Изобретаем библиотеку vusb

Введение

После прочтения названия может возникнуть закономерный вопрос: зачем в наше время изучать программную реализацию low-speed USB, когда существует куча дешевых контроллеров с аппаратным модулем? Дело в том, что аппаратный модуль, скрывая уровень обмена логическими уровнями, превращает протокол USB в своеобразную магию. Для того же, чтобы прочувствовать как эта «магия» работает, нет ничего лучше, чем воспроизвести ее с нуля, начиная с самого низкого уровня.

С этой целью попробуем изготовить на основе контроллера ATmega8 устройство, прикидывающееся USB-HID`ом. В отличие от распространенной литературы, мы пойдем не от теории к практике, от самого нижнего уровня к верхнему, от логических напряжений на выводах, и закончим «изобретением» той самой vusb, после каждого шага проверяя, работает ли код как ожидалось. Отдельно отмечу, что не изобретаю альтернативу этой библиотеке, а напротив, последовательно воспроизвожу ее исходный код, максимально сохраняя оригинальную структуру и названия, поясняя, для чего служит тот или иной участок. Впрочем, привычный для меня стиль написания кода отличается от стиля авторов vusb. Сразу же честно признаюсь, что помимо альтруистического интереса (рассказать другим сложную тему) имею и корыстный — изучить тему самостоятельно и через объяснение выловить для себя максимум тонких моментов. Отсюда же следует, что какой-то важный момент может быть упущен, или какая-то тема не до конца раскрыта.

Для лучшего восприятия кода я постарался выделить измененные участки комментариями и убрать оные из участков, рассмотренных ранее. Собственно, исходный код и будет основным источником информации, а текст — пояснением что и для чего делалось, а также какой результат ожидается.

Также отмечу, что рассматривается только low-speed USB, даже без упоминания, чем отличаются более скоростные разновидности.

Шаг 0. Железо и прочая подготовка

В качестве подопытного возьмем самодельную отладочную платку на основе ATmega8 с кварцем 12 МГц. Схему приводить не буду, она вполне стандартна (см. официальный сайт vusb), единственное что стоит упомянуть, так это используемые выводы. В моем случае выводу D+ соответствует PD2, выводу D- PD3, а подтяжка висит на PD4. В принципе, подтягивающий резистор можно было соединить и с питанием, но ручное управление им кажется чуть более соответствующим стандарту.

С разъема USB подается питание 5 В, однако на сигнальных линиях ожидается не более 3.6 В (зачем так было сделано для меня загадка). Значит нужно либо понизить питание контроллера, либо поставить стабилитроны на сигнальные линии. Я выбрал второй вариант, но по большому счету это не принципиально.

Раз уж мы «изобретаем» реализацию, было бы неплохо видеть, что же происходит в мозгах контроллера, то есть необходима хоть какая-то отладочная информация. В моем случае это два светодиода на PD6, PD7 и, самое главное, UART на PD0, PD1, настроенный на 115200, так что болтовню контроллера можно будет слушать через обычный screen или другую программу для работы с COM-портом:

```
$ screen /dev/ttyUSB0 115200
```

Также полезной утилитой при отладке USB окажется wireshark с соответствующим модулем (он, правда, не всегда запускается с «из коробки», но решение подобных проблем вполне успешно находится в интернете и не является задачей данной статьи).

Здесь можно было бы потратить еще килобайт текста на описание программатора, makefile`ов и прочего, но вряд ли это имеет смысл. Точно так же не буду акцентировать внимание на настройках периферии, не связанных с USB. Если кто-то не может разобраться даже с этим, тем, быть может, и в недра программного USB лезть рано?

Шаг 1. Принимаем хоть что-то

Согласно документации, USB поддерживает несколько фиксированных скоростей, из которых AVR потянет только самую низкую: 1.5 мегабита в секунду. Она определяется по подтягивающему резистору и последующему общению. Для выбранной нами частоты резистор должен соединять D- с питанием 3.3 В и иметь номинал 1.5 кОм, но на практике можно соединять и с +5 В, и номинал немного варьировать. При частоте контроллера 12 МГц на один бит приходится всего 8 тактов. Понятно, что такая точность и скорость достижимы только на ассемблере, так что заведем файл **drvasm.S**. Отсюда же следует необходимость использования прерывания для отлова начала приема байта. Радует, что первый байт, передаваемый по USB, всегда одинаковый, SYNC, так что если попадем не в начало, ничего страшного. В результате с момента начала байта до его конца проходит всего 64 такта контроллера (на самом деле запас еще меньше), так что использовать другие прерывания, не связанные с USB не стоит.

Сразу же вынесем конфигурацию в отдельный файл **usbconfig.h**. Именно там будут заданы выводы, отвечающие за USB, а также используемые биты, константы и регистры.

Теоретическая вставка

Передача по протоколу USB осуществляется пакетами по нескольку байт в каждом. Первым байтом всегда идет байт синхронизации SYNC, равный 0b10000000, вторым — байт-идентификатор пакета PID. Передача каждого байта идет от младшего бита к старшему (это не совсем так, но в vusb эту тонкость игнорируют, учитывая в другом месте) при помощи кодирования NRZI. Этот способ заключается в том, что логический ноль передается изменением логического уровня на противоположный, а логическая единица — не-изменением. Кроме того, вводится защита от рассинхронизации (которой мы пользоваться не будем, но учитывать должны) источника и приемника сигнала: если в передаваемой последовательности есть шесть единиц подряд, то есть шесть тактов подряд состояние выводов не меняется, в передачу добавляется принудительная инверсия, как будто передается ноль. Таким образом размер байта может составлять 8 или 9 бит.

Еще стоит упомянуть о том, что линии данных в USB дифференциальные, то есть когда на D+ высокий уровень, на D- он низкий (это называется K-состояние) и наоборот (J-состояние). Это сделано для лучшей помехозащищенности на высокой частоте. Правда, есть и исключение: сигнал конца пакета (он называется SE0) передается притягиванием обеих сигнальных линий к земле (D+ = D- = 0). Есть еще два сигнала, передаваемые удерживанием на линии D+ низкого напряжения, а на D- высокого, в течение различного времени. Если время небольшое (длина одного байта или чуть больше) то это Idle, пауза между пакетами, а если большое — сигнал сброса.

Итак, передача идет по дифференциальной паре, не считая экзотического случая SE0, но его пока рассматривать не будем. Значит для определения состояния шины USB нам достаточно одной линии, D+ или D-. По большому счету, разницы какую выбрать, нет, но для определенности пусть будет D-.

Начало пакета можно определить по приему байта SYNC после длительного Idle`а. Состоянию Idle соответствует лог.1 на линии D- (оно же J-состояние), а байт SYNC равен 0b100000, но он передается от младшего бита к старшему, да еще закодирован в NRZI, то есть каждый ноль означает инверсию сигнала, а единица — сохранение прежнего уровня. Значит последовательность состояний D- будет следующей:

байт	Idle					SYNC								PID							
USB	-	...	-	-	-	0	0	0	0	0	0	0	1	?	?	?	?	?	?	?	?
D-	1		1	1	1	0	1	0	1	0	1	0	0	?	?	?	?	?	?	?	?

Начало пакета проще всего детектировать по спадающему фронту, на него и настроим прерывание. Но вдруг во время начала приема контроллер будет занят и не сможет войти в прерывание немедленно? Чтобы в такой ситуации не сбиться со счета тактов, воспользуемся байтом SYNC по прямому назначению. Он сплошь состоит из фронтов на границах битов, так что мы можем подождать один из них, потом еще пол-бита, и попадем прямо в середину следующего. Впрочем, ждать «какого-нибудь» фронта — не лучшая идея, нам ведь надо не просто попасть в середину бита, но и знать в какой по счету бит мы попали. И для этого SYNC также подходит: у него в конце идут два нулевых бита подряд (они же K-состояния). Вот их и будем ловить. Итак, в файле **drvasm.S** появляется кусок кода от входа в прерывание до **foundK**. Причем за счет времени на проверку состояния порта, на безусловный переход и прочее, в метку мы попадаем далеко не в начале бита, а как раз к середине. Но проверять этот же бит бессмысленно, мы ведь и так знаем его значение. Поэтому ждем 8 тактов (пока что пустыми nop`ами) и проверяем следующий бит. Если он тоже нулевой, то мы нашли конец SYNC`а и можем переходить к приему значащих битов.

Собственно, весь дальнейший код предназначен для считывания еще двух байтов с последующим выводом на UART. Ну и ожидание состояния SE0 чтобы не попасть случайно в следующий пакет.

Теперь можно скомпилировать полученный код и посмотреть, какие же байты принимает наше устройство. Лично у меня последовательность следующая:

```
4E 55 00 00 4E 55 00 00 4E 55 00 00 4E 55 00 00 4E 55 00 00
```

Напоминаю, выводим мы сырые данные, без учета добавочных нулей и декодирования NRZI. Попробуем декодировать вручную, начиная с младшего бита:

NRZI	4				E				Пред. бит
	0	1	0	0	1	1	1	0	0
байт	0	0	1	0	1	1	0	1	
	2				D				

NRZI	5				5				Пред. бит
	0	1	0	1	0	1	0	1	0
байт	0	0	0	0	0	0	0	0	
	0				0				

Нули декодировать не имеет смысла, поскольку 16 одинаковых значений подряд не могут входить в пакет.

Таким образом, нам удалось написать прошивку, принимающую первые два байта пакета, правда пока без декодирования.

Шаг 2. Демо-версия NRZI

Чтобы не перекодировать вручную, можно поручить это самому контроллеру: операция XOR делает именно то, что нужно, правда результат у нее получается инвертированный, поэтому после нее добавим еще одну инверсию:

```
    mov temp, shift
    lsl shift
    eor temp, shift
```

```
    com temp
    rcall uart_hex
```

Результат вполне ожидаем:

```
2D 00 FF FF 2D 00 FF FF 2D 00 FF FF 2D 00 FF FF 2D 00 FF FF
```

Шаг 3. Избавляемся от цикла приема байта

Сделаем еще один маленький шажок и развернем цикл приема первого байта в линейный код. При этом получается много nop`ов, нужных только чтобы дождаться начала следующего бита. Вместо некоторых из них можно использовать декодер NRZI, другие пригодятся в дальнейшем.

Результат от предыдущего варианта не отличается.

Шаг 4. Читаем в буфер

Читать в отдельные регистры это, конечно, быстро и красиво, но когда данных становится слишком много, лучше использовать запись в буфер, расположенный где-нибудь в оперативке. Для этого в мейне объявим массив достаточного размера, а в прерывании будем туда писать.

Теоретическая вставка

Структура пакетов в USB стандартизована и состоит из следующих частей: байт SYNC, байт PID+CHECK (2 поля по 4 бита), поле данных (иногда 11 бит, но чаще произвольное количество 8-битных байтов) и контрольная CRC сумма размером либо 5 (для 11-битного поля данных), либо 16 (для остальных) бит. И наконец признак конца пакета (EOP) — два бита паузы, но это уже не данные.

Перед работой с массивом надо еще настроить регистры, а свободных nop`ов перед первым битом для этого недостаточно. Поэтому придется вынести чтение двух первых битов в линейный участок кода, между командами которого вставим код инициализации, а потом прыгнем в середину цикла чтения, на метку **rxbit2**. Кстати о размере буфера. Согласно документации, в одном пакете нельзя передавать больше 8 байт данных. Добавляем служебные байты PID и CRC16, получаем размер буфера 11 байт. Байт SYNC и состояние EOP записывать не будем. Контролировать интервал запросов от хоста мы не сможем, но и терять их не хочется, поэтому для чтения возьмем двойной запас. Пока что мы не будем использовать буфер целиком, но чтобы не возвращаться в будущем, лучше сразу выделить нужный объем.

Шаг 5. Работаем с буфером по-человечески

Вместо прямого чтения первых байтов массива, напишем кусок кода, читающий именно столько байт, сколько было реально записано в массив. И заодно добавим разделитель между пакетами. Теперь вывод выглядит так:

```
>03 2D 00 10 >01 FF >03 2D 00 10 >01 FF >03 2D 00 10 >01 FF >03 2D 00 10 >01 FF
>03 2D 00 10 >01 FF
```

Шаг 6. Добавляем добавку добавочных нулей

Наконец-то пришло время добить чтение потока битов до соответствия стандарту. Последний пункт, без которого мы успешно обходились — фальшивый ноль, добавляемый после каждых шести подряд идущих единиц. Раз уж у нас прием байта развернут в линейное тело цикла, проверять придется после каждого бита, во всех восьми местах. Рассмотрим на примере первых двух битов:

```
unstuff0:                   ;1 (за счет breq)
```

```
    andi    x3, ~(1<<0)     ;1 [15] стираем 0-й бит маски. Он инвертирован не
будет
    mov     x1, x2          ;1 [16] подменяем предыдущий принятый бит текущим
(добавочным)
    in      x2, USBIN       ;1 [17] <-- 1-й бит оказался добавочным. Считываем
вместо него настоящий
    ori     shift, (1<<0)   ;1 [18] выставляем 0-й бит в лог.1 чтобы добавка не
срабатывала повторно
    rjmp    didUnstuff0     ;2 [20]
;<---//--->
rxLoop:
    eor     shift, x3       ;1 [0]
    in      x1, USBIN       ;1 [1]
    st      y+, shift       ;2 [3]
    ldi     x3, 0xFF        ;1 [4]
    nop                     ;1 [5]
    eor     x2, x1          ;1 [6]
    bst     x2, USBMINUS    ;1 [7] записываем считанный бит в 0-й бит регистра
shift
    bld     shift, 0        ;1 [8]
    in      x2, USBIN       ;1 [9] <-- считываем 1-й бит(возможно, добавочный)
    andi    x2, USBMASK     ;1 [10]
      breq  se0             ;1 [11]
    andi    shift, 0xF9     ;1 [12]
didUnstuff0:
      breq  unstuff0        ;1 [13]
    eor     x1, x2          ;1 [14];
    bst     x1, USBMINUS    ;1 [15] записываем считанный бит в 1-й бит регистра
shift
    bld     shift, 1        ;1 [16]
rxbit2:
    in      x1, USBIN       ;1 [17] <-- считываем 2-й бит (возможно, добавочный)
    andi    shift, 0xF3     ;1 [18]
      breq  unstuff1        ;1 [19]
didUnstuff1:
```

Для удобства навигации адреса описываемых команд буду отсчитывать по меткам справа. Обратите внимание, что вводились они для отсчета тактов контроллера, поэтому идут не по порядку. Считывание очередного байта начинается на метке **rxLoop**, проводится инверсия предыдущего байта и запись в буфер [0, 3]. Далее на метке [1] считывается состояние линии D-, по XOR`у с предыдущим принятым состоянием декодируем NRZI (напоминаю, что обычный XOR добавляет свою инверсию, для исправления чего мы вводим регистр маски *x3*, инициализируемый единицами 0xFF) и записываем в 0-й бит регистра *shift* [7,8]. Дальше начинается самое интересное — проверяем не был ли принятый бит шестым неизменным. Неизменному биту, принятому с D- соответствует запись нуля (а не единицы! Менять на единицу будем в конце, XOR`ом) в регистр. Поэтому нужно проверить не являются ли биты 0, 7, 6, 5, 4, 3 нулями. Остальные два бита значения не имеют, они остались от предыдущего байта и были проверены раньше. Чтобы от них избавиться, обрежем регистр по маске [12], где выставлены в 1 все интересующие нас биты: 0b11111001 = 0xF9. Если после наложения маски все биты оказались нулями, ситуация добавления бита зафиксирована и идет переход на метку **unstuff0**. Там считывается еще один бит [17] взамен считанного ранее, в промежутке между другими операциями, лишнего [9]. Также меняем местами регистры текущего и предыдущего значений *x1*, *x2*. Дело в том, что на каждом бите значение считывается в один регистр, а потом XOR`ится с другим, после чего регистры меняются местами. Соответственно, при чтении добавочного регистра эту операцию тоже нужно произвести. Но самое интересное, что в регистр данных *shift* записываем не ноль, который получили честно, а единицу, которую пытался передать хост [18]. Связано это с тем, что при приеме следующих битов значение нулевого тоже придется учитывать, и если бы мы записали ноль, проверка по маске не могла бы узнать что добавочный бит уже учтен. Таким образом, в регистре *shift* все биты инвертированы (относительно передаваемого хостом), а нулевой — нет. Чтобы такая каша не была записана в буфер, обратную инверсию будем проводить по XOR`у не с 0xFF [0], а с 0xFE, то есть регистром, в котором соответствующий бит будет сброшен в 0 и, соответственно, не приведет к инверсии. Для этого на отсчете [15] и сбрасываем нулевой бит.

Аналогичная ситуация происходит с битами 1-5. Скажем, 1-й бит соответствует проверке 1, 0, 7, 6, 5, 4, тогда как биты 2, 3 игнорируются. Этому соответствует маска 0xF3. А вот обработка 6 и 7 битов отличается:

```
didUnstuff5:
    andi    shift, 0x3F     ;1 [45] проверка битов 5-0
      breq  unstuff5        ;1 [46]
;<---//--->
    bld     shift, 6        ;1 [52]
didUnstuff6:
    cpi     shift, 0x02     ;1 [53] проверка битов 6-1
      brlo  unstuff6        ;1 [54]
;<---//--->
    bld     shift, 7        ;1 [60]
didUnstuff7:
    cpi     shift, 0x04     ;1 [61] проверка битов 7-2
      brsh  rxLoop          ;3 [63]
unstuff7:
```

Маской для 6-го бита является число 0b01111110 (0x7E), но накладывать ее на регистр *shift* нельзя, поскольку она сбросит 0-й бит, который должен быть записан в массив. Кроме того, на отсчете [45] уже была наложена маска, обнулившая 7 бит. Значит, обрабатывать добавочный бит надо, если биты 1-6 равны нулю, а 0-й не имеет значения. То есть значение регистра должно быть 0 или 1, что прекрасно проверяется сравнением «меньше, чем 2» [53, 54].

Тот же принцип использован для 7-го бита: вместо наложения маски 0xFC идет проверка на «меньше, чем 4» [61, 63].

Шаг 7. Сортируем пакеты

Раз уж мы можем принимать пакет с первым байтом (PID), равным 0x2D (SETUP), попробуем рассортировать принятое. Кстати, почему это я назвал пакет 0x2D SETUP`ом, когда он вроде бы ACK? Дело в том, что передача по USB от младшего бита к старшему осуществляется в пределах каждого поля, а не байта, тогда как мы принимаем побайтно. Первое значащее поле, PID, занимает всего 4 бита, после чего идет еще 4 бита CHECK, представляющее побитовую инверсию поля PID. Таким образом, первым принятым байтом будет не PID+CHECK, а наоборот, CHECK+PID. Впрочем, особой разницы нет, поскольку все значения известны заранее, и переставить полубайты местами несложно. Вот сразу и запишем в файле **usbconfig.h** основные коды, которые могут нам пригодиться.
Пока не начали дописывать код обработки PID`а, отметим, что он должен быть быстрым (то есть на ассемблере), но вот выравнивание по тактам не требуется, ведь пакет мы уже приняли. Поэтому впоследствии этот участок будет вынесен в файл **asmcommon.inc**, который будет содержать ассемблерный код, не привязанный к частоте. А пока просто выделим комментарием.

Теперь перейдем к сортировке принятых пакетов

Теоретическая вставка

Пакеты данных на шине USB объединяются в транзакции. Каждая транзакция начинается с посылки хостом специального маркер-пакета, несущего информацию о том, что же хост хочет делать с устройством: конфигурировать (SETUP), передавать данные (OUT) или принимать их (IN). После передачи маркер-пакета следует пауза длиной в два бита. Дальше следует пакет данных (DATA0 или DATA1), который может быть послан как хостом, так и устройством, в зависимости от маркер-пакета. Далее еще одна пауза в два бита длиной и ответ — HANDSHAKE, пакет подтверждения (ACK, NAK, STALL, их рассмотрим в другой раз).

SETUP	pause	DATA0	pause	HANDSHAKE
host->device		host->device		device->host

OUT	pause	DATA0 / DATA1	pause	HANDSHAKE
host->device		host->device		device->host

IN	pause	DATA0 / DATA1	pause	HANDSHAKE
host->device		device->host		host->device

Поскольку обмен идет по одним и тем же линиям, хосту и устройству приходится постоянно переключаться между передачей и приемом. Очевидно, двухбитная задержка именно для этого и сделана, чтобы они не начали играть в тяни-толкая, пытаясь одновременно передать на шину какие-то данные.

Итак, мы знаем все типы пакетов, необходимые для обмена. Добавляем проверку принятого байта PID на соответствие каждому. В настоящий момент устройство еще не умеет писать в шину даже такие примитивные пакеты как ACK, а значит неспособно рассказать хосту что же оно такое. Поэтому и команд типа IN ожидать не приходится. Так что проверим только прием команд SETUP и OUT, для чего в соответствующих ветках пропишем включение соответствующих светодиодов.
Кроме того, стоит вынести посылку логов за пределы прерывания, куда-нибудь в main.
Прошиваем устройство тем, что получилось после внесения этих правок и наблюдаем следующую последовательность принятых байтов:

```
2D|80|06|00|01|00|00|40|00
C3|80|06|00|01|00|00|40|00
2D|80|06|00|01|00|00|40|00
C3|80|06|00|01|00|00|40|00
```

А кроме того — оба горящих светодиода. Значит, и SETUP и OUT мы поймали.

Шаг 8. Читаем адрес на конверте

Теоретическая вставка

Маркер-пакеты (SETUP, IN, OUT) служат не только для того, чтобы показать устройству, что от него хотят, но и чтобы адресоваться к конкретному устройству на шине и к конкретной конечной точке внутри него. Конечные точки нужны для того, чтобы функционально выделить конкретную подфункцию устройства. Они могут различаться по частоте опроса, скорости обмена и прочим параметрам. Скажем, если устройство представляется переходником USB-COM, его основная задача — принимать данные с шины и передавать их в порт (первая конечная точка) и принимать данные с порта и отдавать в шину (вторая). По смыслу эти точки предназначены для большого потока неструктурированных данных. Но помимо этого, время от времени устройство должно обмениваться с хостом состоянием управляющих линий (всякие RTS, DTR и прочие) и настройками обмена (скорость, проверки четности). А вот тут больших объемов данных не предполагается. Кроме того, удобно, когда служебная информация не смешивается с данными. Вот и получается, что для переходника USB-COM удобно использовать как минимум 3 конечные точки. На практике, конечно, бывает по-разному...

Не менее интересный вопрос, зачем устройству передается его адрес, ведь кроме него в данный конкретный порт все равно ничего не воткнуть. Это сделано для упрощения разработки USB-хабов. Они могут быть достаточно «тупыми» и

просто транслировать сигналы от хоста всем устройствам, не заботясь о сортировке. А уж устройство само разберется, обрабатывать пакет или проигнорировать.

Так вот, и адрес устройства, и адрес конечной точки содержатся в маркер-пакетах. Структура таких пакетов приведена ниже:

поле	SYNC	PID	addr							endpoint				CRC					EOP
Биты USB	0 - 7	8-15	0	1	2	3	4	5	6	0	1	2	3	0	1	2	3	4	0, 1
Принятые биты	-	0-7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	-

Из приведенной ранее схемы обмена видно, что сразу после маркер-пакета идет прием (если маркер-пакет содержал PID = SETUP или OUT) или передача (IN) пакета данных, за которым должно следовать подтверждение.

В глобальном смысле это означает, что мы должны от начала транзакции (маркер-пакет) и до самого подтверждения (Handshake) хранить:
- адрес устройства: если адрес не соответствует нашему, надо не мешать общаться другим, даже NAK не слать
- тип маркер-пакета: если это SETUP или OUT, идет прием, если IN — передача, причем
- адрес конечной точки. Пока что неактуально, ведь точка будет всего одна и не будет сильно отличаться от устройства в целом, но в общем случае все же надо знать, с какой именно точкой желает общаться хост

Первый параметр можно оформить как флажок «отвечать — не отвечать» и совместить со вторым. Для второго параметра корректными значениями являются только три PID`а, к которым мы можем добавить четвертый, обозначающий как раз флажок игнора. Красивым значением для такого «PID» будет ноль. Для хранения этого значения заведем переменную *usbCurrentTok*. А вот PID`ы начала данных (DATA0, DATA1) маркерами не являются, соответственно и сохранять их не будем. Отдельный вопрос что делать с данными, адресованными другому устройству? Варианта два: можно честно их считать, но потом проигнорировать (значение 0 в переменной *usbCurrentTok* позволит это сделать без проблем), либо даже не считывать, а просто подождать окончания транзакции. Выйти из прерывания до окончания транзакции (событие SE0) мы не можем в любом случае, поскольку хост-то данные передает, и уровни на линиях D+, D- скачут. Значит мы тут же попадем в прерывание снова, но уже не в начало байта SYNC, а неизвестно куда. Теоретически, можно было бы завести отдельный таймер на ожидание окончания обмена по неверному адресу, а после его срабатывания запустить прерывание ожидания начала обмена снова. Но на практике «тупые» хабы встречаются уже редко, и подавляющее большинство посылок будет адресовано именно нам. Да и занимать целый таймер под такое дело не хочется.

Адрес конечной точки имеет смысл только внутри прерывания, поэтому заводить под него отдельную переменную не имеет смысла. Вместо этого оставим его в регистре *x3*, по значению которого впоследствии выберем конкретный буфер для обмена (повторюсь, в нашей реализации это неактуально, поскольку конечная точка всего одна, но хоть от обращения к устройству в целом отделим).

Как уже упоминалось ранее, передача по USB осуществляется с младшего бита, но в пределах каждого отдельного поля, тогда как наша функция приема работает с байтами и знать не знает про поля. В результате во втором байте, помимо адреса, оказался младший бит конечной точки, а к остальным битам этой точки примешалась контрольная сумма CRC (которую мы все равно не будем считать). Чтобы правильно считать из пакета адрес, просто побитово сдвинем его влево [21]. При этом в бит переноса улетает 0-й бит адреса конечной точки. Чтобы его приклеить на законное место, воспользуемся циклическим сдвигом [26]. Правда, при этом в бездну улетает старший бит CRC, но мы все равно не собирались его использовать.

Шаг 9. Безотказный прием

Пришло время научиться не только принимать данные, но и передавать хосту отчет о приеме, пока что только «пакет принял», то есть ACK. Но для универсальности стоит предусмотреть также передачу и NAK`а, и любого другого байта (храниться он будет в *cnt* — здесь это не счетчик а просто буфер для данных). Поскольку в USB байты по одному не ходят, сначала сформируем буфер, а потом будем передавать его вместе со всеми обязательными байтами вроде SYNC и PID. Указателем на начало буфера назначим регистр *Y*, а количество элементов будем хранить в переменной с оригинальным названием *cnt* (не путать с передачей произвольного байта через тот же регистр). Это все далеко идущие планы, а пока надо передать всего один байт — ACK. Для этого положим его в регистр *x3* и сообщим функции что он — буфер размером в 1 байт, ведь к регистрам общего назначения можно обратиться как к обычной памяти. В нашем случае регистр *x3* (он же *r20*) расположен по адресу 20.

Поскольку адрес менять мы пока не умеем (это делается специальным запросом в SETUP, да еще и с подтверждением), но передавать ACK`и уже готовы, наступает немного опасный момент, когда устройство потенциально может мешать конфигурировать другие устройства, которым тоже пока не назначили адрес. Впрочем, к такому поведению хост готов и просто отключит наш порт после нескольких неудачных попыток.

Как мы помним, передача единицы осуществляется сохранением состояния линий D+, D- (что вообще не требует усилий от контроллера), а передача нуля — изменением. Проще всего это сделать операцией XOR по маске, чтобы не задеть остальные выводы порта, которые, скорее всего, будут использованы под какие-то другие задачи.

Передача байтов, так же как и прием, завязана на точный подсчет тактов, поэтому располагаться будет в части файла, зависящей от частоты. Но, по сравнению с приемом, во время передачи бита свободного времени чуть больше, так что нет нужды проверять добавочные нули на каждом бите независимо. Но и завернуть все в цикл не выйдет. Поэтому авторы vusb применили хитрый трюк: цикл **txBitloop** обрабатывает по 2 бита за раз ([00], [08]). Прокрутив этот цикл 3 раза, передается 6 бит. Остальные два бита передаются в линейном участке, чтобы можно было разместить код подготовки переменных к передаче следующего байта между строк. Чтобы одновременно уменьшить счетчик на 1 и прокрутить цикл 3 раза применен хитрый способ: от переменной счетчика отнимается число 171. Первые два раза результат получается отрицательным (если счетчик изначально меньше 171, а так как размер буфера не более 11 байт, это условие выполняется всегда), а на третий — неотрицательным, на единицу меньше исходного. Поясню на примере cnt=4:
4 - 171 = -167 = (сколько влезает в байт) 89 (+бит переноса)
89 - 171 = -82 = (сколько влезает в байт) 174 (+бит переноса)
174 - 171 = 3. Столько в байт и влезает, бит переноса не выставляется
то же самое можно доказать и математически, но здесь этого делать не будем.

Таким образом, цикл прокручивается 3 раза, после чего счетчик уменьшается на 1. Добавление добавочных нулей после каждых 6 идущих подряд единиц осуществляется не по маскам, как в случае приема, а просто подсчетом в регистре *x4*. После передачи всего буфера не забываем переключить D+, D- обратно на вход, восстановить биты прерывания и т. п.

После всех манипуляций наблюдаем зажигание зеленого светодиода и следующий вывод:

```
2D|80|06|00|01|00|00|40|00
69|00|10|00|01|00|00|40|00
```

Куда потерялся токен C3 неизвестно. Скорее всего, основной цикл просто не успевает его поймать, ведь весь этот вывод по UART штука довольно медленная. Впрочем, уже то, что к нам начал приходить токен IN означает, что хост подозревает устройство не совсем безнадежным. Значит, мы на верном пути.

Шаг 10. Посылаем хоста NAK

Сама функция передачи NAK у нас уже есть, но пока не было повода ее использовать. А поводом будет наличие принятых данных, которые мы пока не успели обработать. Это хост пусть думает что не успели, на самом-то деле пока что и пытаться не будем.

Но начнем не с посыла хоста, а с двойной буферизации на прием. Мы ведь не хотим, чтобы данные, которые мы только-только приняли и начали обрабатывать, внезапно затерлись другим пришедшим пакетом. Для этого воспользуемся тем же самым буфером *usbRxBuf*, но разделим его пополам. В одном случае будем писать с нулевого адреса, во втором — с середины, то есть по смещению USB_BUFSIZE. Выбор начала записи будет осуществляться по переменной *usbInputBufOffset*, которая принимает как раз эти два значения и прибавляется к адресу буфера. Переключение буфера придется добавить в начало функции приема.

Собственно посылка NAK осуществляется из метки **handleData** в случае если пришел новый пакет, а из предыдущего данные еще не забрали [22]. Факт окончания обработки данных будем отслеживать по равенству нулю счетчика длины (*usbRxLen*), обнулять который будем в Си-шном коде. А если данных пока не было (задел на будущее — если данные уже обработаны), выставляем количество принятых байт в переменную *usbRxLen*, а токен, в честь которого данные посланы — в *usbRxToken*, поскольку разделить SETUP и OUT все-таки надо.

Другая особая ситуация: хост может послать пакет с пустым полем данных, в котором обрабатывать вообще нечего, поэтому шлем ACK сразу.

Во всех же остальных случаях просто сохраняем принятый пакет в буфер и ждем пока основной цикл его обработает. Дело в том, что обработка пакета штука долгая, особенно если это не какой-то из стандартных пакетов запроса, а что-то, связанное с юзерской задачей.

Зачем же задерживать выполнение программы длительным нахождением в прерывании? Лучше дадим хосту знать, что пакет приняли, но ответа пока нет, пусть спросит попозже, а там авось Си-шный код придумает что ответить.

Как следствие, хост посылает запросы вроде

```
2D|80|06|00|01|00|00|40|00
```

Но устройство принимает только первый, отвечая на остальные NAK`ом, поскольку буфер уже занят, а чистить его пока некому.

Шаг 11. Обрабатываем запросы

С приемом и передачей битов, байтов, пакетов и даже транзакций по большому счету закончили. Пора переходить на более высокий уровень — анализировать запросы хоста и пытаться на них адекватно реагировать. Положительный момент заключается в том, что хост не требует немедленной реакции, так что мы можем перестать, наконец, мучить прерывание, и отдать обработку основному коду. Именно для этого мы выносили буфер и тому подобное в глобальные переменные. Теперь мы можем их читать из основного цикла. Вот только в цикле могут быть и другие задачи, помимо работы с USB, так что лучше сразу вынести эту задачу в отдельную функцию **usbPoll**. Главное — надо проверить пришли ли вообще данные, или это был пустой пакет. И если пришли — передать их в соответствующую функцию. Как мы помним на примере SETUP пакета, к данным примешиваются еще PID и CRC, но если в SETUP использовалась 5-битная контрольная сумма, то данные защищают аж 16-битной. В результате у нас целых 3 «мусорных» байта. «Мусорных» потому что PID мы уже сохранили в переменной *usbRxToken*, а CRC проверять не обязательно, вот и не будем отъедать ресурсы контроллера такими глупостями, как надежность. И просто передадим полезные данные в функцию **usbProcessRx**, попутно указав правильное начало буфера приема, с учетом двойной буферизации и игнорирования токена.

Второй момент, который можно обработать, это упомянутое в самом начале особое состояние — сброс шины, то есть длительное удерживание SE0. Если его обнаружили, надо сбросить состояние модуля USB как будто его выключали по питанию.

Вернемся к обработке принятого пакета. Пока что нас интересуют только пакеты типа SETUP, поскольку устройство еще не сумело рассказать хосту что же оно такое. Так что других пакетов ждать все равно приходится. Структура SETUP пакета описана в *usbRequest_t* и занимает всегда 8 байт. Запросы делятся на два основных класса: стандартные (которые должно обрабатывать любое USB-устройство) и нестандартные, то есть специфичные для каких-то конкретных. В перспективе отдадим обработку нестандартных запросов на реализацию юзеру, а сами будем обрабатывать стандартные. Но пока что просто будем выводить символы чего же нам прилетело.

А прилетает нам куча стандартных запросов, что, в принципе, не удивительно.

Шаг 12. подробности SETUP`а

Итак, мы выяснили, что хост посылает только стандартные запросы. Разберемся что считается стандартными запросами. Для этого напишем функцию **usbDriverSetup**, которая на основные известные запросы будет выводить соответствующие символы. Впрочем, некоторые запросы мы можем обработать сразу. Например, запрос или изменение текущей конфигурации (некоторые устройства поддерживают больше одной, но нам это не грозит, так что игнорируем и говорим хосту что у нас все конфигурации одинаковые) или установка адреса. Правда, словарный запас у нашего устройства все равно состоит из двух слов: ACK и NAK, так что сообщить хосту об успешности не выйдет.

Шаг 13. Отправляем ответ

Как было показано раньше, хост начинает общение с пакетов SETUP + DATAx, причем в пакете DATAx хранится всегда 8 байт. После этого хост делает запрос на чтение IN и ждет от устройства пакета DATAx, на который готов выслать подтверждение. До настоящего момента ответный пакет мы отправлять не умели, исправлением чего сейчас и займемся. Собственно функция отправки нескольких байт у нас есть, мы использовали ее чтобы послать ACK или NAK. Но посылка именно одного из них годится только для простых случаев, которые могут быть обнаружены ассемблерным куском кода. Самое главное — посылка всего буфера *usbTxBuf*, точнее не всего, а только *usbTxLen* байт из него. Стандартом low-speed USB задан максимальный размер буфера на прием и передачу в 8 байт (плюс PID, плюс два байта CRC), значит и переменная *usbTxLen* никак не может быть больше 11. Сравним это с допустимыми значениями PID, с учетом того, что старший полубайт является инверсией младшего. Получается, что возможных значений всего 16, из которых наименьшее, 0x0F, вообще считается зарезервированным. Так что значения PID с размером массива никак не спутать, чем и воспользуемся дабы не плодить переменные. Отправлять этот байт будем после приема пакета IN, как единственного места, где устройству вообще разрешено слать данные (handshake пакеты не в счет, они не для данных).

То есть обмен данными выглядит так:
Хост шлет SETUP + DATAx, на что получает ACK или NAK в зависимости от состояния буфера приема. Устройство, точнее, функция **usbPoll**, как дойдут руки, анализирует принятый пакет и заполняет буфер ответа (пока что все заполнение заключается в записи PID=DATA1 (чем отличаются DATA0 и DATA1 расскажу чуть позже, пока что методом тыка выясняем, что хосту нравится только DATA1). Два байта CRC не заполняем. Послать мы их обязаны в любом случае, но раз данных нет, писать туда что-то не имеет смысла. Ну и в конце концов задаем размер буфера для передачи — 4 байта. Обратите внимание, что формируем мы 3 байта, а посылаем 4. Дело в том, что байт SYNC также будет послан и учитывается при подсчете. Во время подготовки ответа хост может периодически попинывать нас «ты там уже IN или еще NAK?» на что следует ответ NAK. Но вот когда данные наконец готовы, ответ меняется на более осмысленный, DATA1 с содержимым буфера.

Попутно обработаем единственный интересующий нас запрос, не требующий развернутого ответа — USBRQ_SET_ADDRESS (остальные либо не интересны, либо требуют подробного ответа). При помощи него хост пытается назначить нам адрес на шине.

Но фактическая запись произойдет только при отправке пустого пакета (**drvsdm.S**, псевдометка **make SE0**). Раньше этого делать не стоило, поскольку хост бы не поверил что мы приняли адрес и готовы на него отзываться, а вот теперь, когда мы послали ему целый DATA1 с пустым полем данных, он понимает, что перед ним полноценное устройство и соглашается назначить ему адрес. Правда, он не понимает, почему вместо ответа на остальные запросы ему выдается не осмысленная информация, а такие же заглушки, так что делает несколько попыток выяснить класс устройства и прочие параметры, а после получения нереалистичных ответов признает устройство безнадежным. Исправлением этого недостатка и займемся в дальнейшем, а пока можно прошить устройство данным кодом и полюбоваться, как ему назначают адрес на шине.

Шаг 14. Сортируем стандартные запросы

Прежде чем писать ответы, проанализируем что же именно спрашивает у нас хост. Как мы помним, типичными запросами были USBRQ_GET_DESCRIPTOR и USBRQ_SET_ADDRESS, впрочем, второй мы уже обработали. Теперь напишем функцию **usbDriverDescriptor**, выясняющую интересующий хоста дескриптор. Находится интересующая нас информация в следующем байте запроса, после собственно USBRQ_GET_DESCRIPTOR. Основные дескрипторы, которые мы будем обрабатывать, это:
USBDESCR_DEVICE — общее описание девайса: тип протокола USB (1.1 в нашем случае), тип устройства, производитель, номер устройства и т. п.
USBDESCR_CONFIG — конфигурация, то есть типы конечных точек, потребляемый ток и т. п.
USBDESCR_STRING — строковые описания устройства, производителя и версии.

Прошиваем, запускаем и видим, что хост много раз пытается выяснить USBDESCR_DEVICE, терпит неудачу, после чего даже не пытается задать остальные вопросы.

Шаг 15. Заполняем анкетные данные

Здесь придется единоразово внести большое количество изменений. Во-первых, надо заполнить структуру описания устройства. Описывать все ее поля не буду, лучше возьмем какую-то более-менее стандартную, например, HID, благо для использования не потребуется драйверов, да и сам протокол обмена не слишком сложный. Среди прочего надо указать Vendor ID и Product ID, которые выдаются только разработчиками стандарта USB, причем за весьма солидную плату. К счастью, авторы vusb приобрели несколько штук и разрешили пользоваться всем желающим при соблюдении соответствующих условий.
Местом для хранения структуры возьмем, разумеется, флеш-память контроллера. Как было сказано выше, у одного устройства может быть несколько дескрипторов, плюс еще строковые значения, так что указатель именно на интересующий нас (точнее, хоста) кусок памяти запишем в переменную *usbMsgPtr*, а размер — в переменную *len*, которую впоследствии передадим в *usbMsgLen*. Размер дескриптора устройства у нас (да и у всех остальных устройств) составляет 18 байт, тогда как в буфер помещается только 8. Значит, будем передавать по частям, в 3 этапа. Если же во время чтения произошла какая-то ошибка, пошлем STALL.

Для чтения данных из флеша в буфер передачи используется функция **usbDeviceRead**. Помимо собственно чтения, с которым справилась бы и стандартная **memcpy_P**, мы обеспечиваем возможность расширения функционала до чтения из оперативной памяти и других источников, юзерского кода, например.

Еще стоит отметить, что теперь поле данных не пустое, а значит, и контрольную сумму придется считать честно. Эта операция достаточно стандартная, да и вызываться будет довольно часто, поэтому имеет смысл вынести ее в ассемблерный код.
Также важно не забыть передавать длину сообщения, которая раньше нас не интересовала, между функциями.

Теоретическая вставка

Отдельные PID`ы для DATA0 и DATA1 используются для дополнительной защиты от ошибок передачи. Эти PID`ы при нормальной передаче должны чередоваться, так что если встречается два нулевых или два первых подряд, значит что-то пошло не так и надо это исправлять.

Когда мы передавали единственный пустой пакет данных, чередование DATA0 / DATA1 нас не интересовало (как не интересовало и не будет интересовать при приеме), но теперь первая же посылка, дескриптор, занимает аж 3 пакета, значит придется добавить чередование. Для этого применяется операция XOR сначала с одним PID`ом, потом с другим.

Как можно понять из определения операции, ее двойное применение равнозначно возвращению в исходное состояние, причем несколько XOR`ов друг другу не мешают. Если первый пакет имел PID равный DATA1, то наложение XOR с тем же PID приведет к обнулению, а наложение XOR с DATA0 доведет изменение до конца.

Прошивка устройства этим кодом дает понять, что хоста ответ удовлетворил, и теперь он запрашивает USBDESCR_CONFIG.

Шаг 16. Наконец-то устройство!

Обработка запроса USBDESCR_CONFIG ничем принципиально не отличается от USBDESCR_DEVICE. Просто добавляем новый дескриптор (он уже менее стандартный, но описывать его я все равно не буду) и отсылаем. И наше устройство, наконец-то определяется системой именно как USB-устройство, а не просто непонятная кривулька, дрыгающая линиями D+, D-.

Раз уж добавление дескрипторов оказалось такой простой задачей, добавим еще и все строковые описания: устройства, производителя, версию и язык. Строки различаются по номерам, указанным в трех предпоследних байтах дескриптора устройства (нулевой номер соответствует списку языков, который не совсем строка). Стоит отметить, что для строк используется кодировка UTF-16, то есть каждый символ кодируется двумя байтами. Что помешало разработчикам стандарта USB использовать стандартную и удобную UTF-8 мне неизвестно.

В случае vusb наличие этих строк не просто приятная особенность, позволяющая просто по выводу lsusb определить что же подключено. Доступных свободно пар VID, PID у нас не так уж много, поэтому отличить устройство именно по ним не выйдет. А вот ограничений на формат строк описания мне неизвестно, поэтому искать нужное устройство будем по паре VID, PID, а потом уточнять наше — не наше именно по строковым описаниям.

Кстати, часть данных отправляется уже не через устройство в целом, а через первую конечную точку (других у нас и нет). Об этом говорит хост при передаче SETUP: как мы помним, он передает не только адрес устройства на шине, но и адрес конечной точки. Если это 0, то обращение идет к устройству в целом, а если нет — то к точке с соответствующим номером. Собственно, с момента запроса дескриптора конфигурации, общение в основном будет идти через точку, указанную там.

Вот теперь устройство отображается в системе как положено.

Шаг 17. Устройство становится человечнее (HID)

теоретическая вставка

HID — human interface device, специальный тип устройств, предназначенный прежде всего для взаимодействия с человеком, с учетом его технических ограничений. То есть для HID важнее скорость реакции, но не объем передаваемых данных или скорость. Кроме того, упор делался на стандартизацию стандартных устройств, чтобы при подключении, скажем, мышки, не пришлось устанавливать специальные драйвера от производителя. Ну и упрощение «мозга» устройств тоже было важным моментом. В результате при разработке стандарта HID решили пожертвовать скоростью обмена (для low-speed она составляет всего

лишь около 800 байт в секунду), но отсутствие необходимости в специальных драйверах компенсирует многое.

Для общения по протоколу HID придется завести еще один дескриптор, передавая его по запросу USBDESCR_HID_REPORT. Поскольку реализация конкретного устройства не привязана к ядру библиотеки vusb, ее стоит отдать на откуп пользователю. Так, для функции **usbDriverSetup** (стандартные запросы) появляется параллельная **usbFunctionSetup** (юзерские запросы). Кроме того, раньше мы использовали только пакеты SETUP, но теперь надо сделать отдельный обработчик для пакетов OUT. Системные данные через них не передаются, так что принятое будет отправлено напрямую в юзерский код, точнее, в функцию **usbFunctionWrite**.
Доработать функцию чтения тоже придется, дополнив **usbDeviceRead** альтернативой **usbFunctionRead**, которая вместо дескрипторов и тому подобного заполняет буфер передачи юзерскими данными. Чтобы код знал, какую из альтернатив запускать, в функции **usbFunctionSetup** взводится (или не взводится, если не нужен) специальный флажок USB_FLG_USE_USER_RW, а в **usbDriverSetu**p он безоговорочно сбрасывается.

Полезный функционал устройства — мигалка диодами и опрос кнопки — осуществляются именно в **usbFunctionWrite** и **usbFunctionRead**. Просто анализируем пришедшие данные и работаем с кнопкой и светодиодами. Единственная сложность — данные могут прийти в несколько этапов, так что их придется склеивать из кусочков.

Напоследок подчистим прочие стандартные запросы в **usbDriverSetup**.

Шаг 18. Общаемся с железкой

Устройство готово, но чтобы его проверить до конца, надо научиться посылать ему запросы и получать ответы. Поскольку устройство у нас HID, писать драйвер не придется, можно ограничиться программой, запускаемой с правами обычного пользователя (но вот права доступа к самому устройству в udev прописать все-таки придется). К сожалению, кроссплатформенной библиотеки под эту задачу я не нашел, пришлось писать свою. Подробно расписывать ее работу особого смысла не вижу, кому будет интересно, разберется по коду, либо воспользуется без подробного изучения.

Теперь устройство успешно распознается операционной системой и реагирует на посылаемые байты.

Шаг 19. Сравниваем с vusb

Чтобы получить оригинальный код vusb осталось разделить наши три файла на части по смыслу и густо обмазать механизмами переносимости, защиты и универсальности.
drvasm.S делится на частото-независимые **usbdrvasm.S** и **asmcommon.inc**, а также частото-зависимый, один на выбор, **usbdrvasm12.inc** — **usbdrvasm20.inc**.
main.c делится на собственно **main.c** (юзерский код) и **usbdrv.c** (ядро библиотеки vusb)
usbconfig.h тонким слоем размазывается между предыдущими (некоторые объявления нужны только конкретному файлу), но основная суть, настройка конфигурации, остается в **usbconfig.h**.

Заключение

В отличие от оригинальной библиотеки vusb, наша версия состоит из одного файла на Си и одного на Ассемблере, не считая констант, общих для обоих файлов. Кроме того, нельзя поменять частоту кварца, конечная точка всего одна и т. п. Впрочем, целью работы являлось не это, а пошаговое написание простого устройства, определяемого системой как USB-HID. Разделение на файлы, добавление других частот, коррекция частоты, добавление других конечных точек и прочие плюшки уже реализованы в оригинальной vusb, либо могут быть

допилены самостоятельно, благо теперь, надеюсь, назначение каждого элемента кода стало понятнее.

Использованная литература и полезные ссылки

https://www.obdev.at/products/vusb/index.html (оф сайт vusb)
http://microsin.net/programming/arm-working-with-usb/usb-in-a-nutshell-part1.html
Агуров П.В. **Интерфейсы USB: Практика использования и программирования**
https://radiohlam.ru/tag/usb/
http://we.easyelectronics.ru/electro-and-pc/usb-dlya-avr-chast-1-vvodnaya.html
http://usb.fober.net/cat/teoriya/