                usb10_endpoint_descr.endp.bInterval,
                usb10_endpoint_descr.endp.wMaxPacketSize);

        ok:
                // Fill-in returning data structures

        if(new_device_address)
                *new_device_address = usb10_device_address;

        if(device_resp)
                *device_resp = &usb10_device_descr;

        if(config_resp)
                *config_resp = &usb10_config_descr;

        if(interface_resp)
                *interface_resp = &usb10_interface_descr;

        if(endpoint_resp)
                *endpoint_resp = &usb10_endpoint_descr;

        usb10_printf("%s: %s\r\n", usb10_init_STR, "ok");

        return 0;

        usb10_error:

        usb10_device_address = 0; // this can be used as flag to indicate device is ready

        usb10_printf("%s: %s\r\n", usb10_init_STR, "fail");

        return ret;
}
```

7. Взаимодействие приложения с USB 1.0 устройством

Выше уже отмечалось, что самым простым типом USB устройств являются USB HID Class устройства такие как клавиатура, мышь или джойстик. В главе *3.8. «Device Class Definition for Human Interface Devices» (HID)*» приведено детальное описание этого расширения протокола USB. Напомню, что большинство USB HID устройств могут работать в двух режимах «Report Protocol» и «Boot Protocol». Последний является очень простым с точки зрения интерпретации получаемых данных — в «Boot Protocol» все значения от Usage IDs находятся на строго предопределенных фиксированных позициях в единственном пакете получаемых данных, а значит распарсить такой пакет не составит труда. Далее мы попробуем сделать это для трех упомянутых HID устройств.

7.1. Тестовое приложение karnix_usb10_test

С целью протестировать описанный выше драйвер разработанного нами USB хост-контроллера, подготовим небольшое тестовое приложение **karnix_usb10_test**. Эта простейшая программа на языке Си сначала проинициализирует разработанный нами хост-контроллер, после чего вызовет функцию **usb10_init()** для выполнения процедуры инициализации устройства и получения от него набора структур конфигурационных дескрипторов. Затем, в цикле будет опрашивать устройство формируя транзакцию «Interrupt IN» с частотой не меньшей чем того требуется устройством в параметре **bInterval** структуры «Endpoint Description». Полученные данные будут отображаться на терминал в шестнадцатеричном виде.

Начнем с того, что создадим подкаталог для тестового приложения **./src/main/c/karnix/karnix_usb10_test** и скопируем в него заготовку **Makefile**-а из подкаталога **./src/main/c/karnix/karnix_soc**:

```
rz@devbox:~/KarnixSOC$ mkdir -p src/main/c/karnix/karnix_usb10_test
rz@devbox:~/KarnixSOC$ cd src/main/c/karnix/karnix_usb10_test
rz@devbox:~/KarnixSOC/src/main/c/karnix/karnix_usb10_test$ cp ../karnix_soc/Makefile ./
```

Отредактируем **Makefile** так, чтобы его заглавная часть имела следующий вид:

Листинг 10.1. Заголовок модифицированного Makefile-а для сборки тестового приложения.

```
rz@devbox:~/KarnixSOC/src/main/c/karnix/karnix_usb10_test$ head -30 Makefile

PROJ_NAME=karnix_usb10_test
#MODEL ?= ram
MODEL ?= xip
#
DEBUG=no
BENCH=no
MULDIV=yes
ATOMIC=yes
FPU=yes
COMPRESSED=yes

SOCDIR=../karnix_soc/src

CRT_RAM=$(SOCDIR)/crt_ram.S
LDSCRIPT_RAM=$(SOCDIR)/linker_ram.ld -Wl

CRT_XIP=$(SOCDIR)/crt_xip.S
LDSCRIPT_XIP=$(SOCDIR)/linker_xip.ld -Wl


SRCS += $(wildcard src/*.c)
SRCS += $(SOCDIR)/utils.c
SRCS += $(SOCDIR)/usb10.c
```

```
SRCS += $(SOCDIR)/memops.S
SRCS += $(CRT)

################################################################################

OBJDIR = build
…
```

В этом сборочном скрипте мы подключаем в сборку драйвер **usb10.c**, набор «утилитных» функций для работы с терминалом **utils.c** и оптимизированную под СнК реализацию функций memcpy/memmove из файла **memops.S**.

Полный текст сборочного файла ./src/main/c/karnix/karnix_usb10_test/Makefile можно получить из репозитория.

Теперь внутри каталога **./src/main/c/karnix/karnix_usb10_test/** создадим подкаталог **./src/** для размещения исходного кода программы:

```
rz@devbox:~/KarnixSOC/src/main/c/karnix/karnix_usb10_test$ mkdir -p src
```

и поместим в него файл **main.c** с текстом будущей тестовой программы.

Написание программы начнем как обычно с подключения требуемых заголовочных файлов. Среди прочих нам потребуется заголовочный файла с описаниями функций драйвера **usb10.h**:

Листинг 10.2. Исходный код программы karnix_usb10_test (файл main.c).

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include "soc.h"
#include "plic.h"
#include "riscv.h"
#include "usb10.h"
#include "utils.h"
```

Добавим ссылки на глобальные символы необходимые для инициализации библиотеки LIBC:

```
extern void __sinit(void *);
extern unsigned int _IMPURE_DATA;
```

Добавим пару используемых в программе глобальных переменных:

```
const char *WELCOME_TEXT = "Welcome to Karnix USB 1.0 Test. Copyright (C) 2024-2025,
Fabmicro, LLC.\r\nBuild #%04u at %s %s. Main addr: %p\r\n\r\n";


uint32_t reg_sys_counter = 0;
uint32_t reg_usb_error_count = 0;
```

Переменная **WELCOME_TEXT** указывает на строку приветствия содержащую дату и время сборки программы — иногда очень полезно понимать, какая версия программы прошита и исполняется в системе.

Переменная **reg_sys_counter** будет выполнять функцию системного 1 мс счетчика необходимого для синхронизации действий программы по времени (регулярное сканирование шины USB и регулярный опрос устройства).

Переменная **reg_usb_error_count** будет выполнять функцию счетчика ошибок возникающих при работе с USB устройством. Будем использовать её значение для того, чтобы определить работоспособность USB устройства. Если значение **reg_usb_error_count** превысит **3**, то будем считать что устройство неработоспособно и требуется повторно выполнять цикл сканирования/инициализации.

Далее опишем тело функции **main()** в начале которого проведем инициализацию аппаратуры.

Листинг 10.2. Исходный код программы karnix_usb10_test (продолжение файла main.c).

```
void main() {

        csr_clear(mstatus, MSTATUS_MIE); // Disable Machine interrupts during hardware init
        PLIC->ENABLE = 0; // Disable MicroPLIC interrupts

        delay_us(2000000); // Wait for FCLK to settle

        init_sbrk(NULL, 0); // Initialize heap for malloc to use on-chip RAM
        __sinit(&_IMPURE_DATA); // Init LIBC impure_data structure

        xprintf(WELCOME_TEXT, BUILD_NUMBER, __DATE__, __TIME__, &main);

        // Enable USB1
        USB1->CONTROL &= ~USB10_CONTROL_ENABLE_BIT;
        delay_us(1000);
        USB1->CONTROL |= USB10_CONTROL_RESET_DELAY_SET(1500000 / 1000 * 10); // Set reset
duration to 11ms (num of ticks at 1.5 MHz
        USB1->CONTROL |= USB10_CONTROL_KEEPALIVE_BIT;
        USB1->CONTROL |= USB10_CONTROL_ENABLE_BIT;
        xprintf("USB1 enabled\r\n");

        csr_set(mstatus, MSTATUS_MIE); // Enable Machine interrupts
```

В приведенном выше коде происходит следующее:

1. Запрет всех аппаратных прерываний;
2. Отключение контроллера PLIC, так как в рамках данного приложения в нем нет необходимости);
3. Ожидание ~2 сек холостых циклов процессора с целью стабилизации системного тактового сигнала;
4. Инициализация «кучи» в с помощью функции **init_sbrk()**;
5. Инициализация глобальный структур LIBC с помощью функции **__sinit()**;
6. Вывод приветственного сообщения на терминал в UART с помощью оберточного макроса **xprintf()**;
7. Конфигурирование и разрешение работы USB хост-контроллера;
8. Вывод сообщения готовности и разрешение всех аппаратных прерываний.

В данной тестовой программе мы не будем использовать прерывания, вся работы с аппаратурой будет происходить методом постоянного периодического опроса (методом «poll-a»).

Далее введем главный цикл, в нем сразу выполним задержку в 1 мс и прирастим на единицу значение переменной счетчика циклов:

```
while(1) {
    delay_us(1000);

    reg_sys_counter++; // This is 1ms counter
```

А теперь самое интересное. Воспользовавшись глобальной переменной **usb10_device_address**, которая содержит текущий адрес подключенного к USB шине устройства (или 0 если ничего не подключено), проверим отсутствие на шине готового к работе устройства и выполним попытку инициализации путем вызова функции **usb10_init()**. Выполнять эту проверку будем один раз в секунду, то есть когда остаток от деления счетчика на 1000 равен нулю.

Листинг 10.2. Исходный код программы karnix_usb10_test (продолжение файла main.c).

```
                if(usb10_device_address == 0 && reg_sys_counter % 1000 == 0) {
                        // Perform USB bus scan each 1 sec
                        xprintf("USB bus scan: reg_sys_counter = %d... ", reg_sys_counter);

                        // No devices connected at the moment, try to init
                        // Note: usb10_device_address is gobal variable holding address of
connected device

                        int8_t ret;
                        uint8_t new_device_address;

                        if((ret = usb10_init(USB1, &new_device_address, NULL, NULL, NULL,
NULL)) == 0) {

                                // New device detected and initialized

                                reg_usb_error_count = 0;

                                xprintf("\r\nUSB1: new device addr = %d, VID/PID =
0x%04X/0x%04X, "
                                        "class/subclass/proto = %d/%d/%d, "
                                        "EndpointAddress = %d, Interval = %d, MaxPower = %d
mA\r\n",
                                        new_device_address,
                                        usb10_device_descr.idVendor,
                                        usb10_device_descr.idProduct,
                                        usb10_interface_descr.bInterfaceClass,
                                        usb10_interface_descr.bInterfaceSubclass,
                                        usb10_interface_descr.bInterfaceProtocol,
                                        usb10_endpoint_descr.bEndpointAddress & 0x0f,
                                        usb10_endpoint_descr.bInterval,
                                        usb10_config_descr.bMaxPower * 2
                                );
                        } else {
                                xprintf("failed with ret = %d\r\n", ret);
                        }
                }
```

Напомню, что функция **usb10_init()** возвращает отрицательный код ошибки если попытка найти и проинициализировать USB устройство не увенчалась успехом, или ноль если новое устройство успешно сконфигурировано и готово к работе.

В случае успешного завершения данной функции, в глобальных переменных **usb10_device_descr**, **usb10_interface_descr**, **usb10_endpoint_descr** и **usb10_config_descr** содержатся данные полученных от устройства дескрипторов «Device Descriptor», «Interface Descriptor», «Endpoint Descriptor» и «Configuration Descriptor» соответственно, часть этих данных выводится в терминал с помощью **xprintf()**. Далее мы будем анализировать содержимое этих структур чтобы выяснить с каким типом устройства мы имеем дело.

Каждые 20 мс будем проверять имеется ли на шине готовое к работе устройство (**usb10_device_address > 0**), после чего сделаем попытку его опроса с помощью транзакции «Interrupt IN». Напомню, что в режиме «Boot Protocol» USB HID устройства опрашиваются именно таким способом. Сама транзакция состоит в отправки токена IN на адрес устройства

с указанием номера конченой точки объявленного в параметре **bEndpointAddress** структуры «Endpoint Descriptor», с последующим чтением одиночного пакета данных фиксированного размера.

Чтобы выяснить размер читаемого блока воспользуемся идентификационными данными которые присланы устройством, а именно - значениями параметров **bInterfaceClass**, **bInterfaceSubclass** и **bInterfaceProtocol**. Будем действовать согласно следующему простому алгоритму:

- Если **bInterfaceClass == 3 и bInterfaceSubclass == 1 и bInterfaceProtocol == 1**, то мы имеем дело с HID устройством типа **«клавиатура»**, длина блока данных для него равна **8 байт**.

- Если **bInterfaceClass == 3 и bInterfaceSubclass == 1 и bInterfaceProtocol == 2**, то мы имеем дело с HID устройством типа **«мышь»**, размер блока данных — **4 байта**.

- Если же **bInterfaceClass == 3 и bInterfaceSubclass == 0 и bInterfaceProtocol == 0**, то к шине подключено HID устройство типа **«геймпад»**, размер блока данных — **8 байт**.

- В противном случае на шине находится устройство неизвестного типа, будем считать что оно выдает блок данных размером 8 байт.

Выяснив размер блока полезных данных сформируем IN запрос с помощью функции **usb10_in_request()**.

Если же функция **usb10_in_request()** вернет ошибку отличную от кодов **-8** (**USB10_IN_ENAK** — нет данных) и **-9** (**USB10_IN_ETIMEOUT** — таймаут), то такая ошибка будет восприниматься как фатальная, т. е. требующая полного сброса и переинициализации. В этом случае просто установим переменную **usb10_device_address** в значение **0**, что приведет к вызову функции **sub10_init()** в следующей итерации цикла.

Если же функция **usb10_in_request()** вернет положительное число, то у нас имеется блок данных от устройства - выведем его содержимое на терминал в шестнадцатеричном виде.

Изложив всё выше сказанное на языке Си получим следующий код:

Листинг 10.2. Исходный код программы karnix_usb10_test (продолжение файла main.c).

```
                if(usb10_device_address > 0 && reg_sys_counter % 20 == 0) {

                        // Perform USB "Interrupt IN" transaction every 20ms

                        uint8_t endpoint = 1; // Commonly used 1, but should be
usb10_endpoint_descr.bEndpointAddress & 0x0f
                        uint8_t response_data[8] = {0};
                        int response_size;
                        int device_type;

                        // Try to guess response data packet size using class info

                        if(usb10_interface_descr.bInterfaceClass == 3 &&
                                  usb10_interface_descr.bInterfaceSubclass == 1 &&
                                  usb10_interface_descr.bInterfaceProtocol == 1) {

                                // We have to check RX packet len (88 bits) to skip empty
packets

                                response_size = 8; // HID keyboard
                                device_type = 1;
```

```
                        } else if(usb10_interface_descr.bInterfaceClass == 3 &&
                           usb10_interface_descr.bInterfaceSubclass == 1 &&
                           usb10_interface_descr.bInterfaceProtocol == 2) {

                                response_size = 4; // HID mouse
                                device_type = 2;

                        } else if(usb10_interface_descr.bInterfaceClass == 3 &&
                                  usb10_interface_descr.bInterfaceSubclass == 0 &&
                                  usb10_interface_descr.bInterfaceProtocol == 0) {

                                response_size = 8; // HID gamepad
                                device_type = 3;

                        } else {
                                response_size = 8; // Unknown
                                device_type = 0;
                        }


                        // Poll Endpoint for new data
                        int ret = usb10_in_request(USB1, usb10_device_address, endpoint,
                                        response_data, response_size);

                        if(ret >= 0) {

                                reg_usb_error_count = 0;

                                xprintf("\rUSB1 (%d:%d) data received: ",
usb10_device_address, endpoint);

                                for(int i = 0; i < response_size; i++)
                                        xprintf("%02X ", response_data[i]);

                                xprintf(", class = %d/%d/%d, RX_STATUS: 0x%08X, RX_STATUS2:
0x%08X, STATUS: 0x%08X\r\n",
                                        usb10_interface_descr.bInterfaceClass,
                                        usb10_interface_descr.bInterfaceSubclass,
                                        usb10_interface_descr.bInterfaceProtocol,
                                        USB1->RX_STATUS, USB1->RX_STATUS2, USB1->STATUS);

                        } else if(ret == USB10_IN_ENAK || ret == USB10_IN_ETIMEOUT) { // NAK
or timeout (dupe)
                                // These are legitimate error codes for not ready device, do
nothing
                                 goto usb_end;
                        } else {
                                if(++reg_usb_error_count > 3) { // More than 3 errors in a
row means connection is broken
                                        xprintf("\rUSB1 (%d:%d) failed, ret = %d\r\n",
usb10_device_address, endpoint, ret);
                                        usb10_device_address = 0; // flag USB as broken
                                        goto usb_end;
                                }
                        }

                        // Some data processing code can be put here
                        // ...

                        usb_end:;
                }

        }
}
```

Приведенной выше тестовой программы достаточно чтобы протестировать все виды имеющихся USB HID устройств. В следующей главе мы выполним сборку этой программы, загрузим её в область NOR flash память отведенной для приложения и посмотрим как выглядят данные поступающие от HID устройств трех различных типов. Полный текст файла **main.c** с исходным кодом программы можно посмотреть в репозитории: ./src/main/c/karnix/karnix_usb10_test/src/main.c.

7.2. Сборка и запуск приложения karnix_usb10_test

Если **Makefile** создан верно, то сборка тестового приложения запускается одной командой **make** из подкаталога **./src/main/c/karnix/karnix_usb10_test/**. Результат сборки — это два бинарных (исполняемых) файла: один для размещения базовую RAM, второй — для размещения и исполнения из NOR Flash памяти. Эти файлы будет находиться в подкаталоге **./build**. Обычно процесс сборки выглядит так:

Листинг 11.1. Выдержка из вывода при сборке программы karnix_usb10_test.

```
rz@devbox:~/KarnixSOC/src/main/c/karnix/karnix_usb10_test$ make

make ram
make[1]: Entering directory '/home/rz/KarnixSOC/src/main/c/karnix/karnix_usb10_test'
*** Building for RAM ***
make MODEL=ram build_sources build/karnix_usb10_test.hexx
make[2]: Entering directory '/home/rz/KarnixSOC/src/main/c/karnix/karnix_usb10_test'
mkdir -p build/src/
/opt/xpack-riscv-none-elf-gcc-14.2.0-2/bin/riscv-none-elf-gcc -c -march=rv32imafc -mabi=ilp32f
-DNDEBUG -flto -fno-common -MD -fstrict-volatile-bitfields -fno-strict-aliasing -fno-builtin-printf
-DBUILD_NUMBER=`cat build/build_number`+1   -Os  -I./src -I -I/include -I../karnix_soc/src
-I../karnix_soc/src/include -o build/src/main.o src/main.c
…
Memory region         Used Size  Region Size  %age Used
             RAM:       14576 B        72 KB     19.77%

/opt/xpack-riscv-none-elf-gcc-14.2.0-2/bin/riscv-none-elf-objcopy -O binary
build/karnix_usb10_test_ram.elf build/karnix_usb10_test_ram.bin
…

make xip
make[1]: Entering directory '/home/rz/KarnixSOC/src/main/c/karnix/karnix_usb10_test'
*** Building for XIP ***
make MODEL=xip build_sources
make[2]: Entering directory '/home/rz/KarnixSOC/src/main/c/karnix/karnix_usb10_test'
mkdir -p build/__/karnix_soc/src/
/opt/xpack-riscv-none-elf-gcc-14.2.0-2/bin/riscv-none-elf-gcc -c -march=rv32imafc -mabi=ilp32f
-DNDEBUG -flto -fno-common -MD -fstrict-volatile-bitfields -fno-strict-aliasing -fno-builtin-printf
-DBUILD_NUMBER=`cat build/build_number`+1   -Os -o build/__/karnix_soc/src/crt_xip.o
../karnix_soc/src/crt_xip.S -D__ASSEMBLY__=1
/opt/xpack-riscv-none-elf-gcc-14.2.0-2/bin/riscv-none-elf-gcc -march=rv32imafc -mabi=ilp32f -DNDEBUG
-flto -fno-common -MD -fstrict-volatile-bitfields -fno-strict-aliasing -fno-builtin-printf
-DBUILD_NUMBER=`cat build/build_number`+1   -Os -o build/karnix_usb10_test_xip.elf build/src/main.o
build/__/karnix_soc/src/utils.o build/__/karnix_soc/src/usb10.o build/__/karnix_soc/src/memops.o
build/__/karnix_soc/src/crt_xip.o -march=rv32imafc  -mabi=ilp32f -specs=nano.specs -lnosys -lc_nano
-nostdlib -lgcc -mcmodel=medany -nostartfiles -ffreestanding
-Wl,-Bstatic,-T,../karnix_soc/src/linker_xip.ld -Wl,-Map,build/karnix_usb10_test_xip.map,--print-
memory-usage -Lbuild

Memory region         Used Size  Region Size  %age Used
           FLASH:       12088 B        15 MB      0.08%
             RAM:        2608 B        60 KB      4.24%

/opt/xpack-riscv-none-elf-gcc-14.2.0-2/bin/riscv-none-elf-objcopy -O binary
build/karnix_usb10_test_xip.elf build/karnix_usb10_test_xip.bin
```

Из приведенного выше листинга видно, что сборка разбивается на две цели: **make ram** и **make xip**, в результате получается два исполняемых файла:

```
rz@devbox:~/KarnixSOC/src/main/c/karnix/karnix_usb10_test$ ls -l build/*.bin
-rwxrwxr-x 1 rz rz 12060 Oct 15 18:31 build/karnix_usb10_test_ram.bin
-rwxrwxr-x 1 rz rz 12088 Oct 15 18:31 build/karnix_usb10_test_xip.bin
```

Файл с суффиксом **_ram.bin** предназначен для размещение в базовой памяти с адреса **0x80000000**, что задается в конфигурации линкера (**./karnix_soc/src/linker_ram.ld**), и требует для работы **14576** байт. Чтобы разместить этот бинарный код в памяти нам потребуется пересобрать весь проект с аппаратурой указав на применение **karnix_usb10_test** вместо **karnix_bootloader**. Этот процесс достаточно долгий и непрактичный. Поэтому мы воспользуемся вторым бинарным файлом.

Файл с суффиксом **_xip.bin** предназначен для размещения в область приложения на имеющейся на плате «Карно» NOR Flash памяти. Эта память Flash память отображается в адресное пространство вычислительной системы СнК начиная с адреса **0xA0000000**, при этом первый мегабайт (**0x100000** байт) занят битстримом загружаемым в ПЛИС и его нельзя использоваться под приложения. А вот область выше адреса **0xA0100000** свободна для приложений. Конфигурационный файл **../karnix_soc/src/linker_xip.ld** для линкера настроен так, чтобы приложение размещалось с этого адреса. Такой режим исполнения программы, т.е. прямо из Flash памяти, принято называть **XiP** от «eXecution in Place» (исполнение по месту размещения). Наша тестовая программа находящаяся в файле **karnix_usb10_test_xip.bin** требует **12088** байт Flash памяти под код и еще **2608** байт для переменных и стека в RAM. Область для переменных линкер выделит начиная с адреса **0x80003000** (первые **0x3000** байт RAM зарезервированы под нужды загрузчика **karnix_bootloader**).

Для того, чтобы разместить (записать) исполняемый код тестового приложения в NOR Flash воспользуемся утилитой **openFPGALoader**, выполним следующую команду:

Листинг 11.2. Вывод утилиты openFPGALoader при прошивке тестовой программы.

```
rz@devbox:~/KarnixSOC/src/main/c/karnix/karnix_usb10_test$ openFPGALoader -f -o 0xA0100000
bin/karnix_usb10_test_xip.bin

empty
write to flash
No cable or board specified: using direct ft2232 interface
unable to open ftdi device: -3 (device not found)
JTAG init failed with: unable to open ftdi device
rz@butterfly:~ % openFPGALoader -f -o 0xA0100000 karnix_usb10_test_xip.bin
empty
write to flash
No cable or board specified: using direct ft2232 interface
Jtag frequency : requested 6.00MHz   -> real 6.00MHz
Open file DONE
Parse file DONE
Enable configuration: DONE
SRAM erase: DONE
Detail:
Jedec ID          : ef
memory type       : 70
memory capacity   : 18
flash chip unknown: use basic protection detection
start addr: a0100000, end_addr: a0110000
Erasing: [==================================================] 100.00%
Done
Writing: [==================================================] 100.00%
Done
Refresh: DONE
```

Сразу после сброса и инициализации ПЛИС будет запущен загрузчик **karnix_bootloader**, он по специальной сигнатуре найдет приложение в адресном пространстве отведенном под NOR Flash и передаст ему управление. Если же во Flash ранее была загружена программа

«Монитор» (**karnix_monitor**), то управление будет передано ей и тогда необходимо воспользоваться командой **call 0xa0100000** монитора, чтобы запустить приложение.

Предположим, что наше тестовое приложение является единственным во Flash памяти. Тогда, подключив терминал к отладочному порту и нажав кнопку «RESET» на плате «Карно», мы увидим следующие сообщения:

Листинг 11.3. Вывод в отладочный порт при исполнении тестовой программы (USB устройство отсутствует).

```
rz@devbox:~/KarnixSOC/src/main/c/karnix/karnix_usb10_test$ sudo minicom -D /dev/ttyUSB1 -b 115200

Welcome to minicom 2.10

OPTIONS: I18n
Compiled on Apr 28 2025, 22:28:07.
Port /dev/ttyU1, 00:18:35 [U]

Press CTRL-A Z for help on special keys


[Bootloader] Karnix SoC Bootloader, build #31 on Aug 21 2025 at 21:43:03
[Bootloader] calling application at 0xa0100000

Welcome to Karnix USB 1.0 Test. Copyright (C) 2024-2025, Fabmicro, LLC.
Build #0011 at Oct 13 2025 19:02:21. Main addr: 0xa0100d58

USB1 enabled
USB bus scan: reg_sys_counter = 1000... failed with ret = -1
USB bus scan: reg_sys_counter = 2000... failed with ret = -1
USB bus scan: reg_sys_counter = 3000... failed with ret = -1
...
```

Из приведенного выше листинга видно, что сначала запустился загрузчик выдав приветствие «**Karnix SoC Bootload**er», он обнаружил наше приложение по адресу **0xA0100000** и передал ему управление. Приложение выдало своё приветствие «**Welcome to Karnix USB 1.0 Test**» сообщив нам что функция **main()** исполняется с адреса **0xA0100D58**, этот адрес находится в области Flash памяти.

7.3. Тестирование karnix_usb10_test с разными типами устройств

Если к USB порту не подключено устройство, то приложение будет непрерывно выдавать сообщение об ошибке вида «**failed with ret = -1** ». Код ошибки -1 возвращаемый функцией **usb10_init()** означает, что на USB шине ничего нет. Давайте подключим USB 1.0 совместимое HID устройство и посмотрим какие сообщения будет выдавать нам приложение.

7.3.1. Тестирование устройства типа «gamepad»

USB HID устройство типа «gamepad» (или по нашему «джойстик») интересно тем, что формат выдаваемых им данных очень прост: это 8 байт, каждый из которых отвечает за положение одного из устройств управления (положение «стика»), а состояние кнопок собраны в битовые поля и тоже имеют фиксированные позиции.

Подключим USB 1.0 совместимый «джойстик» и посмотрим на выдачу в отладочный порт от программы **karnix_usb10_test**:

Листинг 11.4. Вывод программы karnix_usb10_test при подключении устройства типа «gamepad».

```
usb10_init: Device detected: VID/PID = 0x0079/0x0006, class/subclass = 0x00/0x00, bcdUSB = 0x0100
usb10_init: Address = 1, Config wTotal = 41, Interface Class/Subclass/Proto = 3/0/0, EPAddress =
0x81, Interval = 10 ms, MaxPacket = 8
USB1: new device addr = 1, VID/PID = 0x0079/0x0006, class/subclass/proto = 3/0/0, EndpointAddress =
1, Interval = 10, MaxPower = 500 mA
USB1 (1:1) data received: 7F 7F 00 80 80 0F 00 00 , class = 3/0/0, RX_STATUS: 0x4A6E4B58,
RX_STATUS2: 0x00004A6E, STATUS: 0x04000001
USB1 (1:1) data received: 7F 7F 00 80 80 0F 00 00 , class = 3/0/0, RX_STATUS: 0x4A6EC358,
RX_STATUS2: 0x00004A6E, STATUS: 0x04000001
USB1 (1:1) data received: 7F 7F 00 80 80 0F 00 00 , class = 3/0/0, RX_STATUS: 0x4A6E4B58,
RX_STATUS2: 0x00004A6E, STATUS: 0x04000001
...
```

Сразу после подключения джойстика, при следующей итерации сканирования, программой было обнаружено USB устройство с классом **Class/Subclass = 0x00/0x00**. Это означает, что правильные значения класса и подкласса нужно искать в структуре «Interface Descritor», которая для данного устройства содержит три параметра **Class/Subclass/Proto** равные соответственно **3**, **0** и **0**. Согласно HID расширению это и есть устройство типа «gamepad».

Из вывода также видно, что устройство имеет вполне определенный идентификатор производителя **VID = 0x0079** и идентификатор продукта **PID = 0x0006**. Сделав запрос в любой поисковик по сети Интернет, мы обнаружим что производителем данного устройства является «**Shenzhen Longshengwei Technology, Co., Ltd.**», а само устройство именуется как «**PC TWIN SHOCK Gamepad**».

Также из вывода нашей программы мы увидим, что данное устройство просит опрашивать его не реже чем с интервалом **10 мс**, а максимально потребляемый ток — **500** мА. Видимо разработчики поленились измерить и указать реальное значение потребляемого тока, и просто оставили максимально допустимое.

Еще одно важное значение из этого вывода содержит параметр **EPAddress = 0x81**. Здесь младшие 4 бита задают номер конечной точки которую следует опрашивать и она равна **1**, а старший 7-бит когда установлен в 1 указывает на то, что опрос должен производиться методом «Interrup Transfer».

Суммируя проанализированную информацию можно сказать, для получения данных с устройства нам требуется делать запросы «Interrupt IN» каждые 10 мс, но никто не запрещает делать их реже. :-)

Теперь, после того как мы подключили джойстик, программа непрерывно выдает в отладочный порт сообщение содержащее строку вида: **data received: 7F 7F 00 80 80 0F 00 00**. Эти восемь шестнадцатеричных значений и есть данные получаемые от джойстика отображают текущее состояние элементов управления. Попробуем разобраться какой байт здесь за что отвечает.

Покрутив немного левый «стик» в разные стороны мы увидим примерно следующие данные:

Листинг 11.4. Вывод программы karnix_usb10_test при подключении устройства типа «gamepad» (продолжение).

```
USB1 (1:1) data received: 80 00 00 80 80 0F 00 00 , class = 3/0/0, RX_STATUS: 0x89AF4B58, RX_STATUS2: 0x000089AF, STATUS: 0x04000001
USB1 (1:1) data received: 5C 00 00 80 80 0F 00 00 , class = 3/0/0, RX_STATUS: 0x80A2C358, RX_STATUS2: 0x000080A2, STATUS: 0x04000001
USB1 (1:1) data received: 00 2E 00 80 80 0F 00 00 , class = 3/0/0, RX_STATUS: 0xEB694B58, RX_STATUS2: 0x0000EB69, STATUS: 0x04000001
USB1 (1:1) data received: 00 80 00 80 80 0F 00 00 , class = 3/0/0, RX_STATUS: 0x2126C358, RX_STATUS2: 0x00002126, STATUS: 0x04000001
USB1 (1:1) data received: 00 EB 00 80 80 0F 00 00 , class = 3/0/0, RX_STATUS: 0xE7FC4B58, RX_STATUS2: 0x0000E7FC, STATUS: 0x04000001
USB1 (1:1) data received: 80 FF 00 80 80 0F 00 00 , class = 3/0/0, RX_STATUS: 0x86A0C358, RX_STATUS2: 0x000086A0, STATUS: 0x04000001
USB1 (1:1) data received: FF 80 00 80 80 0F 00 00 , class = 3/0/0, RX_STATUS: 0x25694B58, RX_STATUS2: 0x00002569, STATUS: 0x04000001
USB1 (1:1) data received: FF 55 00 80 80 0F 00 00 , class = 3/0/0, RX_STATUS: 0xE8EDC358, RX_STATUS2: 0x0000E8ED, STATUS: 0x04000001
USB1 (1:1) data received: DD 00 00 80 80 0F 00 00 , class = 3/0/0, RX_STATUS: 0x2C6B4B58, RX_STATUS2: 0x00002C6B, STATUS: 0x04000001
```

А понажимав различные кнопки увидим следующие сообщения:

```
USB1 (1:1) data received: 7F 7F 00 80 80 CF 0B 00 , class = 3/0/0, RX_STATUS: 0x4669C358, RX_STATUS2: 0x00004669, STATUS: 0x04000001
USB1 (1:1) data received: 7F 7F 00 80 80 CF 0E 00 , class = 3/0/0, RX_STATUS: 0x166A4B58, RX_STATUS2: 0x0000166A, STATUS: 0x04000001
USB1 (1:1) data received: 7F 7F 00 80 80 CF 0E 00 , class = 3/0/0, RX_STATUS: 0x166AC358, RX_STATUS2: 0x0000166A, STATUS: 0x04000001
USB1 (1:1) data received: 7F 7F 00 80 80 CF 0E 00 , class = 3/0/0, RX_STATUS: 0x166A4B58, RX_STATUS2: 0x0000166A, STATUS: 0x04000001
USB1 (1:1) data received: 7F 7F 00 80 80 CF 0D 00 , class = 3/0/0, RX_STATUS: 0xE66AC358, RX_STATUS2: 0x0000E66A, STATUS: 0x04000001
USB1 (1:1) data received: 7F 7F 00 80 80 CF 09 00 , class = 3/0/0, RX_STATUS: 0x26684B58, RX_STATUS2: 0x00002668, STATUS: 0x04000001
```

Видно, что в первом случае изменения касаются нулевого и первого байта. Во втором — пятого и шестого. Не требуется большой фантазии чтобы догадаться, нулевой байт отвечает за координату «X» левого «стика», первый байт — за координату «Y» левого «стика», а шестой байт является битовым полем для отображения состояния кнопок.

С правым «стиком» несколько сложнее. В данном устройстве «стики» могут работать в аналоговом и дискретном режимах. В аналоговом режиме положение «X» и «Y» правого «стика» отображаются в третьем и четвертом байте. В дискретном — положение правого «стика» совпадает с кнопками «1», «2», «3» и «4» в пятом байте.

Немного поэкспериментировав получим следующую таблицу соответствия между элементами управления и номером элемента в массиве получаемых от джойстика данных:

Таблица 16.1. Соответствие элементов управления в массиве данных устройства типа «gamepad».

Режим	Байт 0	Байт 1	Байт 2	Байт 3	Байт 4	Байт 5	Байт 6	Байт 7
Аналоговый	Ось «X» левый стик	Ось «Y» левый стик	0x00	Ось «X» правый стик	Ось «Y» правый стик	Цветные кнопки «1», «2», «3», «4»	Черные кнопки «1», «2», «Start», «Select»	0x00
Дискретный	Ось «X» левый стик	Ось «Y» левый стик	0x00	0x80	0x80	Правый стик и Цветные кнопки «1», «2», «3», «4»	Черные кнопки «1», «2», «Start», «Select»	0x00

Чтож, теперь у нас есть джойстик и мы можем задействовать его в разработке какой нибудь игры или для пульта управления квадрокоптером ;-).

Далее мы «пропатчим» демонстрационную версию игры «TetRISCV» в составе KarnixSoC на использование USB геймпада.

7.3.2. Тестирование устройства типа «mouse»

Разобравшись с джойстиком, посмотрим как ведет себя указующее устройство типа «мышь» («mouse pointing device»). Подключив в порт USB платы расширения проводную «мышь» с поддержкой USB 1.0 и нажав кнопку «RESET» мы получим следующие сообщения в терминале отладочного порта:

Листинг 12.1. Вывод программы karnix_usb10_test при подключении устройства типа «mouse».

```
usb10_init: Device detected: VID/PID = 0x046D/0xC077, class/subclass = 0x00/0x00, bcdUSB = 0x0200
usb10_init: Address = 1, Config wTotal = 34, Interface Class/Subclass/Proto = 3/1/2, EPAddress =
0x81, Interval = 10 ms, MaxPacket = 4
USB1: new device addr = 1, VID/PID = 0x046D/0xC077, class/subclass/proto = 3/1/2, EndpointAddress =
1, Interval = 10, MaxPower = 100 mA
```

Видим что перед нами USB **2.0** Class HID устройство, VID/PID которого равен **0x046D/0xC077**. Вводим эти данные в поисковый сервис в сети Интернет и на выясняем, что перед нами «**USB Optical Mouse**» производства «**Logitech, Inc.**».

Данное устройство, также как «геймпад», сообщет что его требуется опрашивать с интервалом **10 мс** по номеру конечной точки **1** используя тот же «Interrupt Transfer» метод (запрос типа «Interrupt IN»). Но, в отличие от «геймада», устройство типа «мышь» будет выдавать пакет данных максимальной длины **MaxPacket = 4** байта!

Теперь, если начать перемещать «мышь» по столу, то в терминале будут появляться следующие сообщения:

Листинг 12.2. Вывод программы karnix_usb10_test при перемещении «мыши» по столу.

```
USB1 (1:1) data received: 00 EE 18 00 , class = 3/1/2, RX_STATUS: 0xEE95C338, RX_STATUS2: 0x0000EE95, STATUS: 0x04000001
USB1 (1:1) data received: 00 04 1C 00 , class = 3/1/2, RX_STATUS: 0xDAB64B38, RX_STATUS2: 0x0000DAB6, STATUS: 0x04000001
USB1 (1:1) data received: 00 1F 11 00 , class = 3/1/2, RX_STATUS: 0x4DC2C338, RX_STATUS2: 0x00004DC2, STATUS: 0x04000001
USB1 (1:1) data received: 00 36 F7 00 , class = 3/1/2, RX_STATUS: 0xE5594B38, RX_STATUS2: 0x0000E559, STATUS: 0x04000001
```

Если нажимать и отпускать кнопки «мыши», при этом не перемещая устройство, получим следующие сообщения:

Листинг 12.3. Вывод программы karnix_usb10_test при нажимании клавиш на «мыше».

```
USB1 (1:1) data received: 01 00 00 00 , class = 3/1/2, RX_STATUS: 0x27FE4B38, RX_STATUS2: 0x000027FE, STATUS: 0x04000001
USB1 (1:1) data received: 02 00 00 00 , class = 3/1/2, RX_STATUS: 0x63FEC338, RX_STATUS2: 0x000063FE, STATUS: 0x04000001
USB1 (1:1) data received: 03 00 00 00 , class = 3/1/2, RX_STATUS: 0x9FFF4B38, RX_STATUS2: 0x00009FFF, STATUS: 0x04000001
USB1 (1:1) data received: 02 00 00 00 , class = 3/1/2, RX_STATUS: 0x63FEC338, RX_STATUS2: 0x000063FE, STATUS: 0x04000001
USB1 (1:1) data received: 03 00 00 00 , class = 3/1/2, RX_STATUS: 0x9FFF4B38, RX_STATUS2: 0x00009FFF, STATUS: 0x04000001
```

Ну и если покрутить «колесо», то получим:

Листинг 12.4. Вывод программы karnix_usb10_test вращении ролика быстрой прокрутки.

```
USB1 (1:1) data received: 00 00 00 FE , class = 3/1/2, RX_STATUS: 0x5B7E4B38, RX_STATUS2: 0x00005B7E, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 00 FE , class = 3/1/2, RX_STATUS: 0x5B7EC338, RX_STATUS2: 0x00005B7E, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 00 FF , class = 3/1/2, RX_STATUS: 0x9BBF4B38, RX_STATUS2: 0x00009BBF, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 00 FF , class = 3/1/2, RX_STATUS: 0x9BBFC338, RX_STATUS2: 0x00009BBF, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 00 FE , class = 3/1/2, RX_STATUS: 0x5B7E4B38, RX_STATUS2: 0x00005B7E, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 00 01 , class = 3/1/2, RX_STATUS: 0x1B3EC338, RX_STATUS2: 0x00001B3E, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 00 01 , class = 3/1/2, RX_STATUS: 0x1B3E4B38, RX_STATUS2: 0x00001B3E, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 00 01 , class = 3/1/2, RX_STATUS: 0x1B3EC338, RX_STATUS2: 0x00001B3E, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 00 01 , class = 3/1/2, RX_STATUS: 0x1B3E4B38, RX_STATUS2: 0x00001B3E, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 00 FF , class = 3/1/2, RX_STATUS: 0x9BBFC338, RX_STATUS2: 0x00009BBF, STATUS: 0x04000001
```

Интерпретировать формат получаемых данных не представляет никакого труда. Относительное смещение устройства по осям «X» и «Y» передается в байте 1 и 2. Байт 0 содержит битовое поле отображающее состояние нажатия кнопок, а байт 3 — смещение ролика быстрой прокрутки.

Устройство работает следующим образом. Если «мышь» находится в состоянии покоя, то она не выдает никаких данных (высылает NAK в ответ на запрос типа «Interrupt IN»). Если начать перемещать её относительно стола, то данные будут готовы к выдаче с частотой 100 Гц (интервал 10 мс). Передаваемые значения задают относительное смещение устройства, в диапазоне от -128 до 127, физически пройденное устройством за этот интервал времени.

Аналогичная ситуация с роликом быстрой прокрутки — передаваемое значение «прокрутки» ролика находится в диапазоне от -128 до 127. Значение 0 показывает, что по данной координате изменений не зафиксировано. Собственно это все, что требуется знать программисту про устройство типа «мышь».

7.3.3. Тестирование устройства типа «keyboard»

Так мы постепенно добрались до третьего типа устройств - клавиатуры. Устройства типа «keyboard» (или «keypad») выдают данные немного иначе — в виде стека, под который отводятся байты с номерами от 2 до 7. При нажимании клавиши (если это не клавиша модификатор), её Usage ID код помещается в стек из этих шести элементов и остается там пока данная клавиша удерживается в нажатом положении. Нажимание еще одной клавиши одновременно с уже удерживаемыми приводит к добавлению еще одного значения в стек. Если какая либо из клавиш отжимается, то её код убирается из стека, а все остальные значения в нём сдвигаются влево. Нулевое значение в стеке означает отсутствие какого либо действия. Нажатие клавиш-модификаторов приводит к изменению значений битов в битовом поле — байт 0.

Не все комбинации клавиш могу быть декодированным внутренней аппаратурой клавиатуры из-за устройства её матрицы. Если такое состояние обнаруживается, то клавиатура выдает массив из шести значений **0x01** вместо данных стека нажатий.

Листинг 13.1. Вывод программы karnix_usb10_test при подключении устройства типа «keyboard».

```
usb10_init: Device detected: VID/PID = 0x046D/0xC31C, class/subclass = 0x00/0x00, bcdUSB = 0x0110
usb10_init: Address = 1, Config wTotal = 59, Interface Class/Subclass/Proto = 3/1/1, EPAddress =
0x81, Interval = 10 ms, MaxPacket = 8
USB1: new device addr = 1, VID/PID = 0x046D/0xC31C, class/subclass/proto = 3/1/1, EndpointAddress =
1, Interval = 10, MaxPower = 90 mA
```

Перед нами USB **1.1** Class HID устройство с VID/PID равным **0x046D/0xC31C**. Поиск в Сети показывает, что это некая «**USB Keyboard**» производства «**Logitech, Inc.**». Минимальный заявленный интервал опроса — **10 мс**. Метод опроса всё тот же «Interrupt IN» на конечную точку с номером **1**. Максимальный размер блока данных — **8 байт**.

Наша тестовая программа продолжает выдавать в терминал с небольшой частотой сообщения следующего вида:

```
USB1 (1:1) data received: 00 00 00 00 00 00 00 00 , class = 3/1/1, RX_STATUS: 0xF4BF4B58, RX_STATUS2: 0x0000F4BF, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 00 00 00 00 00 00 , class = 3/1/1, RX_STATUS: 0xF4BFC358, RX_STATUS2: 0x0000F4BF, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 00 00 00 00 00 00 , class = 3/1/1, RX_STATUS: 0xF4BF4B58, RX_STATUS2: 0x0000F4BF, STATUS: 0x04000001
```

Последовательно зажмем и будем удерживать клавиши: «A», «B» и «C». В терминале получим следующие сообщения:

```
USB1 (1:1) data received: 00 00 04 00 00 00 00 00 , class = 3/1/1, RX_STATUS: 0x70BE4B58, RX_STATUS2: 0x000070BE, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 04 05 00 00 00 00 , class = 3/1/1, RX_STATUS: 0x70724B58, RX_STATUS2: 0x00007072, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 04 05 00 00 00 00 , class = 3/1/1, RX_STATUS: 0x7072C358, RX_STATUS2: 0x00007072, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 04 05 00 00 00 00 , class = 3/1/1, RX_STATUS: 0x70724B58, RX_STATUS2: 0x00007072, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 04 05 00 00 00 00 , class = 3/1/1, RX_STATUS: 0x7072C358, RX_STATUS2: 0x00007072, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 04 05 06 00 00 00 , class = 3/1/1, RX_STATUS: 0xF8724B58, RX_STATUS2: 0x0000F872, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 04 05 06 00 00 00 , class = 3/1/1, RX_STATUS: 0xF872C358, RX_STATUS2: 0x0000F872, STATUS: 0x04000001
```

Видно, что в стек последовательно добавляются UsageID коды 0x04 (клавиша «A»), 0x05 (клавиша «B») и 0x06 (клавиша «C»).

Теперь отпустим сначала клавишу «A», потом отпустим клавишу «C», а потом клавишу «B»:

```
USB1 (1:1) data received: 00 00 05 06 00 00 00 00 , class = 3/1/1, RX_STATUS: 0xA1374B58, RX_STATUS2: 0x0000A137, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 05 06 00 00 00 00 , class = 3/1/1, RX_STATUS: 0xA137C358, RX_STATUS2: 0x0000A137, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 05 06 00 00 00 00 , class = 3/1/1, RX_STATUS: 0xA1374B58, RX_STATUS2: 0x0000A137, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 05 00 00 00 00 00 , class = 3/1/1, RX_STATUS: 0xA1BFC358, RX_STATUS2: 0x0000A1BF, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 05 00 00 00 00 00 , class = 3/1/1, RX_STATUS: 0xA1BF4B58, RX_STATUS2: 0x0000A1BF, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 00 00 00 00 00 00 , class = 3/1/1, RX_STATUS: 0xF4BFC358, RX_STATUS2: 0x0000F4BF, STATUS: 0x04000001
```

Видно, что сначала из стека был удален код 0x04 и все данные в стеке сдвинуты влево на одну позицию. Далее из стека удаляются коды 0x06 и код 0x05. Теперь стек пуст — в нем одни нули.

Нажмем одновременно пять рядом расположенных клавиш:

```
USB1 (1:1) data received: 00 00 01 01 01 01 01 01 , class = 3/1/1, RX_STATUS: 0x89134B58, RX_STATUS2: 0x00008913, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 01 01 01 01 01 01 , class = 3/1/1, RX_STATUS: 0x8913C358, RX_STATUS2: 0x00008913, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 01 01 01 01 01 01 , class = 3/1/1, RX_STATUS: 0x89134B58, RX_STATUS2: 0x00008913, STATUS: 0x04000001
USB1 (1:1) data received: 00 00 01 01 01 01 01 01 , class = 3/1/1, RX_STATUS: 0x8913C358, RX_STATUS2: 0x00008913, STATUS: 0x04000001
```

и наблюдаем, как весь стек занят Usage ID кодом ошибки 0x01. Теперь отпустим все клавиши чтобы сбросить «аварийное» состояние:

```
USB1 (1:1) data received: 00 00 00 00 00 00 00 00 , class = 3/1/1, RX_STATUS: 0xF4BFC358, RX_STATUS2: 0x0000F4BF, STATUS: 0x04000001
```

Очевидно, что для правильного декодирования последовательности нажатий клавиш потребуется реализовать программную машину состояний. Сделать это несложно, гораздо сложнее преобразовать Usage ID коды (и их сочетания с клавишами-модификаторами) в последовательность ASCII символов чтобы организовать терминальный ввод. Но это тема отдельного разговора и я надеюсь, мне удастся осветить её в моих дальнейших публикациях. Сейчас же нам вполне достаточно имеющихся данных чтобы организовать несложное управления в играх или для аппаратуры АСУ ТП. Так попробуем пропатчить «TetRISCV».

7.4. Адаптируем игру «TetRISC-V» к USB устройствам ввода

В репозитории «KarnixSoC» имеется простейшая реализация игры «Тетрис» расположенная в каталоге **./src/main/c/karnix/karnix_tetriscv/**. Игра была добавлена мной при реализации CGA видеоадаптера в качестве демонстрации применения спец.эффектов — комбинации текстового и графического режимов. Для управления фигурами (они называются «тетромино») в этой игре используются четыре аппаратных кнопки расположенных на плате «Карно». Это очень неудобный способ. Настало время исправить этот недочет и добавить поддержку управления с помощью клавиатуры, джойстика и «манипулятора типа «мышь».

Начнем с того, что добавим некоторые изменения в файл сборочного скрипта **Makefile**. Во-первых, в список макроопределений задаваемых переменной окружения **DEFS** добавим еще одну константу препроцессора с именем **USB10_ENABLE**, для этого в заголовке файла добавим следующую строку:

```
DEFS += -DUSB10_ENABLE
```

Аналогичным образом в переменную окружения **SRCS** добавим имя файла с кодом драйвера USB хост-контроллера для того, чтобы функции драйвера подключались при сборке:

```
SRCS += $(SOCDIR)/usb10.c
```

Теперь создадим новый заголовочный файл **src/usb_hid_keys.h** для констант с кодами Usage ID для клавиатуры. Добавим в него четыре кода для клавиш управления курсором:

```
#define KEY_RIGHT 0x4f // Keyboard Right Arrow
#define KEY_LEFT 0x50 // Keyboard Left Arrow
#define KEY_DOWN 0x51 // Keyboard Down Arrow
#define KEY_UP 0x52   // Keyboard Up Arrow
```

Теперь начнем модифицировать исходного кода программы **karnix_tetriscv** находящийся в файле **src/main.c**. Используя условную компиляцию основанную на определении константы **USB10_ENABLE**, подключим заголовочные файлы драйвера USB и файл с кодами клавиатуры, а также введем несколько глобальных переменных, добавив в заголовке файл следующий код:

```
#if(USB10_ENABLE)
#include "usb10.h"
#include "usb_hid_keys.h"
uint32_t reg_usb_timestamp = 0;         // Timestamp of last USB transfer
uint32_t reg_usb_error_count = 0;       // Number of errors on USB bus
uint32_t reg_usb_response_size = 0;     // Recall max transfer size in bytes
uint32_t reg_usb_device_type = 0;       // Recall device type: 1 - keyboard, 2 - mouse, 3 -
gamepad, 0 - unknown
#endif
```

Назначение этих глобальный переменных следующее:

- **reg_usb_timestamp** — содержит временную отметку момента когда последний раз происходито обращение к USB устройству.
- **reg_usb_error_count** — будет содержать счетчик числа накопленных ошибок возникших на шине USB. Если из количество превысит некоторое значение, то будем сбрасывать USB шину и инициализировать устройство повторно.
- **reg_usb_response_size** — будет содержать размер блока данных получаемого от устройства. Значение этой переменной будет вычисляться в зависимости от типа подключенного устройства.
- **reg_usb_response_size** — будет содержать код типа устройства: 1 — клавиатура, 2 — «мышь», 3 — «геймпад» или 0 — неизвестное HID устройство.

Далее, в теле функции **main()**, после инициализации CGA видеоадаптера, вставим код инициализации USB хост-контроллера:

```
        // Enable USB1
        #if(USB10_ENABLE)
        USB1->CONTROL &= ~USB10_CONTROL_ENABLE_BIT;
        delay_us(1000);
         USB1->CONTROL |= USB10_CONTROL_RESET_DELAY_SET(1500000 / 1000 * 10); // Set reset duration
to 11ms (num of ticks at 1.5 MHz
        USB1->CONTROL |= USB10_CONTROL_KEEPALIVE_BIT;
        USB1->CONTROL |= USB10_CONTROL_ENABLE_BIT;
        printf("USB1 enabled\r\n");
        #endif
```

Этот код полностью повторяет тот, что мы уже имеем в тестовой программе **karnix_usb10_test**.

Теперь в начало главного цикла, сразу после кода отображения накопленной статистики, добавим кусок кода взаимодействия с USB хост-контроллером. По своему содержанию он будет очень близок к тому, что мы делали в тестовом приложении:

Листинг 14.1. Поддержка USB HID устройств в игре «TetRISC-V»: детектирование устройства.

```
                #if(USB10_ENABLE)
                if(timestamp - reg_usb_timestamp >= 100000) { // Send USB command every 100ms

                        reg_usb_timestamp = timestamp;

                        if(usb10_device_address == 0) {

                                uint8_t new_device_address = 0;

                                if(usb10_init(USB1, &new_device_address, NULL, NULL, NULL, NULL) ==
0) {

                                        reg_usb_error_count = 0;

                                        xprintf("\rUSB1: new device addr = %d, VID/PID =
0x%04X/0x%04X, "
                                                "class/subclass/proto = %d/%d/%d, "
                                                "EndpointAddress = %d, Interval = %d, MaxPower = %d
mA\r\n",
                                                new_device_address,
                                                usb10_device_descr.idVendor,
                                                usb10_device_descr.idProduct,
                                                usb10_interface_descr.bInterfaceClass,
                                                usb10_interface_descr.bInterfaceSubclass,
                                                usb10_interface_descr.bInterfaceProtocol,
                                                usb10_endpoint_descr.bEndpointAddress & 0x0f,
                                                usb10_endpoint_descr.bInterval,
                                                usb10_config_descr.bMaxPower * 2
                                        );

                                        // Try to guess response data packet size using class info
                                       if(usb10_interface_descr.bInterfaceClass == 3 &&
                                                  usb10_interface_descr.bInterfaceSubclass == 1 &&
                                                  usb10_interface_descr.bInterfaceProtocol == 1) {

                                                // We have to check RX packet len (88 bits) to skip
empty packets

                                                reg_usb_response_size = 8; // HID keyboard
                                                reg_usb_device_type = 1;

                                        } else if(usb10_interface_descr.bInterfaceClass == 3 &&
                                           usb10_interface_descr.bInterfaceSubclass == 1 &&
                                           usb10_interface_descr.bInterfaceProtocol == 2) {

                                                reg_usb_response_size = 4; // HID mouse
                                                reg_usb_device_type = 2;

                                        } else if(usb10_interface_descr.bInterfaceClass == 3 &&
                                                  usb10_interface_descr.bInterfaceSubclass == 0 &&
                                                  usb10_interface_descr.bInterfaceProtocol == 0) {

                                                reg_usb_response_size = 8; // HID gamepad
                                                reg_usb_device_type = 3;

                                        } else {
                                                reg_usb_response_size = 8; // Unknown
                                                reg_usb_device_type = 0;
                                        }

                                }
```

```
                }
```

Как и в тестовом приложении **karnix_usb10_test**, приведенный выше фрагмент кода производит регулярную (10 раз в секунду) проверку состояния шины USB и инициализирует вновь подключенное устройство вызовом функции **usb10_init()** драйвера. Если новое устройство готово к работе, то вычисляет тип устройства и размера пакета данных, сохраняет их в глобальных переменных **reg_usb_device_type** и **reg_usb_device_type** для того, чтобы воспользоваться этими данными в следующем фрагменте кода:

Листинг 14.2. Поддержка USB HID устройств ввода в игре «TetRISC-V»: опрос устойства.

```
                else {
                                uint8_t endpoint = 1; //should be
usb10_config_resp.conf.endp.bEndpointAddress & 0x0f ?
                                uint8_t response_data[8] = {0};

                                // Poll Endpoint for new data
                                int ret = usb10_in_request(USB1, usb10_device_address, endpoint,
                                                response_data, reg_usb_response_size);

                                if(ret >= 0) {

                                        reg_usb_error_count = 0;

                                } else if(ret == USB10_IN_ENAK || ret == USB10_IN_ETIMEOUT) { // NAK
or timeout (dupe)
                                        // No new data, do nothing
                                        goto usb_end;
                                } else {
                                        // Three errors is enough to disable USB
                                        if(++reg_usb_error_count > 3) {
                                                xprintf("\rUSB1 (%d:%d) failed, ret = %d\r\n",
usb10_device_address, endpoint, ret);
                                                usb10_device_address = 0; // flag USB as broken
                                                goto usb_end;
                                        }
                                }
```

Приведенный выше фрагмент кода выполняет опрос USB HID устройства методом «Interrupt IN» транзакции используя функцию драйвера **usb10_in_request()**. В случае если в процессе исполнения транзакции возникла ошибка, производится приращение счетчика ошибок и, если он достиг значения более 3-х, устройство помечается как «нерабочее» присваиванием **usb10_device_address = 0**. Если в процессе транзакции возник таймаут или получен код ошибки ENAK (признак того, что у устройства сейчас нет данных), то процесс завершается переходом к метке **usb_end** без каких-либо дальнейших действий. Иначе, в случае успеха, процесс переходит к обработке полученных данных представленной следующим фрагментом кода:

Листинг 14.3. Поддержка USB HID устройств в игре «TetRISC-V»: трансляция состояний элементов управления во внутренний код.

```
                                // Process USB HID data

                                switch(reg_usb_device_type) {
                                        case 1: // keyboard
                                                switch(response_data[2]) {
                                                        case KEY_UP:    keys = last_keys =
GPIO_IN_KEY1;
                                                                        break;
                                                        case KEY_DOWN:  keys = last_keys =
GPIO_IN_KEY2;
                                                                        break;
                                                        case KEY_LEFT:  keys = last_keys =
GPIO_IN_KEY3;
                                                                        break;
```

```
                                                        case KEY_RIGHT: keys = last_keys =
GPIO_IN_KEY0;
                                                                break;
                                                }
                                                break;

                                        case 2: // mouse
                                                if(response_data[1] >= 0x90) // left
                                                        keys = last_keys = GPIO_IN_KEY3;
                                                if(response_data[1] > 0x10 && response_data[1] <
0x80) // right
                                                        keys = last_keys = GPIO_IN_KEY0;
                                                if(response_data[0] & 0x01) // Up
                                                        keys = last_keys = GPIO_IN_KEY1;
                                                if(response_data[0] & 0x02) // down
                                                        keys = last_keys = GPIO_IN_KEY2;
                                                break;

                                        case 3: // gamepag
                                                if(response_data[0] == 0x00 || response_data[5] ==
0x8f) // left
                                                        keys = last_keys = GPIO_IN_KEY3;
                                                if(response_data[0] == 0xff || response_data[5] ==
0x2f) // right
                                                        keys = last_keys = GPIO_IN_KEY0;
                                                if(response_data[1] == 0x00 || response_data[5] ==
0x1f) // up
                                                        keys = last_keys = GPIO_IN_KEY1;
                                                if(response_data[1] == 0xff || response_data[5] ==
0x4f) // down
                                                        keys = last_keys = GPIO_IN_KEY2;
                                                break;
                                }

                                usb_end: ;
                        }

                }
                #endif
```

Как видено, процесс обработки полученных данных состоит в том, чтобы в зависимости от типа устройства хранящегося в переменной **reg_usb_device_type** (где 1 — клавиатура, 2 — «мышь», 3 — «геймпад»), транслировать состояния элементов управления или нажатых клавиш во внутренний код (**GPIO_IN_KEY**x) сохраняемый в переменной **keys** для перемещения тетрамино.

На этом модификация кода игры «TetRISC-V» заканчиваются и можно приступать к тестированию. Для этого запустим сборку командой **make**:

Листинг 14.3. Процесс сборки игры «TetRISC-V».

```
rz@devbox:~/KarnixSOC/src/main/c/karnix/karnix_tetriscv$ make
make ram
make[1]: Entering directory '/home/rz/KarnixSOC/src/main/c/karnix/karnix_tetriscv'
mkdir -p build
echo 1 > build/build_number
*** Building for RAM ***
make MODEL=ram build_sources build/karnix_tetriscv.hexx
make[2]: Entering directory '/home/rz/KarnixSOC/src/main/c/karnix/karnix_tetriscv'
mkdir -p build/src/
/opt/xpack-riscv-none-elf-gcc-14.2.0-2/bin/riscv-none-elf-gcc -c -march=rv32imafc
-mabi=ilp32f -DNDEBUG -flto -fno-common -MD -fstrict-volatile-bitfields -fno-strict-aliasing
-fno-builtin-printf -DBUILD_NUMBER=`cat build/build_number`+1 -DUSB10_ENABLE  -Os  -I./src
-I -I/include -I../karnix_soc/src -I../karnix_soc/src/include -o build/src/tetris.o
src/tetris.c
/opt/xpack-riscv-none-elf-gcc-14.2.0-2/bin/riscv-none-elf-gcc -S -march=rv32imafc
-mabi=ilp32f -DNDEBUG -flto -fno-common -MD -fstrict-volatile-bitfields -fno-strict-aliasing
-fno-builtin-printf -DBUILD_NUMBER=`cat build/build_number`+1 -DUSB10_ENABLE  -Os  -I./src
```

```
-I -I/include -I../karnix_soc/src -I../karnix_soc/src/include -o build/src/tetris.o.disasm
src/tetris.c
…
/opt/xpack-riscv-none-elf-gcc-14.2.0-2/bin/riscv-none-elf-gcc -c -march=rv32imafc
-mabi=ilp32f -DNDEBUG -flto -fno-common -MD -fstrict-volatile-bitfields -fno-strict-aliasing
-fno-builtin-printf -DBUILD_NUMBER=`cat build/build_number`+1 -DUSB10_ENABLE  -Os -o
build/__/karnix_soc/src/crt_ram.o ../karnix_soc/src/crt_ram.S -D__ASSEMBLY__=1
...

Memory region         Used Size  Region Size  %age Used
             RAM:       37760 B        72 KB     51.22%
/opt/xpack-riscv-none-elf-gcc-14.2.0-2/bin/riscv-none-elf-objcopy -O binary
build/karnix_tetriscv_ram.elf build/karnix_tetriscv_ram.bin
…

make xip
make[1]: Entering directory '/home/rz/KarnixSOC/src/main/c/karnix/karnix_tetriscv'
*** Building for XIP ***
make MODEL=xip build_sources
make[2]: Entering directory '/home/rz/KarnixSOC/src/main/c/karnix/karnix_tetriscv'
mkdir -p build/__/karnix_soc/src/
/opt/xpack-riscv-none-elf-gcc-14.2.0-2/bin/riscv-none-elf-gcc -c -march=rv32imafc
-mabi=ilp32f -DNDEBUG -flto -fno-common -MD -fstrict-volatile-bitfields -fno-strict-aliasing
-fno-builtin-printf -DBUILD_NUMBER=`cat build/build_number`+1 -DUSB10_ENABLE  -Os -o
build/__/karnix_soc/src/crt_xip.o ../karnix_soc/src/crt_xip.S -D__ASSEMBLY__=1

Memory region         Used Size  Region Size  %age Used
           FLASH:       34932 B        15 MB      0.22%
             RAM:        3664 B        60 KB      5.96%
/opt/xpack-riscv-none-elf-gcc-14.2.0-2/bin/riscv-none-elf-objcopy -O binary
build/karnix_tetriscv_xip.elf build/karnix_tetriscv_xip.bin
```

Аналогично, в подкаталоге **./build/** получаем два бинарных (исполняемых) файла для вариантов размещения в RAM и XiP:

```
rz@devbox:~/KarnixSOC/src/main/c/karnix/karnix_tetriscv$ ll build/*.bin
-rwxrwxr-x 1 rz rz 34900 Oct 17 23:48 build/karnix_tetriscv_ram.bin*
-rwxrwxr-x 1 rz rz 34932 Oct 17 23:48 build/karnix_tetriscv_xip.bin*
```

Воспользуемся вариантом для XiP, запишем его в NOR Flash память утилитой **openFPGALoader** в область отведенную для приложений:

```
$ openFPGALoader -f -o 0xA0100000 karnix_tetriscv_xip.bin

empty
write to flash
No cable or board specified: using direct ft2232 interface
Jtag frequency : requested 6.00MHz   -> real 6.00MHz
Open file DONE
Parse file DONE
Enable configuration: DONE
SRAM erase: DONE
Detail:
Jedec ID          : ef
memory type       : 70
memory capacity   : 18
flash chip unknown: use basic protection detection
start addr: a0100000, end_addr: a0110000
Erasing: [==================================================] 100.00%
Done
Writing: [==================================================] 100.00%
Done
Refresh: DONE
```

Подключим терминал к отладочному порту и после выполнения аппаратного сброса (нажатия кнопки «RESET») наблюдаем в нем следующие сообщения:

```
TetRISC-V for Karnix SoC. Build 00002 on Oct 17 2025 at 23:48:31
Copyright (C) 2024-2025 Fabmicro, LLC., Tyumen, Russia.

=== Configuring ===

Press '*' to reset config.....
eeprom_probe(0x50) done
config_load() CRC16 mismatch: 0xB001 != 0xFFFF
Defaults loaded by EEPROM CRC ERROR!
=== Hardware init ===
Filling SRAM at: 0x90000000, size: 524288 bytes...
Checking SRAM at: 0x90000000, size: 524288 bytes...
Enabling SRAM...
SRAM at 0x90000000 is enabled!
CGA init done
USB1 enabled
UART0 init done
TIMER0 init done
TIMER1 init done
audiodac_init: divider = 234
audiodac0_isr: tx ring buffer underrun!
audiodac0_start_playback: done, fifo depth is 1024 samples
AUDIODAC0 init done
Video double-buffers allocated
=== Hardware init done ===
audiodac0_isr: tx ring buffer underrun!
...
audiodac0_isr: tx ring buffer underrun!
audiodac0 samples available: 0 -> 2047
```

Подключим «геймпад» в порт USB макетной платы и наблюдаем следующие сообщения:

```
usb10_init: Device detected: VID/PID = 0x0079/0x0006, class/subclass = 0x00/0x00, bcdUSB =
0x0100
usb10_init: Address = 1, Config wTotal = 41, Interface Class/Subclass/Proto = 3/0/0,
EPAddress = 0x81, Interval = 10 ms, MaxPacket = 8
USB1: new device addr = 1, VID/PID = 0x0079/0x0006, class/subclass/proto = 3/0/0,
EndpointAddress = 1, Interval = 10, MaxPower = 500 mA
```

Нажимая клавиши или «стики» в разные стороны наблюдаем на экране перемещения фигуры, при этом в терминал выдаются следующие отладочные сообщения:

```
Inputs: last_keys = 0001, new_keys = 0000
Inputs: last_keys = 0008, new_keys = 0000
Inputs: last_keys = 0001, new_keys = 0000
Inputs: last_keys = 0001, new_keys = 0000
Inputs: last_keys = 0008, new_keys = 0000
Inputs: last_keys = 0008, new_keys = 0000
Inputs: last_keys = 0008, new_keys = 0000
Inputs: last_keys = 0001, new_keys = 0000
Inputs: last_keys = 0001, new_keys = 0000
Inputs: last_keys = 0002, new_keys = 0000
Inputs: last_keys = 0001, new_keys = 0000
Inputs: last_keys = 0004, new_keys = 0000
Inputs: last_keys = 0002, new_keys = 0000
Inputs: last_keys = 0002, new_keys = 0000
Inputs: last_keys = 0002, new_keys = 0000
…
```

Отключив «геймпад» и подключив в место него клавиатуру увидим в терминале знакомые сообщения:

```
USB1 (1:1) failed, ret = -6
usb10_init: Device detected: VID/PID = 0x046D/0xC31C, class/subclass = 0x00/0x00, bcdUSB =
0x0110
usb10_init: Address = 1, Config wTotal = 59, Interface Class/Subclass/Proto = 3/1/1,
EPAddress = 0x81, Interval = 10 ms, MaxPacket = 8
```

```
USB1: new device addr = 1, VID/PID = 0x046D/0xC31C, class/subclass/proto = 3/1/1,
EndpointAddress = 1, Interval = 10, MaxPower = 90 mA
Inputs: last_keys = 0008, new_keys = 0000
Inputs: last_keys = 0001, new_keys = 0000
Inputs: last_keys = 0004, new_keys = 0000
Inputs: last_keys = 0008, new_keys = 0000
Inputs: last_keys = 0002, new_keys = 0000
```

То же самое с «мышью»:

```
USB1 (1:1) failed, ret = -6
usb10_init: Device detected: VID/PID = 0x046D/0xC077, class/subclass = 0x00/0x00, bcdUSB =
0x0200
usb10_init: Address = 1, Config wTotal = 34, Interface Class/Subclass/Proto = 3/1/2,
EPAddress = 0x81, Interval = 10 ms, MaxPacket = 4
USB1: new device addr = 1, VID/PID = 0x046D/0xC077, class/subclass/proto = 3/1/2,
EndpointAddress = 1, Interval = 10, MaxPower = 100 mA
Inputs: last_keys = 0008, new_keys = 0000
Inputs: last_keys = 0008, new_keys = 0000
Inputs: last_keys = 0001, new_keys = 0000
```

Перемещение «мыши» по столу «горизонтально» производит перемещение фигуры тетромино, нажатие левой кнопки «мыши» производит вращение фигуры, а правой — движение её вниз стакана.

Чтож, теперь, наконец-то можно сполна насладиться игрой в Тетрис используя любое из удобных устройств ввода!

8. Выводы из полученного опыта

Объем кода USB хост-контроллера вышел очень небольшим, менее 900 строк на SpinalHDL и около 1000 строк кода на языке Си для драйвера. Код хост-контроллера получился хорошо читаемым и легко обслуживаемым — его не сложно наращивать увеличивая функционал, что позволяет надеяться на продолжение и реализацию режима «Full Speed» (12 МГц) или даже USB 2.0 «Full Speed» без привлечения аппаратных блоков SERDES.

В процессе отладки мне ни разу не приходилось прибегать к использованию симуляции на Verilator-е. Все проблемы по большей части возникали из-за неполного или неправильного понимания протокола USB, что разрешалось записью дампов анализатором сигналов с последующим погружением в их анализ вперемешку со чтением спецификации. Мне пришлось изрисовать большое количество бумаги на ручное декодирование снятых осциллограмм.

В процессе написания статьи, когда я построил диаграммы переходов состояний для всех автоматов, то проанализировав их «с высоты птичьего полета» мне стало понятно, что реализация хост-контроллера выглядит очень неоптимальной и требует серьезной переработки. В частности, два вспомогательных автомата для формирования «Bus Reset» и «KeepAlive» оказались дублирующими функции друг-друга, их я сразу объединил в один **USBSendSE0** с входным параметром задающим число битовых интервалов.

Во-вторых, стали очевидны повторяющиеся паттерны, такие как формирование строба clock_strob и приращение bit_count, конвертирование бита в символ, расчет CRC16. Этот повторяющийся код был вынесен в отдельные процедуры базового класса, что сильно сократило объем текста и сделало его более понятным. Но в то же время проявился нежелательный эффект — в порождаемом Verilog коде появилось много дублирующей аппаратуры, ведь на каждый автомат приходится одна реализация класса со всеми его функциями и переменными (регистрами). Эта проблема легко решилась путем условной активации кода по параметрам передаваемым в конструктор базового класса (аналог параметризации в Verilog).

В общем, визуализация и повторное переосмысливание кода вызванное необходимостью изложения его в данной статье, положительно сказалось на структуре кода. В процессе рефакторинга даже были выявлены и исправлены несколько серьезных ошибок остававшихся незамеченными ранее. К сожалению, мои познания в языке Scala на котором основывается SpinalHDL почти что ничтожны. Я уверен, что опытный разработчик-скалист сразу обнаружит массу мест для еще больше оптимизации и сделает код еще более наглядным. Призываю программистов желающих погрузиться в цифровую аппаратура изучать Scala и SpinalHDL, и присоединяться к моему проекту KarnixSoC.

При написании драйвера я столкнулся с тем, что очень часто (но не всегда), драйвер не успевает отправить подтверждающий пакет устройству, что приводило к странным и, казалось бы, совершенно необъяснимым эффектам. Но я быстро догадался в чем проблема и выстроил код драйвера так, чтобы минимизировать число проверок и отправлять подтверждение с опережением, если это допустимо. В целом, проблема разрешилась, хотя это сделало логику работы драйвера немного запутанной. Но я рад тому, что не пришлось делать ассемблерные вставки и еще больше нарушать «чистоту кода».

Какие из всего этого можно сделать выводы?

Во-первых, я убедился в том, что при должном старании и наличии огромного количества времени, одному человеку вполне по силам разобраться в том, что наверчено

сотней других за десяток лет. Особенно если под рукой имеются хорошие приборы, доступна документация/спецификация и такой мощный инструмент такой как SpinalHDL.

Во-вторых, визуализация — это обязательный инструмент при рефакторинге или погружении в сложную систему. Мне никогда ранее не доводилась строить диаграммы для собственного кода, такая деятельность всегда казалась мне бессмысленной тратой времени — ведь вполне достаточно комментариев и набора тестов. ;)

В третьих. Даже такую сложную технологию как USB (сложнее, пожалуй, только PCIe), можно реализовать в очень минималистичном, понятном и достаточном для покрытиях большого спектра задач, исполнении. Иными словами, сложная система может быть простой и понятной. Но для того, чтобы сделать её простой, необходимо прикладывать массу усилий, проводить неоднократную переработку не только её отдельных элементов, но и всей структуры. Простота и лаконичность требуют времени и усилий, и в современном мире это, конечно же, недостижимо. Но к этому стоит стремиться!

9. Использованные источники

1. Universal Serial Bus Specification Revision 1.0. [Электронный ресурс] URL: https://ieeemilestones.ethw.org/w/images/4/44/USB_1.0_Specification.pdf (дата обращения: 15.08.2025).
2. Universal Serial Bus Common Class Specification. [Электронный ресурс] URL: https://usb.org/sites/default/files/usbccs10.pdf (дата обращения: 22.09.2025).
3. USB Communication. Wikipedia the Free Encyclopedia. [Электронный ресурс] URL: https://en.wikipedia.org/wiki/USB_communications (дата обращения: 18.10.2025).
4. USB in a NutShell. Making sense of the USB standard. [Электронный ресурс] URL: https://www.beyondlogic.org/usbnutshell/usb1.shtml (дата обращения: 03.11.2025).
5. CYCLIC REDUNDANCY CHECKS IN USB. [Электронный ресурс] URL: https://www.usb.org/sites/default/files/crcdes.pdf (дата обращения: 10.06.2025).
6. AN57294 USB 101: An Introduction to Universal Serial Bus 2.0. [Электронный ресурс] URL: https://www.infineon.com/dgdl/Infineon-AN57294_USB_101_An_Introduction_to_Universal_Serial_Bus_2.0-ApplicationNotes-v09_00-EN.pdf?fileId=8ac78c8c7cdc391c017d072d8e8e5256 (дата обращения: 27.07.2025).
7. HID Usage Tables FOR Universal Serial Bus (USB) Version 1.6. [Электронный ресурс] URL: https://usb.org/sites/default/files/hut1_6.pdf (дата обращения: 12.08.2025).
8. Universal Host Controller Interface (UHCI) Design Guid. REVISION 1.1. March 1996. [Электронный ресурс] URL: https://stuff.mit.edu/afs/sipb/contrib/doc/specs/protocol/usb/UHCI11D.PDF (дата обращения: 19.09.2025).
9. Open Host Controller Interface Specification for USB. Release: 1.0a. 09/14/99. [Электронный ресурс] URL: https://bankowe.net.pl/b/hcir1_0a.pdf (дата обращения: 19.09.2025).

10. eXtensible Host Controller Interface Specification version 1.2. [Электронный ресурс] URL: https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/extensible-host-controler-interface-usb-xhci.pdf (дата обращения: 20.09.2025).
11. NES эмулятор «nand2mario», by hi631 (Hiromichi Kitahara). [Электронный ресурс] URL: https://github.com/hi631/tang-nano-9K/tree/master/NES (дата обращения: 23.07.2025).
12. Karnix synthesizable System-on-Chip based on VexRiscv RISC-V soft-core (KarnixSoC). [Электронный ресурс] URL: https://github.com/pointcheck/KarnixSOC.git (дата обращения: 04.11.2025).
13. Разработка цифровой аппаратуры нетрадиционным методом: Yosys, SpinalHDL, VexRiscv. ч.1 и ч.2. [Электронный ресурс] URL: https://habr.com/ru/articles/801191/ (дата обращения: 05.11.2025) и https://habr.com/ru/articles/802127/ (дата обращения: 05.11.2025).