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 разрабатывают 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 на столько 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***. Не многим 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 на сколько 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. Разъемы стандарта USB 1.0: Tun A (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-323. 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 недостатоков. 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 шины 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).

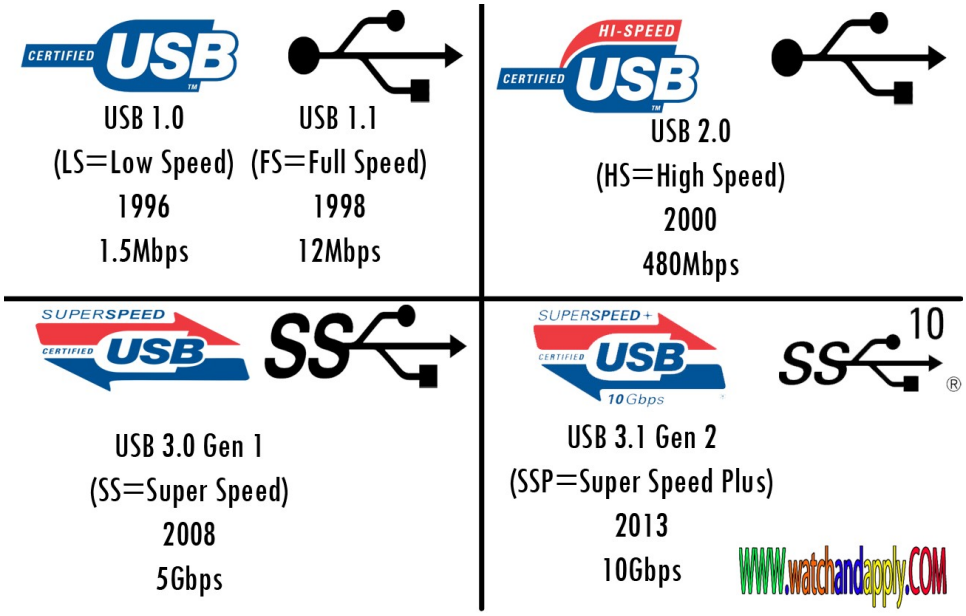

Рис. 2. Логотипы для разъемов USB различных версий.

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 «герлянде» не должен превышать установленный спецификацией для версии порта.

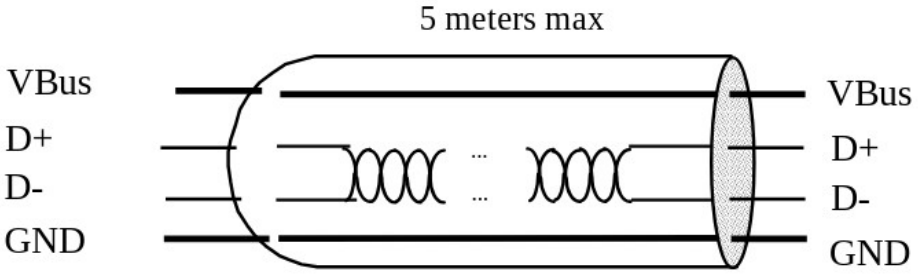

Рис. 3. Физический уровень USB: четыре проводника — два сигнальных и два для питания.

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 находится 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 посмотрим в таблицу 2 позаимствованную со страницы Wikipedia посвященной протоколу обмена USB.

Таблица 2. Состояния шины USB версии 1.0 и 1.1

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.

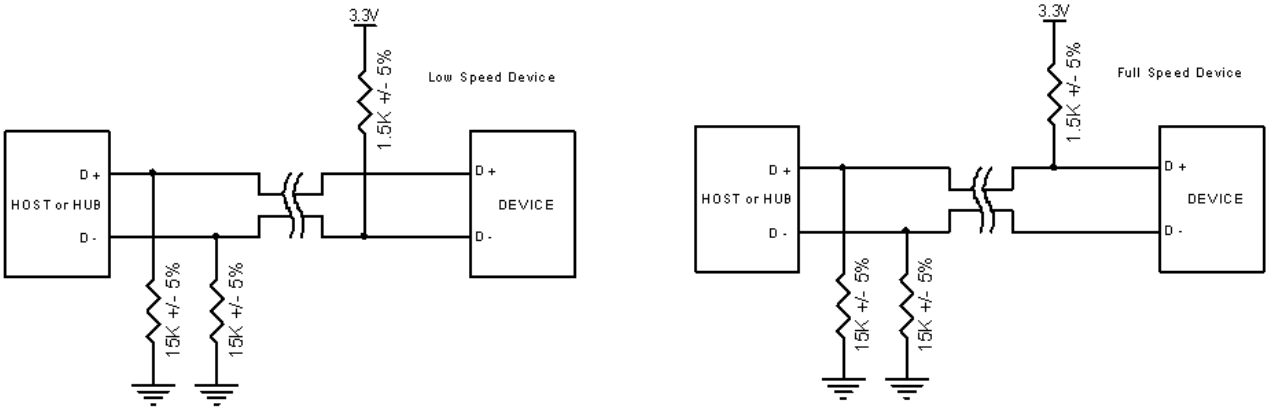

Рис. 4. Физический уровень USB: резисторы подтяжки при подключении «Low Speed» (слева) и «Full Speed» (справа) устройства.

2.2. Передача данных по шине USB

Не смотря 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).

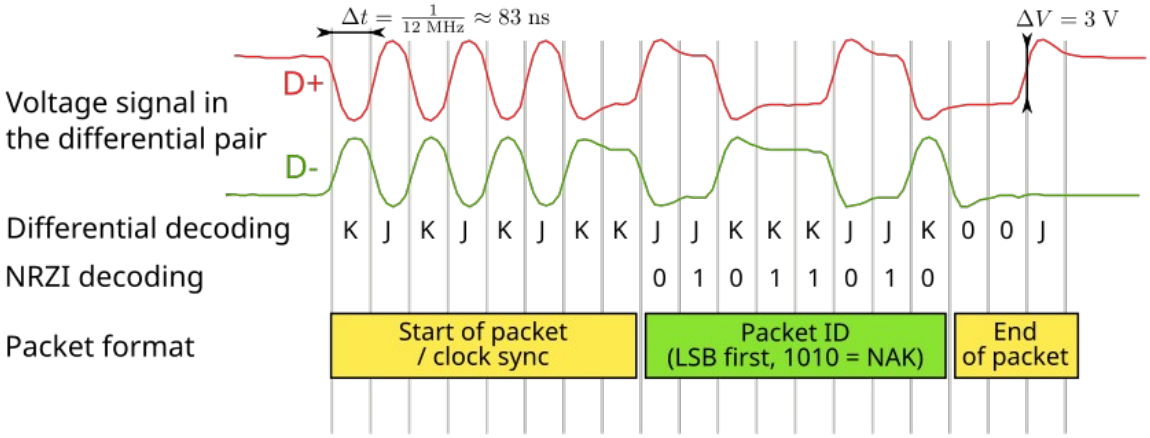

Рис. 5. Последовательность смены состояний шины USB 1.1 «Full Speed» при передаче пакета типа NAK.

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	Пакет с данными НЕ принят, требуется перепосылка (retransmit required).
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 в ответ на принятый пакет DATAx если пакет с данными был принят с ошибкой CRC или принят не полностью. Отправка NAK означает, что передающая сторона должна повторить отправку пакета с данными.

Пакет STALL — высылается в ответ на принятый пакет DATAx в том случае, если на принимающей стороне произошла невосстановимая ошибка (не верный формат данных, не поддерживаемая функция или сбой конечного автомата). Посылка этого пакета означает, что дальнейшая передача данных невозможно 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 котором 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 интервала. 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 (retrnasmission).

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.

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 определены следующим образом: **не более 8 байт** для «Low Speed» и **не более 64 байт** для «Full Speed». В USB 2.0 размер полезных поля для данных увеличен до 1024 байт. Пакет с данными всегда содержит поле циклического кода CRC16 для определения ошибки возникшей при передачи. Полный формат пакета данных приведен в таблице 6.

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

Поле	SYNC	PID	Data	CRC16	EOP
Битовые	**8**	**8**	**64**	**16**	**3**

интервалы					
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 содержащий 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 «тайамаут»), 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 пакета 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. Алгоритм расчета контрольных сумм CRC5 и CRC16

Вычисление циклических контрольных сумм («Cyclic Dedundency 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 передачи должна вызывать несовпадение на стороне приемника сумм 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, на сколько хорошо данный алгоритм позволяет обнаруживать ошибки. Также очевидно что делителем не может быть число ноль. Поэтому форму порождающего полинома подбирают под конкретных случай, используют *неприводимые* полиномы, а входные данные в алгоритме расчета CRC подвергают дополнительной обработки — 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

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

```
// Cortesy: 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 модет 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 передача 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 Trnasfer» 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 содержащего 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 течении последующих 10 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	KJ KJ KJ KK	8'b1101_0010	SE0

Device → Host			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 другу 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_0000	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 управлющих 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 редка 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 точке 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 «Bulf Trnasfer» 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 Transfer» - 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

(определенной 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 ни как 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 блок 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, то 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.

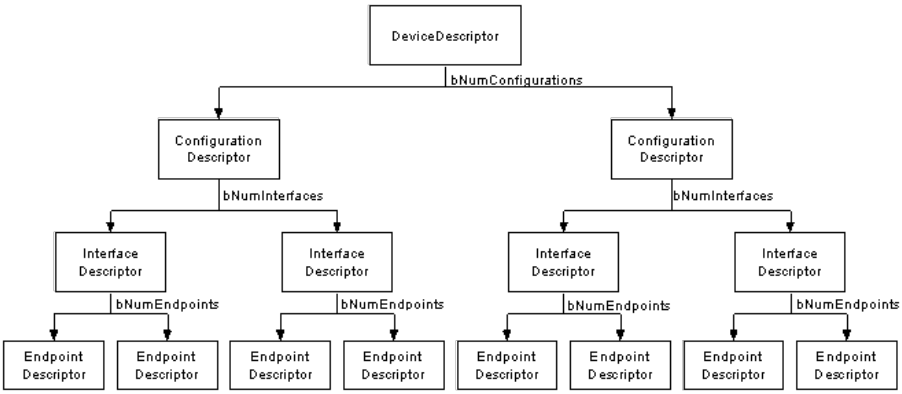

Рис. 6. Древовидная структура данных USB (дескрипторы).

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). По мимо этого, разные конфигурации могут содержать различные наборы интерфейсов и конечных точек. Не смотря 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.

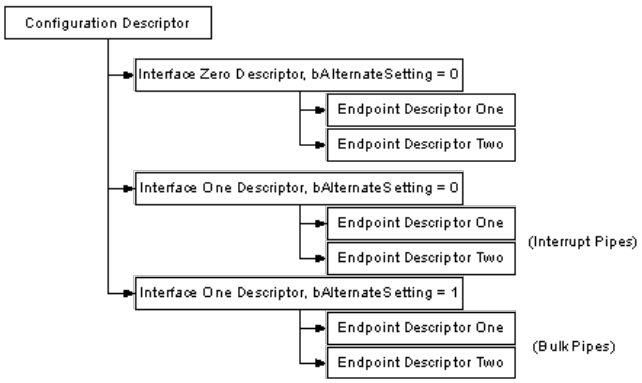

Рис. 7. Пример конфигурации интерфейсас двумя альтернативными настройками.

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.

Важно! 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»

Смеще-ние	**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.

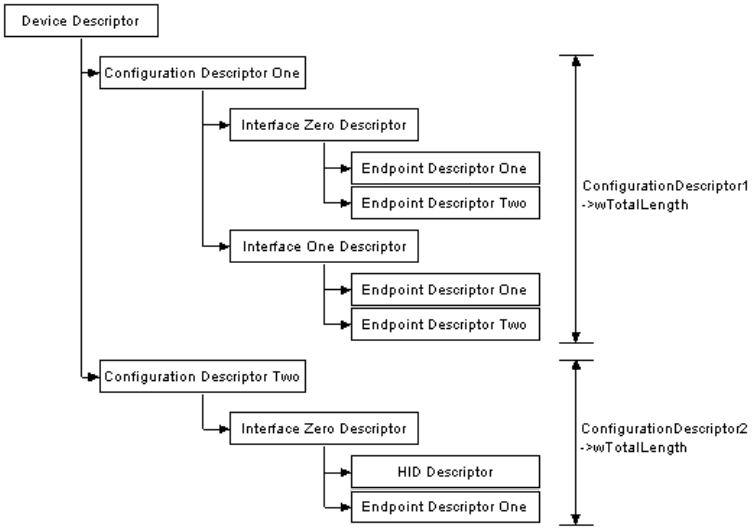

Рис. 8. Пример конфигурационной иерархии.

3.3.3. «Interface Descriptor»

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

Таблица 8.3. Описание полей структуры «Interface Descriptor»

Смеще-ние	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»

Смеще-ние	**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 единицах. Для USB 2.x — в 125 мкс. Может принимать значения от 1 до 255. Для «Isochronous Transfer» должен быть установлен в 1, то есть 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 представляются 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 стоки **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.

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	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 вольных 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 92мА (**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 каналы 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 хам 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 заблокированнымии) «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. По скольку 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.

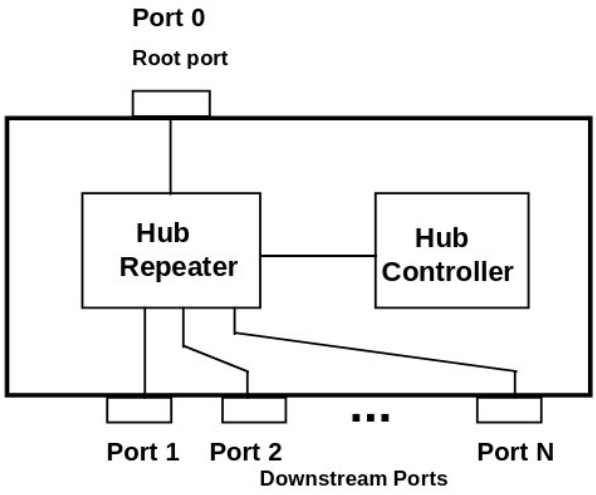

Рис. 9.1. Структура типового USB Hub-а.

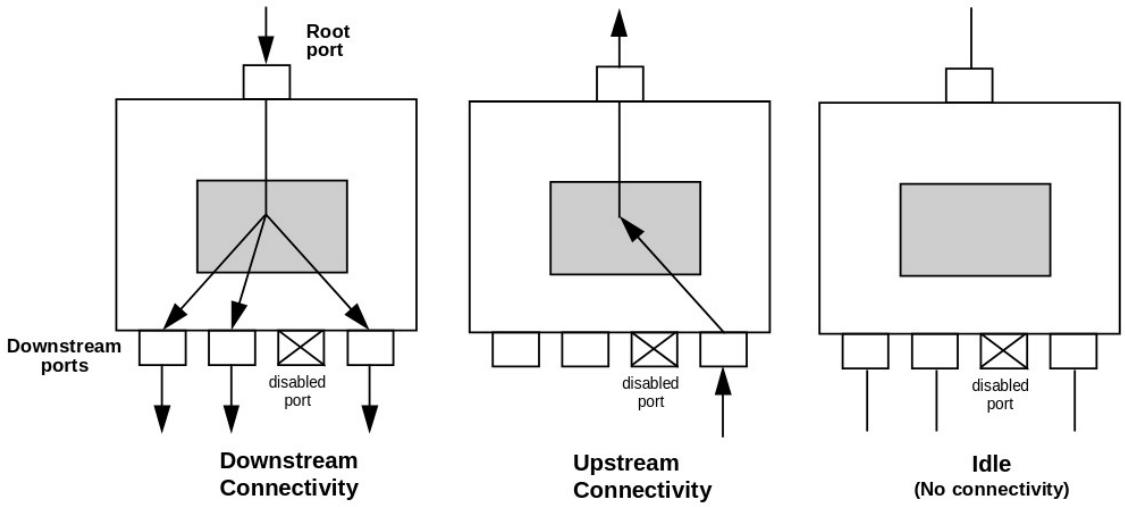

Рис. 9.2. Варианты коммутации портов внутри USB хаба.

3.6. Процедура инициализация USB устройства

Теперь когда у нас есть понимание того, как передаются данных 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 пакету 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 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 не полный 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» содержащий 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 будет 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 не малый. 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 битсримом содержащий 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».

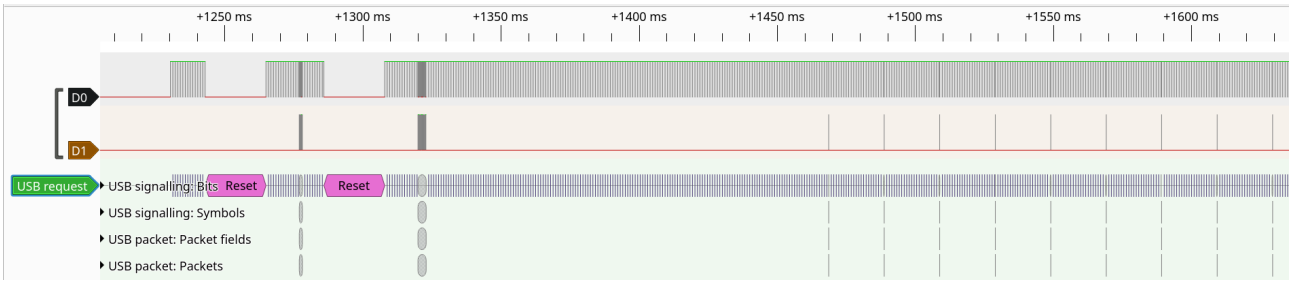

Рис. 10.1. Общий вид процедуры инициализации USB устройства.

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».

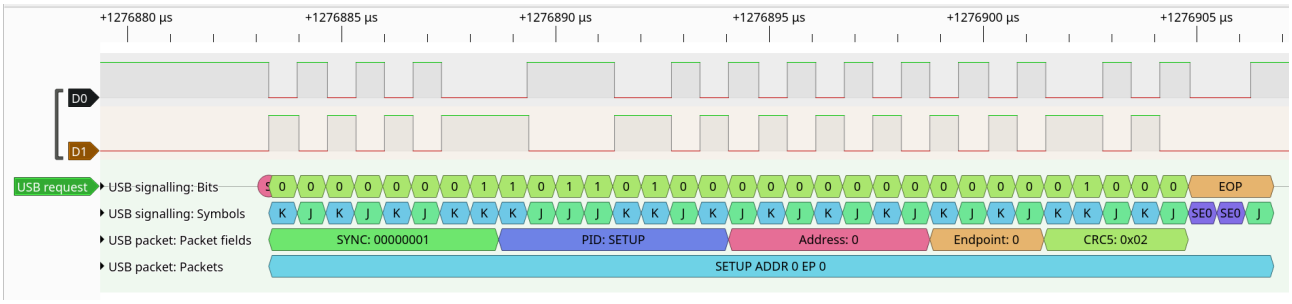

Рис. 10.2а. Пакет SETUP адресованный на 0:0.

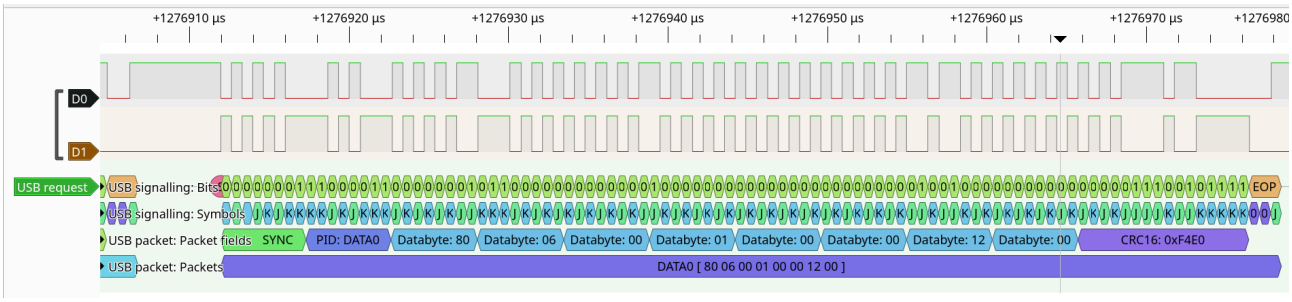

Рис. 10.2б. Пакет DATA0 содержащий запрос «Device Descriptor Request».

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.

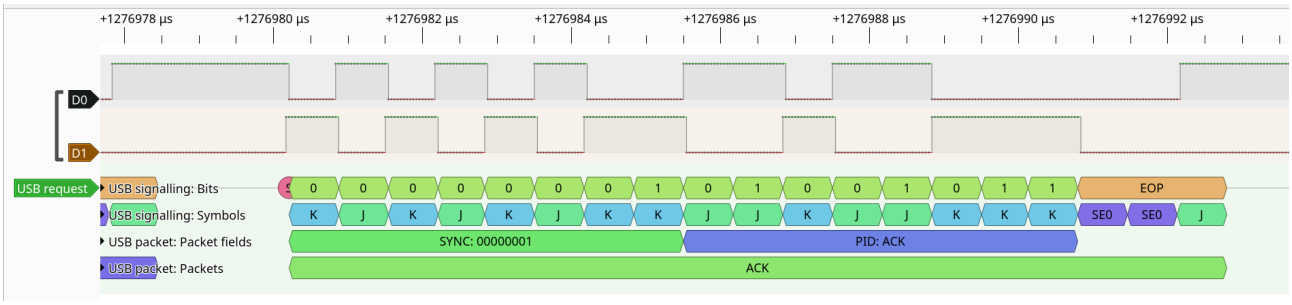

Рис. 10.2в. Пакет ACK от устройства подтверждающие успешный прием данных.

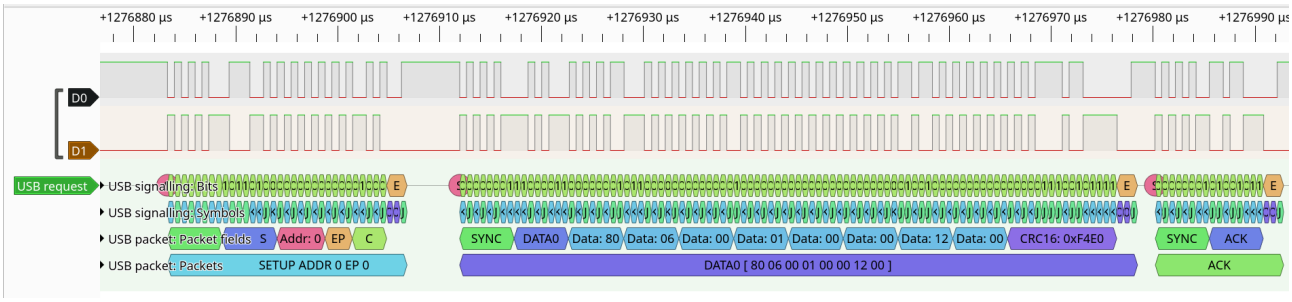

Рис. 10.2г. Пакеты SETUP, DATA0 и ACK на одной осциллограмме.

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.

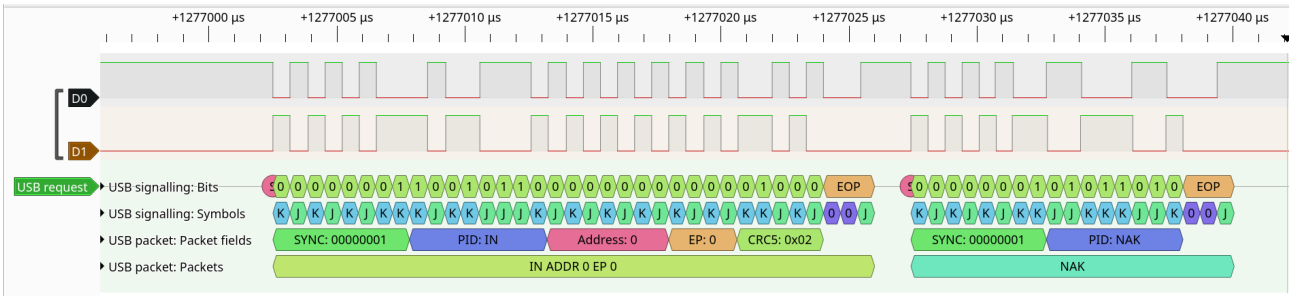

Рис. 10.3а. Хост опрашивает устройство на предмет наличие ответа отсылая токен IN. Устройство отрицает готовность данных токеном NAK.

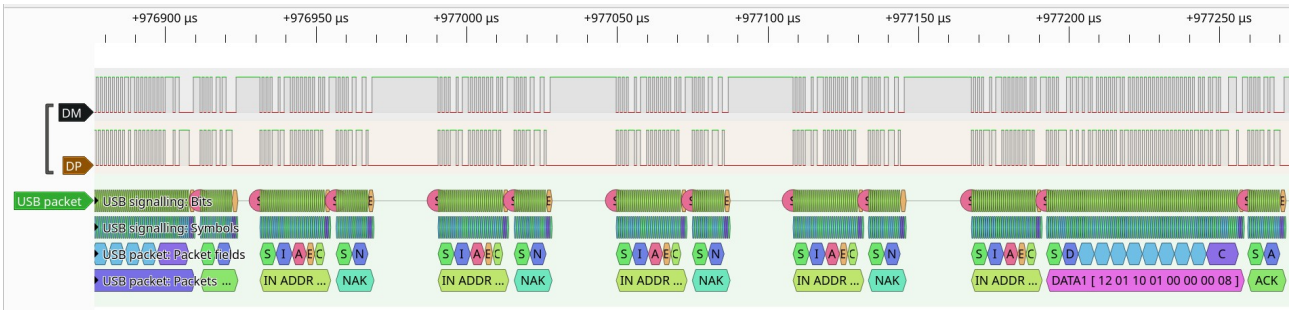

Рис. 10.3б. Хост отправляет пять раз IN перед тем, как получит пакет с данным DATA1.

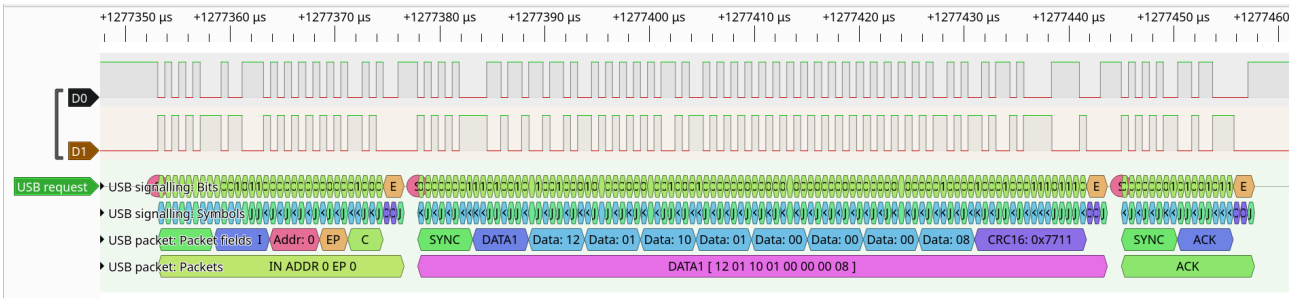

Рис. 10.3в. Хост отправляет токен IN, получает пакет с данным DATA1 и подтверждает его приём токеном ACK.

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.

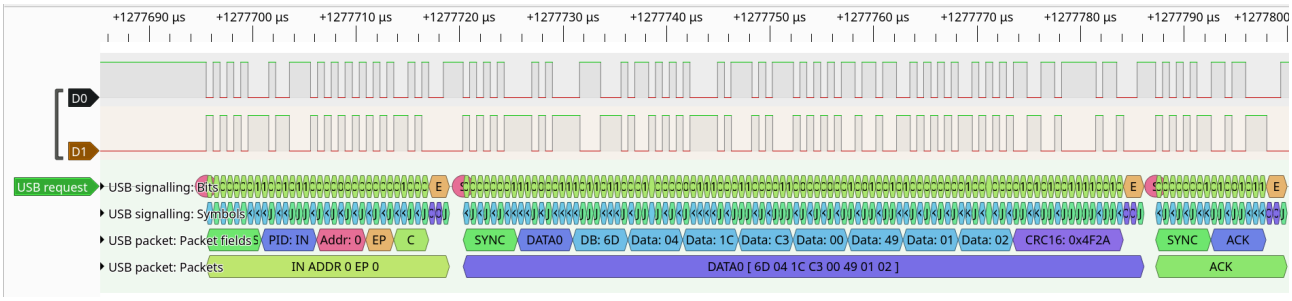

Рис. 10.3г. Хост повторяет токен IN, незамедлительно получает пакет с данным DATA0 и подтверждает его приём токеном ACK.

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».

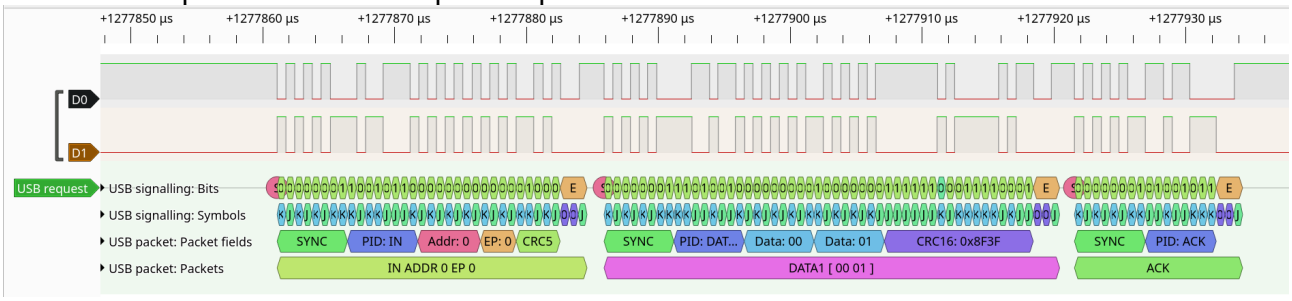

Рис. 10.3д. Хост в очередной раз отправляет токен IN, на что устройство выдает последний блок данных размером 2 байта в пакете DATA1.

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 течении заданного времени, то хост будет вынужден перепослать токен OUT и пакет DATA1 еще раз (таймаут).

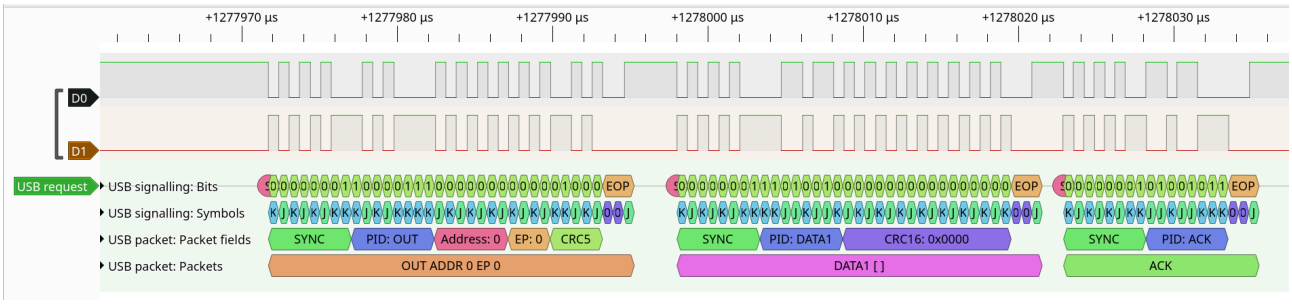

Рис. 10.3е. Хост отправляет токен OUT и следом пустой пакет DATA1 сигнализируя о конце обмена. Устройство подтверждает прием токеном ACK. Конец обмена!

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:

```
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.

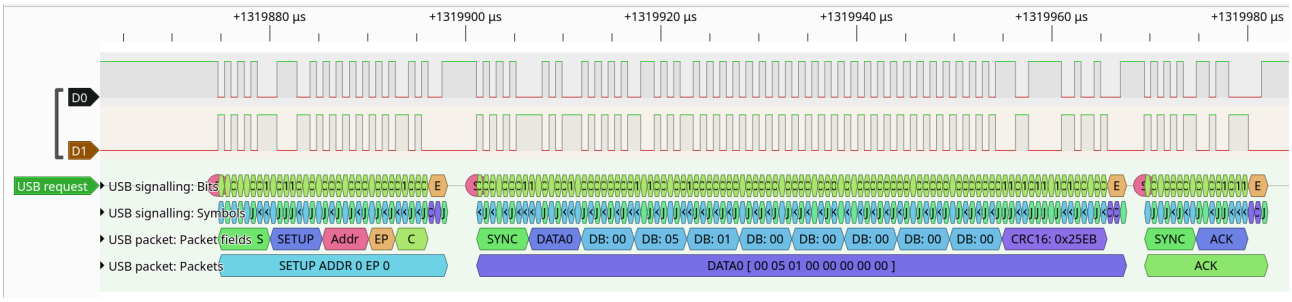

Рис. 10.4а. Хост отправляет токен SETUP и пакет с данными запроса «Device SET_ADDRESS Request». Устройство подтверждает успешность приема токеном ACK.

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.

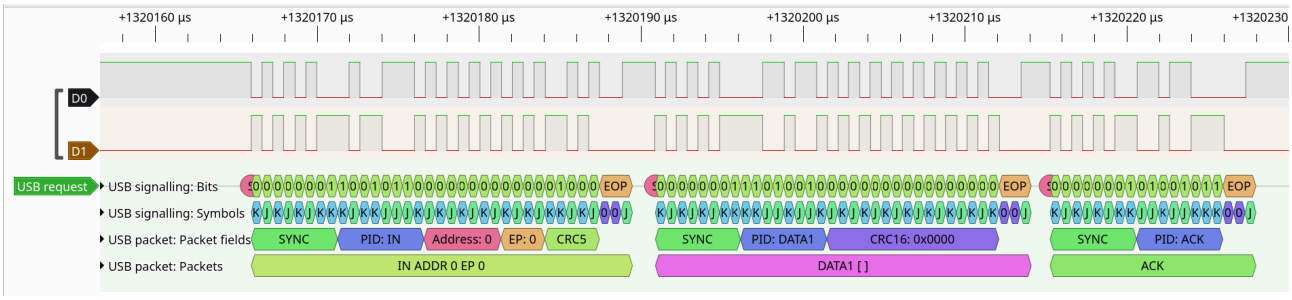

Рис. 10.4б. Хост отправляет токен IN чтобы узнать состояние устройства. Устройство отвечает пустым пакетом DATA1 — признак того, что всё в порядке, запрос обработан.

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 дескпритором 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 быйт) 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.

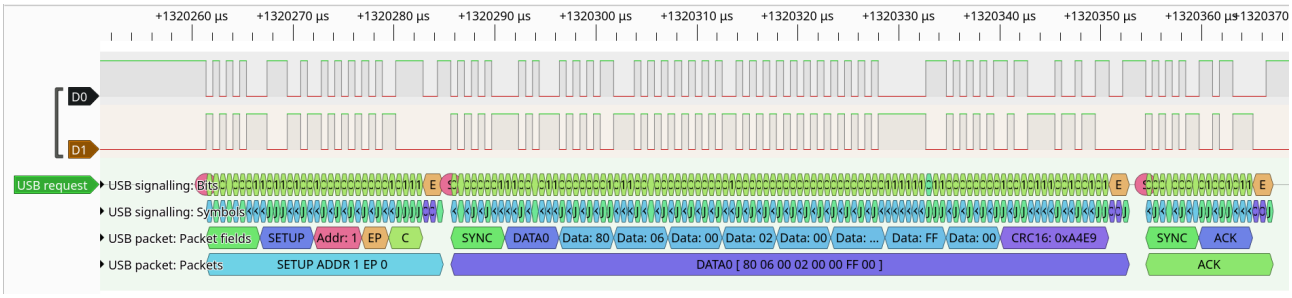

Рис. 10.5а. Хост отправляет токен SETUP и пакет DATA0 содержащий запрос «Configuration Descriptor Request». Устройство подтверждает прием запроса.

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.

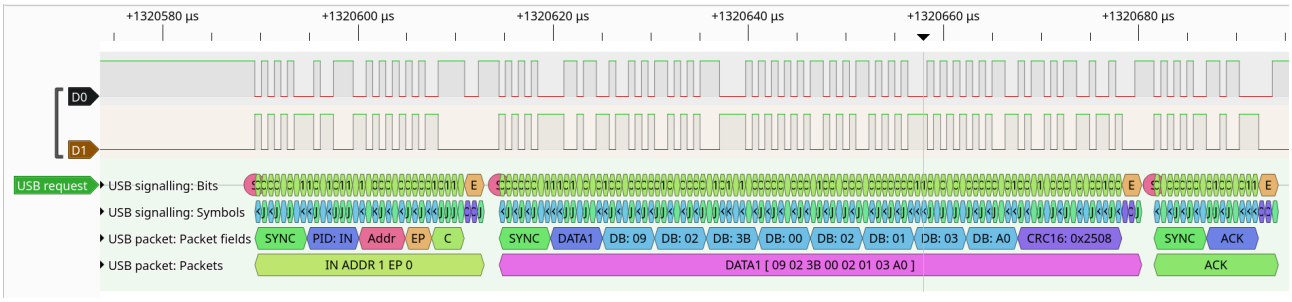

Рис. 10.5б. Хост отправляет токен IN, на что устройство отвечает пакетом DATA1 содержащий часть ответ на запрос «Configuration Descriptor Request». Хост подтверждает прием токеном ACK.

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.

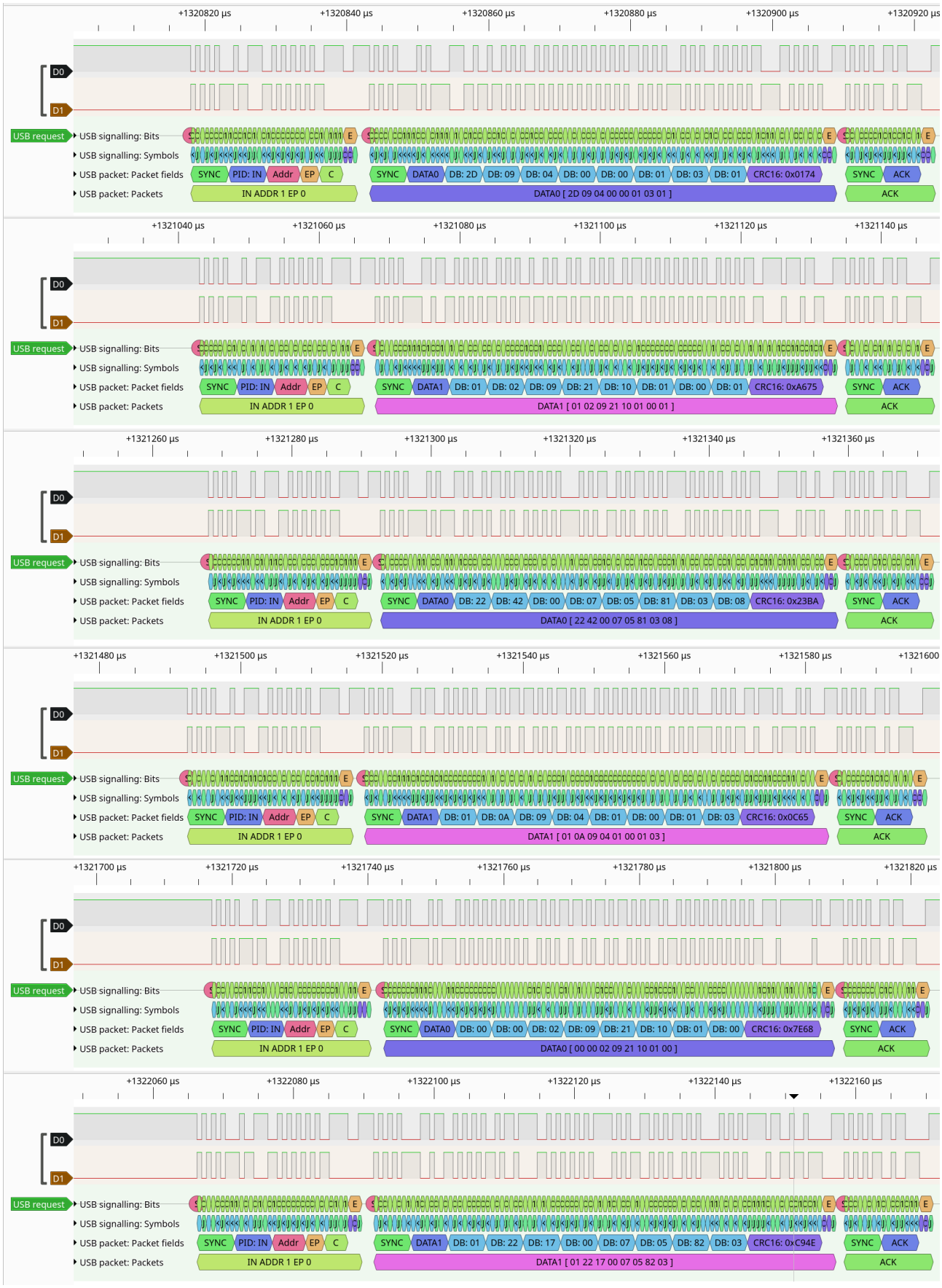

Рис. 10.5в. Хост многократно отправляет токены IN и получает от устройства следующую порцию данных в пакетах DATAx подтверждая прием отсылкой токена ACK.

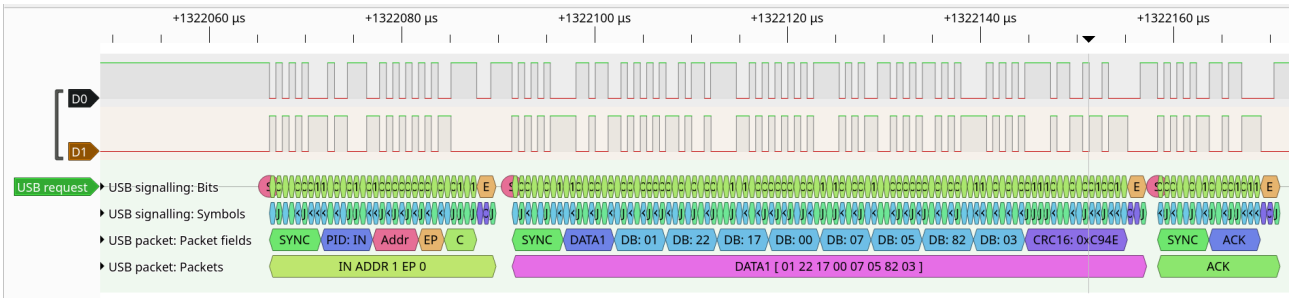

Рис. 10.5в. (продолжение)

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.

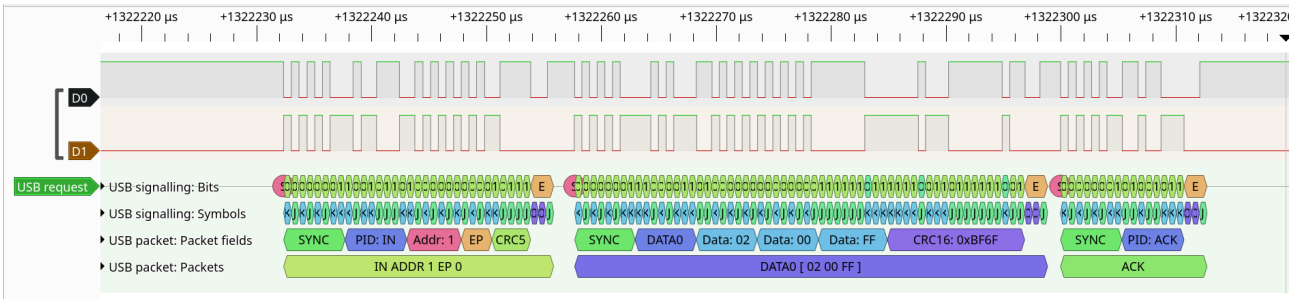

Рис. 10.5г. На очередной токен IN устройство отвечает пакетом DATA0 с размером блока данных менее максимального, что является признаком конца передаваемого ответа.

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

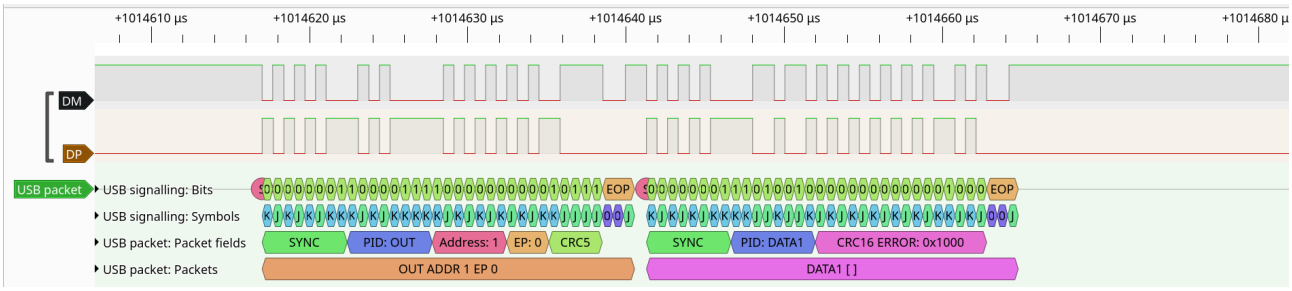

Рис. 10.5д. Зафиксирован сбой при передачи пакета DATA1 — неверный CRC16 (должно быть 0x0000, а принято 0x1000).

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.

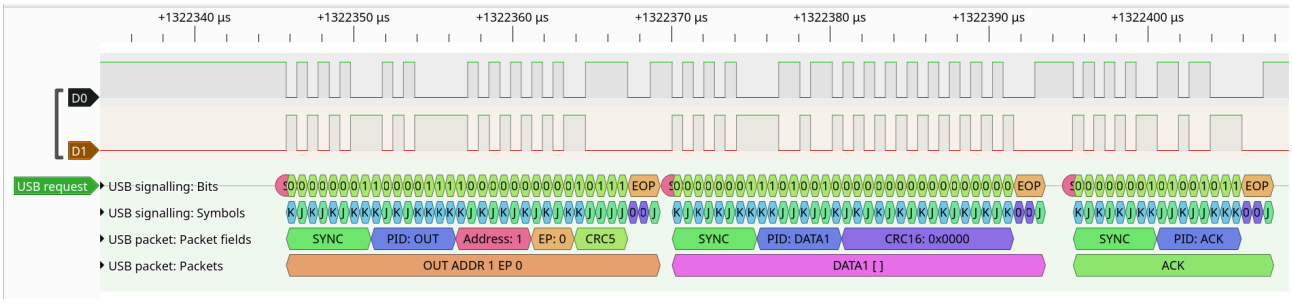

Рис. 10.5е. Перепосылка завершающего OUT и пустого DATA1.

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       – тип структуры «Interface Descriptor»;
bInterfaceNumber      – индексный номер интерфейса (начиная с нуля) равен 0;
bAlternateSetting     – альтернативная функция интерфейса: 0 (отсутствует);
bNumEndpoints         – количество конечных точек в данном интерфейса: 1;
bInterfaceClass       – класс 3 (HID устройство);
bInterfaceSubclass    – подкласс 1 (клавиатура);
bInterfaceProtocol    – протокол опроса 1 (для клавиатуры это «Interrupt Transfer»);
iInterface            – 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 осту (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 - «Pages» (или «страницы»). 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 Descrior» 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.

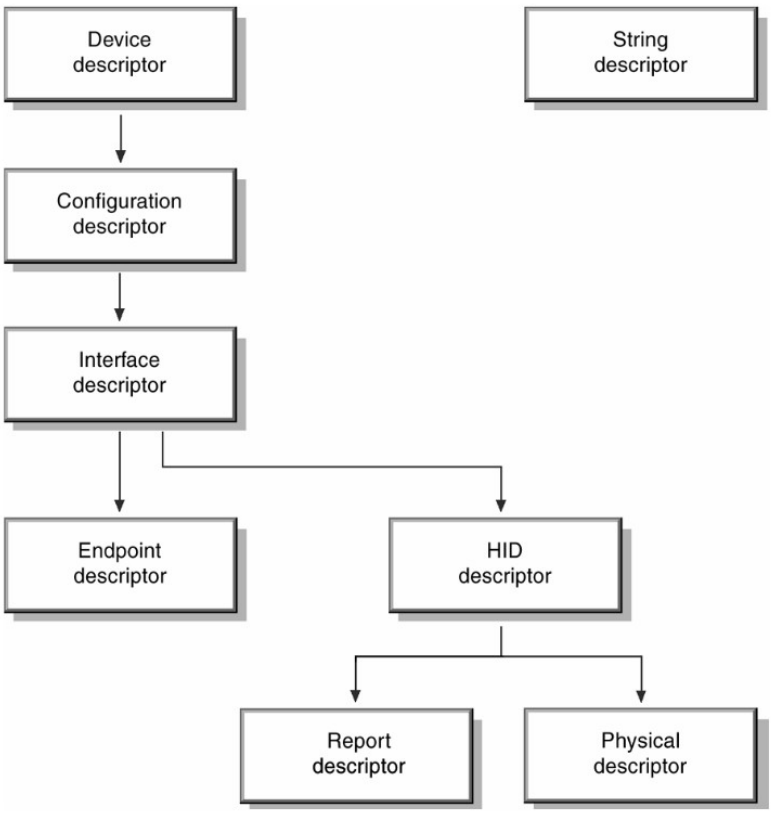

Рис. 11. Дерево дескрипторов для HID устройства.

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

Смещение	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	8’b0010_01_nn		Максимальное логическое

Maximum			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	8’b0100_10_nn		Минимальный идентификатор

Minimum			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 было уже 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».

// Cortesy: 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 изменения конфигурации или состояния отдельных «фич» осуществляется стандартными запросами типа «класс» отправляемыми 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, 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».

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	**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».

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	**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 данным 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** - «Report Protocol».

Таблица 12.5. Формат пакета «GET_PROTOCOL 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	**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.

3.8.5. USB HID «Report Protocol»

Как уже 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.

3.8.6. «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

3.8.7. «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.

3.8.8. USB HID «Boot Protocol»

В какой-то момент разработчики спецификации USB HID осознали, что придуманный им 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 загрузки ПК, код которой ограничен размером микросхемы ПЗУ. Иными словами, потребовалось вводить упрощения и зафиксировать (сделать не изменяемым) 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 не изощренному программному обеспечению легко парсить данных от «Boot» устройств. Чтобы устройство начало выдавать данные в таком предопределенном формате, необходимо установить тип активного протокола в «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	Нет данных	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

3.8.9. 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.

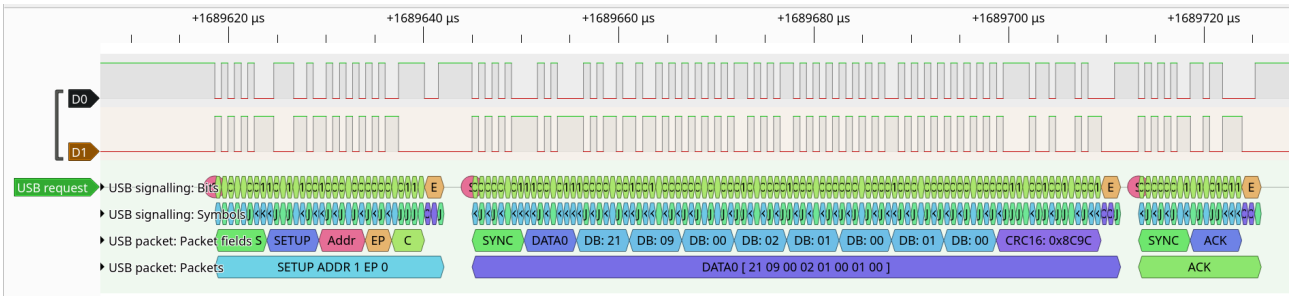

Рис. 12.1. Запрос «SET_REPORT Request» к устройству #1 на дефолтную конечную точку (ENDP=0).

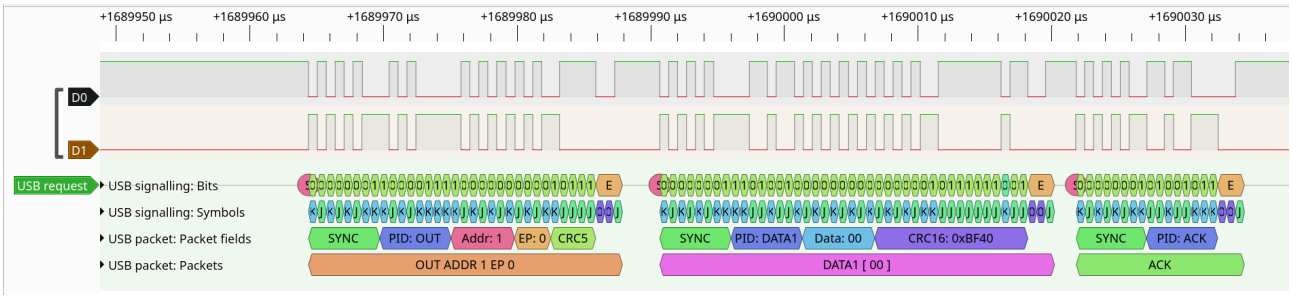

Рис. 12.2. Передача блока данных для запроса «SET_REPORT Request». Содержит один байт полезной нагрузки.

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

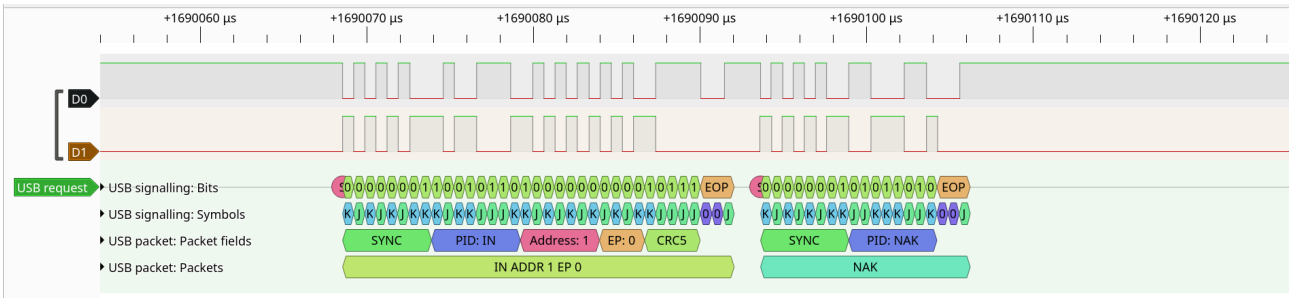

Рис. 12.3. Завершение обмена после запроса «SET_REPORT Request». Устройство отвечает токеном NAK сообщая, что никаких данных у него для хоста нет.

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 устройство, для дальнейшей работы, 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 это 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

3.8.10. 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».

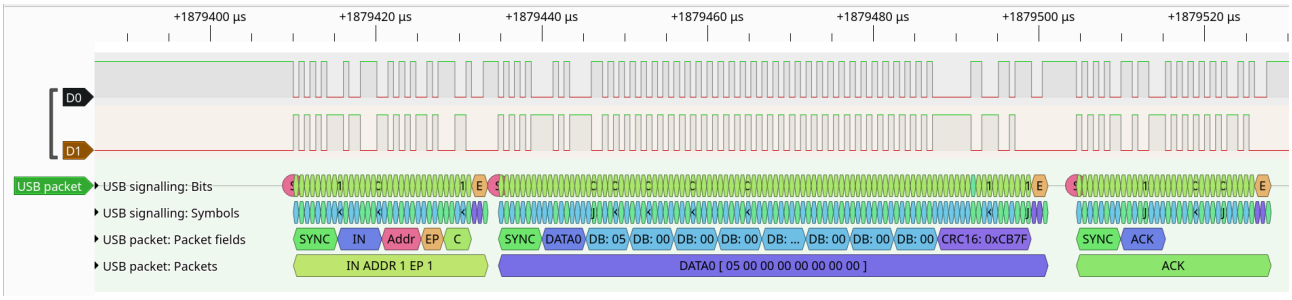

Рис. 12.4. Опрос состояния клавиатуры отправлен на ADDR:ENDP=1:1. Устройство отвечает рапортом в формате определенном Appendix B.1. Нулевой байт значением 0x05 показывает что нажаты две клавиши модификаторов.

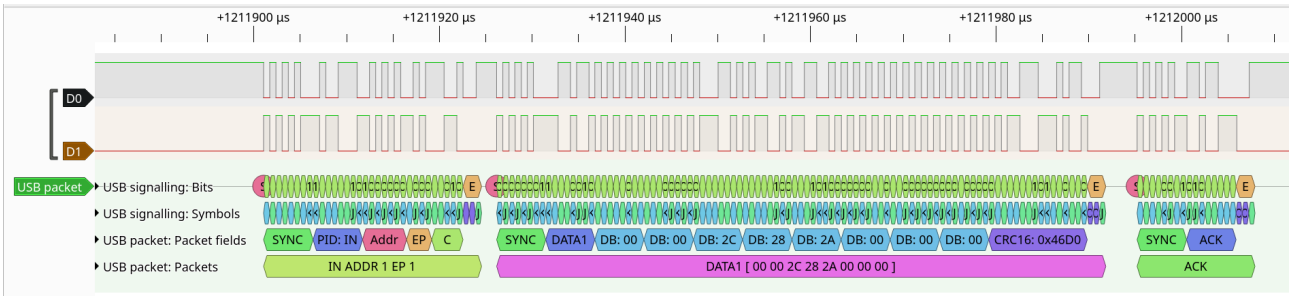

Рис. 12.5. Опрос состояния клавиатуры отправлен на ADDR:ENDP=1:1. Устройство отвечает рапортом в формате определенном Appendix B.1 — одновременно нажаты три клавиши с кодами 0x2C, 0x28 и 0x2A.

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.

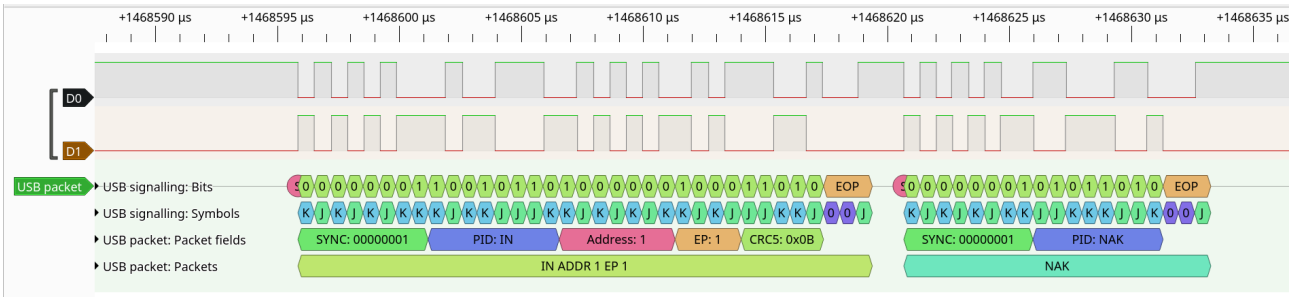

Рис. 12.6. Опрос состояния клавиатуры отправлен на ADDR:ENDP=1:1. Устройство отвечает отсутствием готовых данных (NAK).

3.8.12. 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.

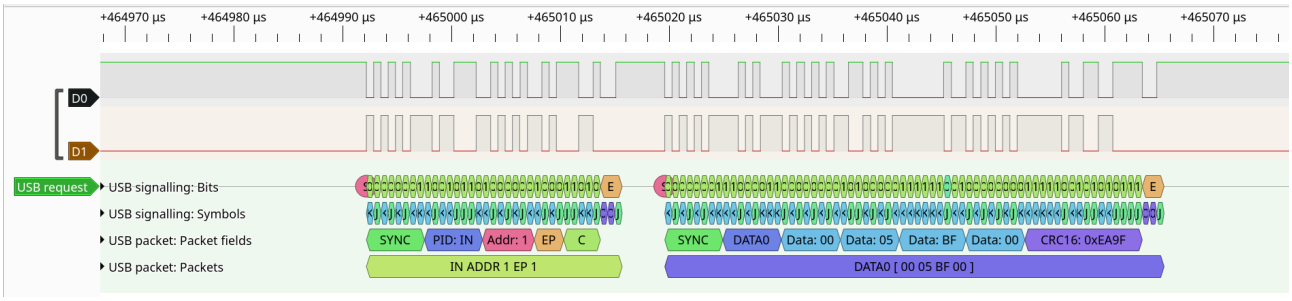

Рис. 12.7. Опрос состояния манипулятора «мышь» отправлен на ADDR:ENDP=1:1. Устройство отвечает рапортом в формате определенном Appendix B.2.

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 контроллера.

4. Разработка контроллера

4.1. Хост контроллеры UHCI и OHCI

UHCI (Universal Host Controller Interface) — от Intel, больше нагрузки на драйвер.

OHCI (Open Host Controller Interface) — от Compaq, Microsoft и National Semiconductors, упрощенная софтверная модель за счет более сложной аппаратуры.

EHCI (Enhanced Host Controller Interface) — для USB 2.0, от Intel, Compaq, NEC, Microsoft и все остальные. Единый драйвер который сейчас поддерживается всеми опеарционными системами.

XHCI — еще более расширенный.

Задачи USB контроллера на хосте:

• Detecting the attachment and removal of USB devices

• Managing USB standard control flow between the host and USB devices

• Managing data flow between the host and USB devices

• Collecting status and activity statistics

• Controlling the electrical interface between the host controller and USB devices, including the provision of a limited amount of powe

Замечение для Хабов:

All hubs provide a status change pipe (see Chapter 11) on which status changes for hubs and their ports are reported. This includes a notification of when a USB device is attached to or removed from one of their ports.

Про 1-мс фреймы:

SOF (Start of Frame) Every millisecond (12000 full-bandwidth bit times), the USB host transmits a special *SOF* (start of frame) token, containing an 11-bit incrementing frame number in place of a device address. This is used to synchronize isochronous and interrupt data transfers.

The host controller must allow the length of the USB frame to be adjusted by ±1 bit time (refer to

Section 10.5.3.2.2). The host controller maintains the current frame number which may be read by the USB system. The current frame number is used to uniquely identify one frame from another.

5. Использованные источники

1. Universal Serial Bus Specification Revision 1.0.
https://ieeemilestones.ethw.org/w/images/4/44/USB_1.0_Specification.pdf

2. Universal Serial Bus Common Class Specification.
https://usb.org/sites/default/files/usbccs10.pdf

3. USB Communication. Wikipedia the Free Encyclopedia.
https://en.wikipedia.org/wiki/USB_communications

4. USB in a NutShell. Making sense of the USB standard.
https://www.beyondlogic.org/usbnutshell/usb1.shtml

5. CYCLIC REDUNDANCY CHECKS IN USB.
https://www.usb.org/sites/default/files/crcdes.pdf

6. AN57294 USB 101: An Introduction to Universal Serial Bus 2.0.
https://www.infineon.com/dgdl/Infineon-AN57294_USB_101_An_Introduction_to_Universal_Serial_Bus_2.0-ApplicationNotes-v09_00-EN.pdf?fileId=8ac78c8c7cdc391c017d072d8e8e5256

7. HID Usage Tables FOR Universal Serial Bus (USB) Version 1.6.
https://usb.org/sites/default/files/hut1_6.pdf