Практикум программиста USB-устройств Часть 3. Расширение функций ядра USB-устройства

Дмитрий Чекунов (г. Ижевск)

Наше ядро уже вполне состоялось как USB-устройство, но для придания ядру функциональной законченности нам ещё предстоит расширить его возможности в части самообслуживания.

Требования USB

Начнём расширение функций ядра с уже хорошо изученной нами области – обслуживания требований.

Однако на этот раз мы обратимся к дополнительным (vendor) требованиям. Зачем могут понадобиться требования данного типа? Прежде чем ответить на этот вопрос, давайте проанализируем, какова роль требований в работе USB-устройства.

Багаж теоретических знаний [1] характеризует требование как унифицированную команду, используемую для настройки устройства. Практический опыт [2] подтверждает теоретическую основу требований и подчеркивает их исключительность для функционирования устройства. Итак, требования играют роль команды управления, имеющей наивысший приоритет в обслуживании устройством. Следовательно, для выполнения USB-устройством некоторых безотлагательных действий целесообразно использовать дополнительное требование.

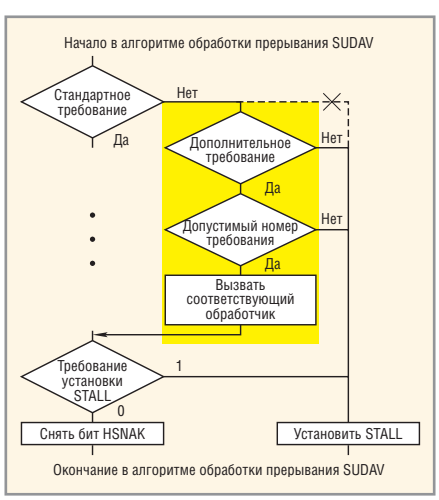

Рис. 1. Изменение алгоритма обработчика SUDAV для поддержки дополнительных требований

Включение поддержки дополнительных требований в проект

Анализ требования и определение его типа происходит в обработчике прерывания SUDAV [2, рис. 7]. До сих пор требования, не относящиеся к стандартным, мы считали недопустимыми и завершали контрольную транзакцию маркером STALL.

Теперь для того, чтобы наше ядро поддерживало дополнительные требования, изменим разработанный ранее алгоритм, и в разрыв ветви для нестандартных требований включим дополнительные действия по анализу и обслуживанию прочих допустимых требований (см. рис. 1). Хочу обратить внимание, что допустимыми могут быть и требования класса, но это уже определяется назначением USB-устройства.

Как видим, алгоритм изменился незначительно. Рассмотрим поподробнее его работу. В случае, если требование не относится к стандартным, мы проверяем его на принадлежность к дополнительным (биты D5, D6 поля bmRequestType). При отрицательном результате завершаем транзакцию маркером STALL, иначе переходим к анализу номера требования (поле bRequest). Номера для дополнительных требований программист волен выбирать по своему усмотрению (ограничений со стороны шины USB нет). Если номер полученного требования является допустимым, то выполняем соответствующую подпрограмму обработки и возвращаемся в основной алгоритм на анализ требования установки STALL.

Входные и выходные параметры обработчиков дополнительных требований абсолютно идентичны параметрам стандартных. Точно так же при обнаружении какой-либо ошибки необходимо установить требование flagStallEp0, для передачи дескриптора задать его адрес и установить признак flagGetDesc, а для передачи данных «ручным» способом заполнить буфер EP0BUF. Все прочие завершающие транзакцию действия будут выполнены обработчиком SUDAV.

Внесём в наш проект необходимые изменения.

Файл config.asm. К списку включенных файлов добавим ещё один:

```
$INCLUDE (ep0vr.asm)
```

Именно файл ep0vr.asm и ep0vrtable.asm, рассмотренный далее, будут обеспечивать поддержку дополнительных требований.

Файл ep0vrtable.asm: здесь будет храниться таблица векторов обработчиков требований (vrtable – Vendor Request TABLE) и константа – количество поддерживаемых требований:

```
DEAL_VENDOR_REQUEST          EQU       0
```

Формат таблицы векторов следующий:

```
; смещение 0
; lcall подпрограмма_требования_0 ; sjmp
endVendorRequestTable
; смещение 5
; lcall подпрограмма_требования_1
; sjmp endVendorRequestTable
```

```
; …
; смещение N*5
; lcall подпрограмма_требования_N
endVendorRequestTable:
```

Пока все записи в таблице закомментированы, поскольку количество требований равно нулю. Из данного примера хорошо видно, что вектора сведены в упорядоченные записи размером по 5 байт (3 байта – lcall, 2 байта – sjmp). При такой организации поиск подпрограммы обработки сводится к вычислению смещения (номер требования умножить на 5) и переходу на полученный адрес. Следовательно, при присвоении номеров требованиям необходимо придерживаться простого правила: номера начинаются с 0 и увеличиваются без разрывов.

Итак, любое изменение набора поддерживаемых требований начинается с редактирования файла ep0vrtable.asm, где в таблице указываются названия реальных подпрограмм обработчиков.

Файл ep0vr.asm: название файла, пожалуй, говорит само за себя (vr – Vendor Request). Здесь располагаются подпрограммы обслуживания дополнительных требований, именно на них указывают ссылки из таблицы векторов обработчиков.

Файл intusb.asm. Выполним доработку обработчика прерывания SUDAV в соответствии с алгоритмом, представленным на рис. 1. Вызовы подпрограмм обслуживания дополнительных требований зададим в неявном виде, что позволит в будущем легко изменять набор поддерживаемых требований, не возвращаясь при этом к тексту обработчика SUDAV. Итак, нам предстоит отредактировать следующий участок кода программы:

```
  jnz isrSudav_2
  mov a,usbBufSetup+bR
  cjne a,#DEAL_STANDARD_REQUEST,isrSudav_5
isrSudav_2:
```

Переход на ошибку заменяем переходом на анализ и запуск обработчиков стандартных требований.

```
  jz isrSudav_1
```

Проверяем тип требования – биты D6 и D5 для vendor-требований должны иметь, соответственно, значения 1 и 0. В случае несовпадения типа требования переходим на ошибку:

```
  cjne a,#40h,isrSudav_2
```

Далее следует анализ номера требования и при обнаружении ошибки аналогичный переход на установку требования flagStallEp0:

```
  ; считываем номер
  ; дополнительного требования
  mov a,usbBufSetup+bR
  cjne a,#DEAL_VENDOR_REQUEST,isrSudav_Vr
  sjmp isrSudav_2
isrSudav_Vr:
  jnc isrSudav_2
```

Если тип и номер требования являются допустимыми, то вычисляем смещение в таблице векторов до соответствующей записи и осуществляем переход на нее:

```
  mov dptr,#vendorRequestTable
  mov b,#5
  mul ab
  jmp @a+dptr
vendorRequestTable:
```

А вот и неявно заданные обработчики дополнительных требований:

```
  $INCLUDE        (ep0vrtab.asm)
```

После выполнения обработчиков переходим на анализ результатов завершения:

```
  ljmp isrSudav_4
```

На этом изменения в файле intusb.asm можно считать завершёнными. Далее начинается анализ и обработка стандартных требований:

```
isrSudav_1:
  ; считываем номер
  ; стандартного требования
  mov a,usbBufSetup+bR
  cjne a,#DEAL_STANDARD_REQUEST,
isrSudav_5
```

Действительно, созданная структура проекта и организация текста программы позволяют легко изменять список дополнительных требований без коррекций обработчика SUDAV.

Разработка дополнительного требования

Приступим к проектированию собственного дополнительного требования и обработчика для него.

Из обзора организации памяти FX2LP [3] известно, что во встроенном ОЗУ располагается основная часть системных ресурсов: регистры управления модулями, буферы FIFO и прочие служебные элементы. В таком случае весьма полезным будет требование, способное обеспечить обмен данными с любой ячейкой встроенного ОЗУ: это позволит контролировать текущее состояние ячейки и устанавливать новое. При помощи такого требования можно будет более подробно изучить принципы работы «внутренностей» FX2LP, также это пригодится при отладке программы устройства.

Назовём требование LOAD_DATA. В таблице 1 показаны форматы требования для обоих направлений передачи данных. Как можно заметить, отличие ограничено битом направления D7 в поле bmRequestType, а все остальные поля идентичны. Признаком дополнительного (vendor) требования являются биты D6 и D5, имеющие значения 1 и 0 соответственно. Для требования взят наименьший номер из допустимых, т.е. 0 (поле bRequest). Адрес интересующей нас ячейки задаётся в поле wValue. Поле wIndex не используется, поэтому имеет значение 0. Для увеличения эффективности предусмотрим возможность доступа сразу к нескольким ячейкам памяти МК. С этой целью будем использовать фазу данных. Объём передаваемых данных задаётся в поле wLength, и оно однозначно показывает количество обслуживаемых ячеек памяти. Поскольку размер буфера контрольной точки равен 64 байтам, ограничим размер фазы данных этим

Таблица 1. Формат требования LOAD_DATA

Требование	bmRequestType	bRequest	wValue	wIndex	wLength
Записать данные в ячейку ОЗУ FX2LP	40h	0	Начальный адрес	0	1...64 байт
Считать данные из ячейки ОЗУ FX2LP	0C0h	0	Начальный адрес	0	1...64 байт

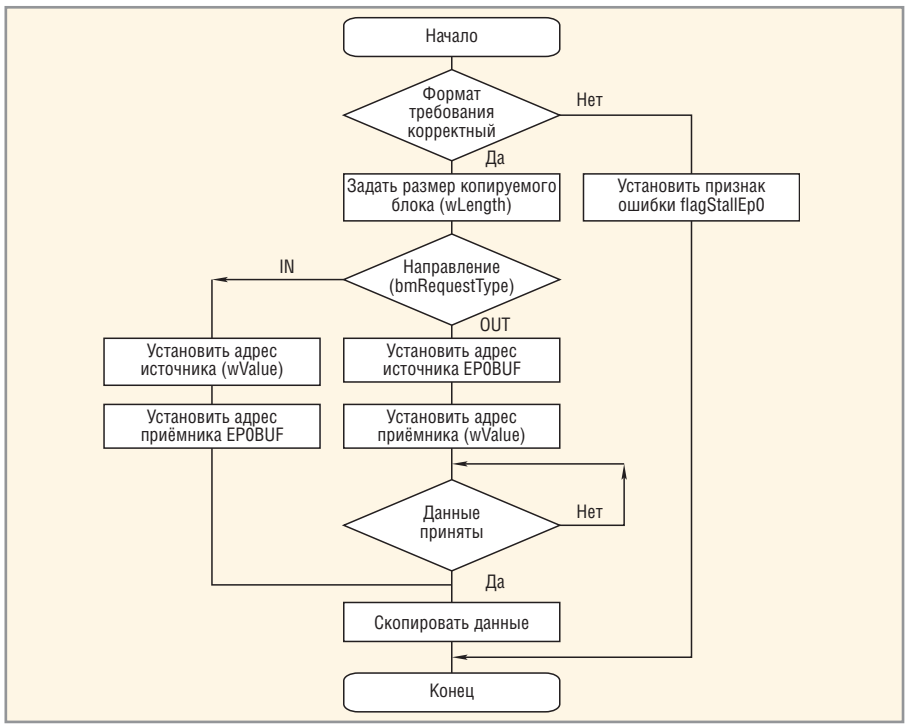

Рис. 2. Алгоритм обработки дополнительного требования LOAD_DATA

значением, чтобы весь требуемый объём можно было передать за один раз.

Теперь посмотрим, как будет функционировать подпрограмма обработки нашего требования LOAD_DATA. Входными параметрами для подпрограммы являются буфер usbBufSetup, флаг flagStallEp0, сброшенный в 0, и при направлении передачи OUT буфер EP0BUF. К выходным параметрам относятся тот же флаг flagStallEp0, установленный в случае ошибки, данные в буфере EP0BUF для передачи с направлением IN. Ключевое действие в подпрограмме – это копирование данных.

На рис. 2 представлен алгоритм обработки требования LOAD_DATA. Рассмотрим его подробнее.

Как обычно, на первом этапе проверяем корректность формата требования. При отсутствии ошибок определяем размер обслуживаемой области, заданный в поле wLength. Далее анализируем направление передачи данных (бит D7 поля bmRequestType). На ветви IN (передача данных хосту) задаём адрес источника встроенного ОЗУ МК (поле wValue) и адрес приёмника (буфер EP0BUF). На ветви OUT (приём данных от хоста) задаём адреса источника (буфер EP0BUF), приёмника – ячейки встроенного ОЗУ (поле wValue) и ждём поступления данных в буфер контрольной точки. Поскольку теперь у нас определены источник, приёмник и размер обслуживаемой области, выполняем копирование данных и на этом завершаем работу.

Далее обработчик SUDAV самостоятельно завершит фазу данных для направления IN записью размера пакета в регистры EP0BCH, EP0BCL. Фаза Setup при отсутствии ошибок будет закончена также самостоятельно.

Теперь приступим к практической реализации поддержки требования LOAD_DATA в составе нашего проекта.

Файл ep0vrtable.asm. Изменим количество поддерживаемых требований:

```
DEAL_VENDOR_REQUEST          EQU       1
```

Внесём первую запись в таблицу векторов:

```
lcall vrLoadData
; sjmp endVendorRequestTable
endVendorRequestTable:
```

Оставим команду перехода закомментированной, поскольку потребность в ней возникнет при добавлении следующих требований.

Файл ep0vr.asm: по алгоритму, представленному на рис. 2, напишем обработчик vrLoadData.

Файл util.asm: здесь потребуется добавить простую подпрограмму копирования данных между ячейками внешнего ОЗУ с синхронизационной задержкой. Назовем её moveExt2ExtSynch. Синхронизационная задержка нужна при последовательном обращении к регистрам FX2LP [4], расположенным в разных областях. Поскольку требование может обратиться к любой области, мы и вводим использование задержки.

Выполняем трансляцию программы и, если нет ошибок, переходим к проверке новой функции ядра.

Проверка работоспособности дополнительного требования

Для проверки работоспособности требования воспользуемся уже хорошо знакомой нам программой CyConsole. Подключим ядро к шине USB и загрузим исполняемый файл (mydevice.hex) в ОЗУ микроконтроллера. После переподключения ядра можно приступить к передаче требования устройству.

В списке устройств выделим MY USB-DEVICE и перейдём на закладку Control Endpt Xfers. Для начала считаем значение какого-нибудь регистра, например, регистра управления и статуса точки EP1IN – EP1INCS. В соответствии с форматом требования, представленным в таблице 1, корректно установим элементы управления на форме:

- Direction – выбираем Host ← Device;
- Request Type – выбираем Vendor;
- Recipient – выбираем Device (поскольку в поле bmRequestType нашего требования биты D4 – D0 сброшены в 0);
- Request Code – вводим 0;
- wValue – задаём 0xE6A2 (адрес регистра EP1INCS);
- wIndex – вводим 0;
- Bytes of Data – задаём 1 (данное поле называется в пакете Setup – wLength, и в нашем требовании значение этого поля показывает количество обслуживаемых ячеек памяти). Сейчас нас интересует значение только в одной ячейке.

Нажимаем <Transfer Data> и в поле сообщений видим результат завершения операции и значение, считанное из регистра EP1INCS (см. рис. 3). По считанному значению можно сделать вывод, что буфер точки EP1IN находится под управлением МК и со-

стояние STALL для точки не установлено.

Теперь можно попробовать послать требование с недопустимым номером. Введём в поле Request Code любое значение, отличное от нуля, и нажмём кнопку <Transfer Data>. В результате получим сообщение об ошибке.

Итак, в работоспособности требования мы убедились, и немного позднее мы используем его для экспериментов над FX2LP.

«Малые» точки FX2LP

Во всех МК семейства FX2LP имеются так называемые «малые» точки – это EP1OUT (адрес 1) и EP1IN (адрес 0x81). Такое название они получили потому, что принадлежащие им буферы имеют маленький размер (всего 64 байта) и не буферизированы. В связи с этим обстоятельством названные точки не применяются для передачи больших объёмов данных и поддерживают только два типа передачи данных – bulk и interrupt.

Ресурсы для управления «малыми» точками

В таблице 2 представлены ресурсы МК, используемые для взаимодействия с точками EP1OUT и EP1IN:

- EP1xxxCFG – регистр конфигурации. Регистр определяет работоспособность точки (бит VALID) и тип передачи данных (биты TYPE0 и TYPE1). Значение в данном регистре должно соответствовать состоянию точки, описанному в дескрипторе конфигурации для текущей альтернативной установки. То есть если точка в текущем состоянии устройства должна быть выключена, то это значит, что бит VALID будет иметь состояние 0. При изменении состояния устройства, когда, например, точка должна быть работоспособной с типом передачи interrupt, биты VALID, TYPE0 и TYPE1 необходимо установить в 1. В нашем проекте синхронное изменение значений регистров конфигурации осуществляется подпрограммой setConfigEp, которая копирует новые значения регистров из таблицы tableCfg1If0 [2];
- EP1xxxCS – регистр контроля и статуса. Регистр позволяет управлять состоянием точки (бит STALL) и контролировать готовность буфера к обмену данными (бит BUSY):
  - бит STALL напрямую связан с маркером подтверждения STALL, поэтому, когда данный бит установлен, точка с позиции хоста переходит в состояние HALT, и дальнейший обмен с ней невозможен. В этом случае на все запросы хост будет получать маркер подтверждения STALL;
  - бит BUSY отражает текущее состояние буфера точки. Если BUSY равен единице, то буфер занят, в противном случае буфер свободен;
- EP1xxxBUF – собственно это и есть буфер точки;
- EP1xxxBC – счётчик байтов. Для точки EP1OUT он показывает размер пакета, принятого и размещённого в буфере EP1OUTBUF. При работе с точкой EP1IN в этот счётчик необходимо записать размер пакета, помещённого в буфер EP1INBUF;
- EPIRQ – регистр запросов прерываний;
- EPIE – регистр разрешения прерываний;
- IBNIRQ – регистр запросов прерываний для точек с направлением IN. IBN – это аббревиатура от сочетания слов IN-BULK-NAK, что обозначает: на запрос хоста IN точка с типом передачи BULK отвечает NAK, т.е. данные не готовы и запрос следует повторить позднее;
- IBNIE – регистр разрешения прерываний IBN;
- NAKIE – регистр разрешения прерываний для событий, формируемых точками OUT, однако в нём имеется бит IBN, с помощью которого можно запретить сразу все прерывания IBN.

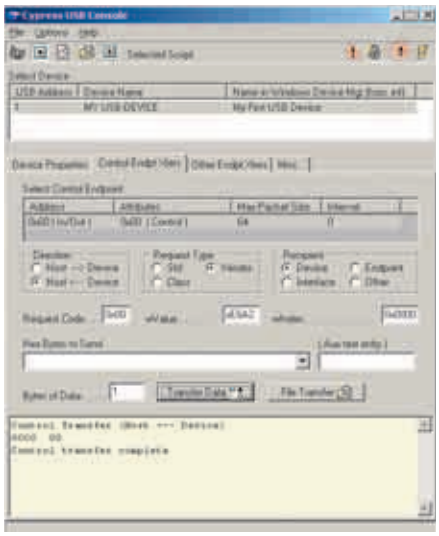

Рис. 3. Проверка работоспособности дополнительного требования

Взаимодействие точек с хостом

С типом передачи bulk мы уже знакомы [5]. «Малые» точки в нашем проекте будут использовать именно этот тип передачи, поскольку нам предстоит организовать достоверный обмен данными. Поэтому сейчас мы посмотрим, как же реально происходит взаимодействие между «малыми» точками и хостом. Начнём с EP1OUT, потому что логика работы точки с направлением OUT проще для понимания.

Всё начинается с того, что хост передаёт пакет данных, а USB-модуль помещает его в буфер EP1OUTBUF. Далее USB-модуль записывает размер полученного пакета в регистр EP1OUTBC и устанавливает флаг BUSY в регистре EP1OUTCS. Одновременно происходит установка запроса прерывания EP1OUT в регистре EPIRQ. Если в регистре EPIE имеется разрешение на прерывание от точки EP1OUT, то последует переход на обработчик прерывания isrEp1Out,

Таблица 2. Ресурсы FX2LP для взаимодействия с «малыми» точками

Ресурсы	Точка EP1OUT	Точка EP1IN
Регистр конфигурации	EP1OUTCFG	EP1INCFG
Регистр управления и статуса	EP1OUTCS	EP1INCS
Буфер размером 64 байта	EP1OUTBUF	EP1INBUF
Регистр размера пакета	EP1OUTBC	EP1INBC
Регистр запроса прерывания при изменении состояния буфера	EPIRQ (буфер заполнен)	EPIRQ (буфер пуст)
Регистр разрешения прерывания при изменении состояния буфера	EPIE	EPIE
Регистр запроса прерывания при запросе хоста и пустом буфере	–	IBNIRQ
Регистр разрешения прерывания при запросе хоста и пустом буфере	–	IBNIE
Общее разрешение прерывания для событий IBN	–	NAKIE

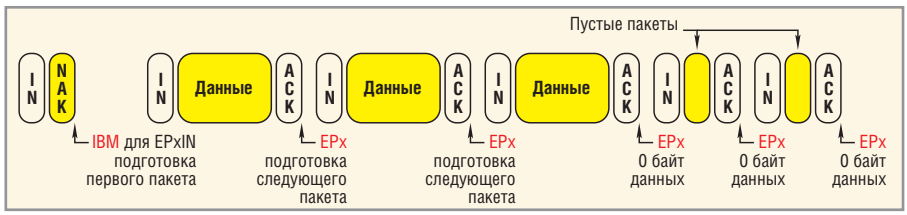

Рис. 4. Взаимодействие хоста и типовой точки IN

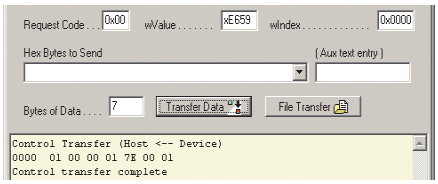

Рис. 5. LOAD_DATA: чтение регистров прерываний

иначе FX2LP должен самостоятельно обнаружить изменение состояния бита BUSY.

Теперь принятые данные необходимо обработать… МК по значению в EP1OUTBC узнаёт размер полученного пакета и приступает к выборке и обслуживанию данных из EP1OUTBUF. Флаг BUSY по-прежнему установлен в 1, поскольку МК ещё не сообщил об освобождении буфера.

В это время хост посылает второй пакет данных. Модуль USB, обнаружив, что буфер ещё занят, отвечает хосту маркером подтверждения NAK. В результате этого хост предпримет повторную передачу пакета позднее.

Наконец МК закончил обработку данных. Теперь, для того чтобы освободить буфер EP1OUTBUF, ему достаточно записать любое число в регистр EP1OUTBC, и после этого бит BUSY будет аппаратно сброшен в 0, а буфер EP1OUTBUF перейдёт под управление USB-модуля. Следующий пакет, переданный хостом, уже достигнет адресата.

Теперь рассмотрим логику работы точки EP1IN. На рис. 4 показана последовательность запросов и ответов для любой точки с направлением IN. Напомню ещё раз, что ни одно USB-устройство не может самостоятельно послать хосту данные. Любая передача осуществляется только по запросу хоста.

Итак, изначально буфер EP1INBUF пуст, о чём свидетельствует сброшенный в 0 бит BUSY. Казалось бы, если буфер пуст, то можно заполнить его данными. Однако это будет неправильно, потому что неизвестно, когда хост запросит данные, запросит ли вообще и будут ли они на тот момент ещё достоверными. Поэтому подготовка данных для точки с направлением IN начинается с момента реального обращения хоста.

При запросе хоста (IN) модуль USB проверяет состояние буфера EP1INBUF и, поскольку он пуст, возвращает маркер подтверждения NAK (данные не готовы) и формирует прерывание IBN (IN-BULK-NAK).

Теперь, после действительного запроса хоста, МК должен немедленно приступить к подготовке данных. После размещения данных в буфере EP1INBUF МК записывает размер помещённого пакета в регистр EP1INBC. Данное действие переводит буфер под управление модуля USB, и сразу же происходит установка флага BUSY – буфер занят.

При следующем запросе IN модуль USB передаёт данные из буфера хосту и, получив подтверждение ACK, сбрасывает флаг BUSY и возвращает EP1INBUF микроконтроллеру. Одновременно происходит установка запроса прерывания EP1IN в регистре EPIRQ, и, если в регистре EPIE имеется разрешение на прерывание от точки EP1IN, то последует переход на обработчик прерывания isrEp1In. Однако если прерывание не разрешено, то FX2LP должен самостоятельно контролировать состояние бита BUSY.

Итак, хост успешно забрал подготовленный пакет, и теперь буфер снова свободен. МК начинает готовить следующий пакет данных. Размещение нового пакета и передача буфера модулю USB будут соответствовать рассмотренным выше.

А теперь подумаем, какой объём данных может передать точка EP1IN. В принципе – бесконечно большой. Но в действительности, если, например, точка обслуживает микросхему памяти объёмом 16 Кб, то после передачи этого объёма транзакция должна завершиться. Как же сообщить хосту об отсутствии данных? Оказывается, очень просто. Если данные для передачи закончились, то необходимо записать в регистр EP1INBC значение 0, и хост получит пустой пакет данных (размер 0 байт). Подобный метод и позволяет точке сообщить хосту о завершении транзакции.

Внешнее управление точкой EP1IN

Теперь, после знакомства с «малыми» точками, воспользуемся нашим требованием LOAD_DATA и, манипулируя изученными ресурсами, попытаемся осуществить транзакцию с точкой EP1IN.

Ранее мы уже считали значение из регистра EP1INCS (см. рис. 3) и теперь знаем, что биты BUSY и STALL исследуемой точки сброшены в 0. Это говорит о том, что буфер EP1INBUF принадлежит МК и данные для хоста не готовы.

Перед тем как приступить к активным действиям, считаем значение регистров IBNIRQ и EPIRQ. Это позволит лучше понять логику работы точки IN. Изменим следующие элементы управления на закладке Control Endpt Xfers формы:

- wValue – задаём 0xE659 (адрес регистра IBNIRQ);
- Bytes of Data – задаём 7 (чтобы за один раз считать значение интересующих нас регистров).

Нажимаем <Transfer Data>, и в поле сообщений (см. рис. 5) видим результаты завершения операции. По значению в регистре IBNIRQ (01) можно сделать вывод, что запросов IN к исследуемой точке не было. Прерываний, формируемых битом BUSY, также не было, поскольку значение регистра EPIRQ также равно 01.

Перейдём на закладку Other Endpt Xfers и попытаемся принять данные из точки 0x81 (In). Обратите внимание, что от момента нажатия кнопки <Transfer Data> до появления сообщения пройдёт довольно значительное время.

В результате мы получим сообщение об ошибке (см. рис. 6). Это вполне логично, поскольку буфер EP1INBUF ещё пуст и не передан под управление модулю USB.

Вернёмся на закладку Control Endpt Xfers и повторим чтение регистров (см. рис. 7). Как видим, состояние битов регистра IBNIRQ изменилось – произошла установка бита EP1IN. Это

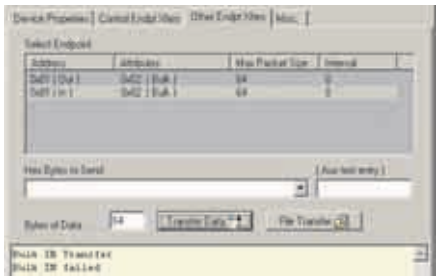

Рис. 6. Тайм-аут при чтении из точки EP1IN

Рис. 8. LOAD_DATA: запись данных в буфер EP1INBUF

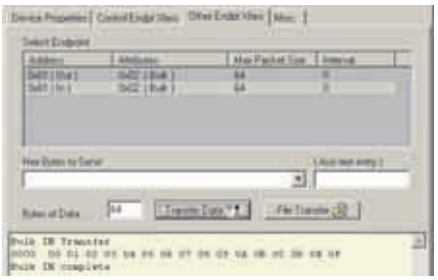

Рис. 10. Успешное чтение из точки EP1IN

Рис. 7. LOAD_DATA: контроль изменений в регистре прерываний IBNIRQ

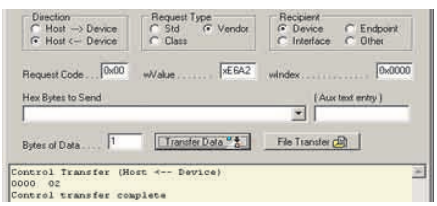

Рис. 9. LOAD_DATA: контроль бита BUSY в EP1INCS

Рис. 11. LOAD_DATA: контроль изменений в регистре прерываний EPIRQ

означает способность МК «понять», что хост требует данные, в связи с чем необходимо срочно заполнить буфер.

Для заполнения буфера изменим следующие элементы на форме:

- Direction – выбираем «Host → Device»;
- wValue – задаём 0xE7C0 (адрес буфера EP1INBUF);
- Hex Bytes to Send – через пробел вводим данные для записи в буфер точки;
- Bytes of Data – программа автоматически устанавливает значение для этого поля в соответствии с распознанными данными в поле Hex Bytes to Send.

Нажимаем <Transfer Data> и в поле сообщений (см. рис. 8) видим результаты завершения операции – данные переданы успешно. Сейчас мы только записали информацию в буфер, а для того, чтобы эти данные стали доступны хосту, необходимо передать буфер под управление модулю USB. Делается это просто. Записываем количество байт в буфере в регистр EP1INBC:

- wValue – задаём 0xE68F (адрес регистра EP1INBC);
- Hex Bytes to Send – задаём 10, поскольку вводимые здесь числа распознаются как шестнадцатеричные;
- Bytes of Data – программа автоматически устанавливает значение 1.

Нажимаем <Transfer Data> и видим, что данные переданы успешно.

Вот теперь буфер перешёл под управление модуля USB, и данные уже можно получить из точки EP1IN. Но перед этим считаем значение регистра EP1INCS:

- Direction – выбираем «Host ← Device»;
- wValue – задаём 0xE6A2 (адрес регистра EP1INCS);
- Bytes of Data – задаём 1.

Нажимаем <Transfer Data> и из считанного значения видим, что бит BUSY действительно установлен в 1 (см. рис. 9). Таким образом, наши утверждения верны – буфер находится под управлением модуля USB, и теперь можно выполнить чтение данных из точки EP1IN.

Переходим на закладку Other Endpt Xfers и пытаемся повторно принять данные из точки 0x81 (In). На этот раз операция завершилась успешно (см. рис. 10), а количество и значения принятых данных совпадают с заданными нами ранее.

Снова вернёмся на закладку Control Endpt Xfers и повторим чтение регистра EP1INCS. Видим, что бит BUSY принял значение 0, то есть буфер свободен и готов к новому заполнению данными микроконтроллером. Помним, что при сбросе бита BUSY для точки EP1IN должен сформироваться запрос на прерывание.

Считаем значение регистров прерывания:

- wValue – задаём 0xE659 (адрес регистра IBNIRQ);
- Bytes of Data – задаём 7.

Нажимаем <Transfer Data> и в поле сообщений (см. рис. 11) видим результаты завершения операции. Значение в регистре EPIRQ (07) показывает, что установлен запрос на прерывание для точки EP1IN.

Вот мы и организовали обмен данными с точкой, используя требование LOAD_DATA.

Давайте также посмотрим, как хост отреагирует на переход точки в состояние HALT. Для этого установим бит STALL в регистре EP1INCS:

- Direction – выбираем «Host → Device»;
- wValue – задаём 0xE6A2 (адрес регистра EP1INCS);
- Hex Bytes to Send – задаём 1;
- Bytes of Data – программа автоматически устанавливает значение 1.

После нажатия <Transfer Data> точка должна перейти в состояние HALT, то есть на любой запрос выдавать маркер подтверждения STALL – «точка неработоспособна».

Возвращаемся на закладку Other Endpt Xfers и пытаемся ещё раз принять данные из точки 0x81 (In). На этот раз операция завершается с ошибкой, но, в отличие от случая с пустым буфером, очень быстро. Из этого можно сделать вывод, что хост, обнаружив неработоспособность точки, не предпринимает к ней повторных обращений.

Окончание следует

Литература

1. *Чекунов Д.* Стандартные требования USB. Современная электроника. 2004. № 2.
2. *Чекунов Д.* Разработка аппаратно-программного ядра USB-устройства. Современная электроника. 2005. № 5, 6.
3. *Чекунов Д.* EZ-USB FX2LP – универсальное USB-решение. Современная электроника. 2005. № 4.
4. EZ-USB FX2 Technical Reference Manual. www.cypress.com.
5. *Чекунов Д.* Знакомство с USB. Современная электроника. 2004. № 1.