Практикум программиста USB-устройств Часть 2. Разработка аппаратно-программного ядра USB-устройства

Окончание. Начало см. СЭ № 5, 2005

Дмитрий Чекунов (г. Ижевск)

В заключительной части статьи описываются действия, выполняемые в стандартных требованиях, создание дескрипторов устройства, реальные последовательности команд, используемые при инициализации нового устройства на шине и, наконец, включение разрабатываемого ядра USB-устройства и его обнаружение в среде ОС Windows.

Создание дескрипторов устройства

Форматы дескрипторов нам уже известны [5], но для того чтобы приступить к их созданию, необходимо знать допустимые режимы работы USB-устройства и его топологию.

FX2LP поддерживает два режима работы – полноскоростной и высокоскоростной. При получении возможности работать в высокоскоростном режиме модуль USB вырабатывает прерывание HISPEED. Следовательно, изначально ряд дескрипторов должен быть рассчитан на полноскоростную шину, а при появлении этого прерывания будем динамически менять некоторые поля, подстраивая дескриптор для высокоскоростного режима работы. Динамическому редактированию могут подвергнуться следующие поля дескрипторов:

- дескриптор устройства. Поле bcdUSB содержит номер спецификации USB, поддерживаемой устройством. В полноскоростном режиме значение должно быть «1.1», в высокоскоростном – «2.0». Если подключить устройство к полноскоростной шине при значении «2.0», то Windows, например, выдаст предупреждение о подключении высокоскоростного устройства к медленной шине;
- дескриптор точки:
  – поле wMaxPacketSize – максимальный размер пакета данных. Для каждого типа передачи размер может изменяться в зависимости от режима работы в соответствии с таблицей 2;
  – поле bInterval – характеризует интервал времени, через который может произойти обмен данными для точек с типом передачи interrupt и «изохронный». Значение поля вычисляется в зависимости от режима работы.

Таблица 2. Зависимость размера пакета данных от режима работы устройства

Тип передачи	Full Speed USB 1.1	High Speed USB 2.0
Control	8, 16, 32, 64	64
Bulk	64	512
Interrupt	1–64	1–1024
Isochronous	1–1023	1–1024

Теперь определимся с топологией USB-устройства. На данном этапе развития нашего проекта выполнение каких-либо полезных функций устройством ещё не заложено. Поэтому для разрабатываемого ядра спроектируем простую топологию с возможностью её легкого расширения в перспективе.

Итак, основой будет служить топология устройства, представленная на рис. 8. В ней имеется всего одна конфигурация, один интерфейс и одна альтернативная установка с двумя точками: ep1 и ep81. Эти точки задействуем для обслуживания загрузочной микросхемы памяти (AT24C128) и выберем для них тип передачи bulk.

Передачу дескрипторов будем осуществлять с помощью системы SDP, поэтому необходимо, чтобы адрес начала любого дескриптора был чётным. Для выравнивания адресов используем следующие директивы:

```
IF $ MOD 2 = 1
DB 0
ENDIF
```

Теперь можно приступить к подготовке дескрипторов. Все дальнейшие действия выполняем в файле ep0sd.asm. Объявляем идентификаторы VID, PID, DID константами:

```
VID     EQU 3112h
PID     EQU 1973h
DID     EQU 0001h
```

При описании устройства определяем метку поля bcdUSB для динамического редактирования при изменении рабочей скорости, и назовём эту метку speedDevice:

```
IF $ MOD 2 = 1
DB 0
ENDIF
dscrDevice:
  DB 18 ; длина дескриптора
  DB 1 ; тип свойства -
  устройство
  speedDevice:
  DB 1,1 ; версия USB
  DB 0FFh ; класс устройства
  DB 0FFh ; подкласс устройства
  DB 0FFh ; протокол устройства
  DB 64 ; максимальная длина
  пакета для EP0
  DB LOW(VID),HIGH(VID) ; Vendor
  id
  DB LOW(PID),HIGH(PID) ; Product
  id
```

```
  DB LOW(DID),HIGH(DID) ; Device
  release id
  DB 1 ; индекс строки
  производителя
  DB 2 ; индекс строки названия
  DB 3 ; индекс строки серийного
  номера
  DB 1 ; количество конфигураций
  в устройстве
```

Поскольку наше устройство поддерживает высокоскоростной режим работы, оно должно иметь описание устройства для другой скорости:

```
dscrDeviceQualifier:
  DB 10 ; длина дескриптора
  DB 6 ; тип свойства -
  устройство на другой скорости
  DB 1,1 ; версия USB
  DB 0FFh ; класс устройства
  DB 0FFh ; подкласс устройства
  DB 0FFh ; протокол устройства
  DB 64 ; максимальный размер
  пакета
  DB 1 ; количество конфигураций
  DB 0 ; зарезервировано
```

Далее описываем конфигурацию. При этом помним, что она включает в себя дескрипторы всех интерфейсов и точек, поэтому необходимо в полном размере конфигурации указывать общую длину всех этих дескрипторов. При работе устройство запитано от внутреннего источника питания шины USB и потребляет 80 мА.

```
dscrCfg1: ; начало описания
конфигурации
  DB 9 ; длина дескриптора
  DB 2 ; тип свойства -
  конфигурация
  ; общая длина описания конфигу-
  рации, интерфейса и точек
  DB LOW(endDscrCfg1 - dscrCfg1)
  DB HIGH(endDscrCfg1 - dscrCfg1)
  DB 1 ; количество интерфейсов
  в конфигурации
  DB 1 ; значение для установки
  конфигурации
  DB 0 ; индекс строки
  наименования конфигурации
  DB 10000000b ; атрибуты
  конфигурации
  DB 40 ; максимальное
  потребление 80 мА (указываем
  потребление/2)

dscrCfg1If0Alt0:
  DB 9 ; длина дескриптора
  DB 4 ; тип свойства -
  интерфейс
  DB 0 ; база для интерфейса
  DB 0 ; альтернативное значение
  DB 2 ; количество точек
  DB 0FFh ; класс интерфейса
  DB 0FFh ; подкласс интерфейса
  DB 0FFh ; протокол интерфейса
  DB 0 ; индекс строки
  наименования интерфейса
```

В дескрипторах точек определяем метки для динамического редактирования максимальной длины пакета:

```
dscrCfg1If0Alt0Ep1out:
  DB 7 ; длина дескриптора
  DB 5 ; тип свойства - точка
  DB 1 ; адрес точки
  DB 2 ; тип передачи bulk
  sizeAlt0Ep1out:
  DB 64,0 ; максимальная длина
  пакета
  DB 0 ; интервал для EP_ISO

dscrCfg1If0Alt0Ep1in:
  DB 7 ; длина дескриптора
  DB 5 ; тип свойства - точка
  DB 81h ; адрес точки
  DB 2 ; тип передачи bulk
  sizeAlt0Ep1in:
  DB 64,0 ; максимальная длина
  пакета
  DB 0 ; интервал для EP_ISO
  endDscrCfg1:
; конец описания конфигурации
```

При поддержке высокоскоростного режима устройство должно иметь дескриптор конфигурации для другой скорости. Зададим количество интерфейсов в этом дескрипторе равным нулю, чтобы не занимать лишнее место в памяти для описания интерфейса и точек. Практически данный дескриптор не несёт какой-либо полезной информации, потому что в процессе работы хост не меняет режим работы устройства.

```
dscrOtherCfg:
  DB 9 ; длина дескриптора
  DB 7 ; тип свойства - конфигу-
  рация для другой скорости
  DB LOW(endDscrOtherCfg -
  dscrOtherCfg)
  DB HIGH(endDscrOtherCfg -
  dscrOtherCfg)
  DB 0 ; количество интерфейсов
  в конфигурации
  DB 1 ; значение для установки
  конфигурации
```

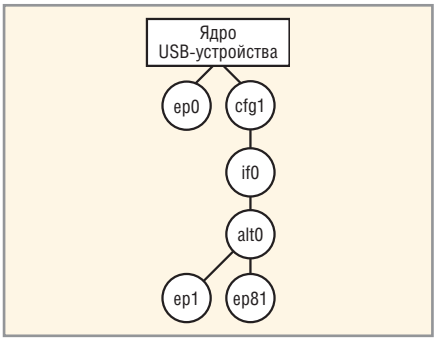

Рис. 8. Топология ядра USB-устройства

```
  DB 0 ; индекс строки
  наименования конфигурации
  DB 10000000b ; атрибуты
  конфигурации
  DB 40 ; максимальное
  потребление 80 mA
endDscrOtherCfg:
```

Далее включаем файлы с закодированными строками. Первым следует дескриптор строки с идентификаторами поддерживаемых языков:

```
dscrString0:
  DB endDscrString0 - dscrString0
  DB 3 ; тип свойства - строка
  DB 9,4 ; Идентификатор языка
  English(US)
  DB 19h,4 ; Идентификатор языка
  Russian()
endDscrString0:

dscrString1Us:
  DB endDscrString1Us -
  dscrString1Us
  DB 3 ; тип свойства - строка
  $INCLUDE(str1us.asm)
endDscrString1Us:

dscrString1Ru:
  DB endDscrString1Ru -
  dscrString1Ru
  DB 3 ; тип свойства - строка
  $INCLUDE(str1rus.asm)
endDscrString1Ru:

dscrString2Us:
  DB endDscrString2Us -
  dscrString2Us
  DB 3 ; тип свойства - строка
  $INCLUDE(str2us.asm)
endDscrString2Us:

dscrString2Ru:
  DB endDscrString2Ru -
  dscrString2Ru
  DB 3 ; тип свойства - строка
  $INCLUDE(str2rus.asm)
endDscrString2Ru:
```

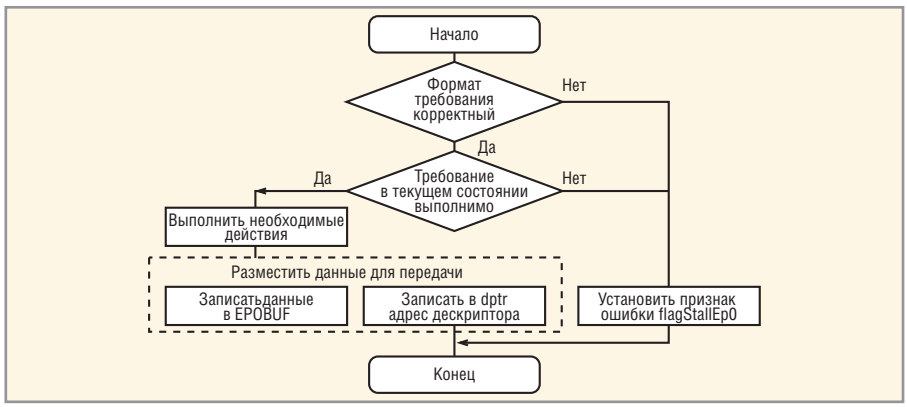

Рис. 9. Логика работы подпрограммы обслуживания требования

```
dscrString3:
  DB endDscrString3 - dscrString3
  DB 3 ; тип свойства - строка
  $INCLUDE(str3.asm)
endDscrString3:
```

Для упрощения поиска адреса нужного строкового дескриптора создадим две таблицы для русского и английского языка. Теперь по индексу строки мы можем легко получить её адрес в памяти:

```
tableStringRu:
  DW dscrString0,dscrString1Ru
  DW dscrString2Ru,dscrString3
  tableStringUs:
  DW dscrString0,dscrString1Us
  DW dscrString2Us,dscrString3
```

Для закодированных строк зарезервированы файлы str1rus.asm, str2rus.asm, str1us.asm, str2us.asm. Добавляем в них строковые описания на свой вкус. Например, название нашего устройства будет выглядеть так (файл str2us.asm):

```
  DB 'M',0
  DB 'Y',0
  DB ' ',0
  DB 'U',0
  DB 'S',0
  DB 'B',0
  DB '-',0
  DB 'D',0
  DB 'E',0
  DB 'V',0
  DB 'I',0
  DB 'C',0
  DB 'E',0
```

Или то же самое на русском (str2rus.asm):

```
  DB 1Ch,4 ; M
  DB 3Eh,4 ; o
  DB 51h,4 ; ё
  DB ' ',0
  DB 'U',0
  DB 'S',0
  DB 'B',0
  DB '-',0
  DB 43h,4 ; у
  DB 41h,4 ; с
  DB 42h,4 ; т
  DB 40h,4 ; р
  DB 3Eh,4 ; о
  DB 39h,4 ; й
  DB 41h,4 ; с
  DB 42h,4 ; т
  DB 32h,4 ; в
  DB 3Eh,4 ; о
```

С дескрипторами закончили. Теперь приступим к созданию обработчика прерывания HISPEED. В файл intusb.asm добавим код, изменяющий поля wMaxPacketSize (для точек) и bcdUSB (для устройства):

```
  mov a,#LOW(512)
  mov dptr,#sizeAlt0Ep1in
  movx @dptr,a
  mov dptr,#sizeAlt0Ep1out
  movx @dptr,a
  mov a,#HIGH(512)
  mov dptr,#sizeAlt0Ep1in+1
  movx @dptr,a
  mov dptr,#sizeAlt0Ep1out+1
  movx @dptr,a
  clr a
  mov dptr,#speedDevice
  movx @dptr,a
  inc dptr
  mov a,#2
  movx @dptr,a
```

Для того чтобы разрешить обработку прерывания HISPEED, в файл system.asm добавим строки:

```
mov a,#21h
mov dptr,#USBIE
movx @dptr,a    ; разрешаем
прерывания HISPEED и SUDAV
```

Обзор действий, выполняемых в стандартных требованиях

Для того чтобы ядро нашего USB-устройства «ожило», осталось написать подпрограммы обслуживания требований. Но перед тем как приступить к этому, надо более подробно рассмотреть действия, выполняемые в том или ином требовании.

Сейчас, когда мы уже определились с логикой работы обработчика прерывания SUDAV, можно спроектировать обобщённый алгоритм подпрограммы, обслуживающей требование. Такой алгоритм показан на рис. 9. Нам уже известно, что значения полей пакета SETUP индивидуальны для каждого требования, поэтому все подпрограммы начинаются с проверки формата полученного требования. В случае обнаружения ошибки будет установлен флаг flagStallEp0, что приведёт к завершению контрольной транзакции маркером STALL. Далее необходимо выяснить, допустимо ли полученное требование в текущем состоянии устройства. Если да, то выполняем действия, предопределённые для данной команды.

В подпрограммах, которые должны вернуть некоторую информацию хосту, дополнительно будут присутствовать действия по размещению передаваемых данных. Для передачи дескриптора необходимо вернуть его адрес в указателе dptr. Для передачи «простых» данных запишем их сразу в буфер EP0BUF.

С предварительной точки зрения, алгоритм простой. Теперь посмотрим, какие действия необходимо выполнить в каждом отдельном требовании. Следует также обратить внимание на то, что требование SET_ADDRESS в FX2LP всегда обслуживается аппаратно, поэтому подпрограмма для его обработки не требуется, а наше устройство может находиться только в одном из двух состояний: адресованном или сконфигурированном.

Итак, действия, выполняемые в требованиях:

- GET_STATUS. Необходимо вернуть 2 байта с информацией о статусе адресата. Если требование обращено к устройству, то биты D0 и D1 содержат информацию о состоянии фла-

гов SelfPowered и RemoteWakeup соответственно. Если получателем требования является интерфейс или точки, то устройство должно находиться в состоянии «сконфигурированное». Статус интерфейса всегда описывается нулями во всех 16 битах. Информационным битом для статуса точки является D0. Он отражает состояние флага Halt для точки, что служит признаком её работоспособности. О статусе контрольной точки можно сообщить, даже если устройство находится в состоянии «адресованное».

Для обработчика данного требования введём несколько программных переменных (файл var.asm):

```
flagSelfPower: DBIT 1
flagRemoteWake:DBIT 1
```

Флаги Halt для точек специально создавать не будем, потому что функцию этого флага выполняет бит STALL, содержащийся в регистре управления соответствующей точки (EP0CS, EP1OUTCS, EP1INCS, EP2CS, EP4CS...).

- CLEAR_FEATURE. В данном требовании необходимо лишь выполнить предопределённые действия. При обращении к устройству – очистить флаг RemoteWakeup (флаг SelfPowered для изменения недоступен). Когда получателем требования является интерфейс или точка, устройство должно быть сконфигурированным. При обращении к интерфейсу никаких действий предпринимать не надо. А вот при обращении к точке необходимо сбросить флаг Halt. Что подразумевается под этим? Допустим, хост передавал данные в некоторую точку N, и в какой-то момент произошло событие, в результате которого точка N не может больше обслуживать передаваемые данные. В таком случае точка N устанавливает флаг Halt, и при всех последующих обращениях к ней хост получает ответ STALL, что служит признаком неработоспособности точки N. Для того чтобы вернуть точку в работоспособное состояние, хост посылает ей требование CLEAR_FEATURE. При получении этого требования флаг Halt сбрасывается и устраняются причины, вызвавшие его установку – то есть происходит инициализация функции, обслуживающей точку.

  Итак, при обращении CLEAR_FEATURE к точке необходимо выполнить следующие действия:
  - убедиться, что в текущем состоянии (конфигурация, интерфейс, альтернативная установка) точка существует;
  - если тип передачи данной точки – interrupt или bulk, а направление передачи – IN, то маркер данных для этой точки следует сбросить в DATA0;
  - очистить буфер точки;
  - инициализировать функцию точки;
  - сбросить флаг Halt (очистить бит STALL).

  В результате выполнения этих действий точка переходит в условное «начальное» состояние и готова к работе.

  При адресации данного требования к контрольной точке никаких действий выполнять не надо, так как бит STALL очищается аппаратно по событию SUTOK.
- SET_FEATURE. Требование по своей цели противоположно предыдущему. При обращении к устройству необходимо установить флаг RemoteWakeup. Эта же команда при соответствующих параметрах может перевести устройство в режим тестирования. В случае, если в подпрограмме не предусмотрена поддержка такого режима, необходимо установить флаг flagStallEp0. На контрольную точку данное требование никак не влияет, потому что она всегда должна быть работоспособна. Даже если мы установим в EP0CS бит STALL, он будет аппаратно сброшен маркером SETUP в следующей же контрольной транзакции. Остальные ситуации возможны только для сконфигурированных устройств. При обращении к интерфейсу, опять же, не надо совершать никаких действий. При обращении к точке необходимо перевести её в нерабочее состояние. Для этого следует:
  - убедиться, что в текущем состоянии точка существует;
  - установить флаг Halt (установить бит STALL);
  - выполнить отключение функции данной точки.
- SET_ADDRESS. Всегда выполняется аппаратно модулем USB. До тех пор, пока устройство не получило уникальный адрес, работа контрольной точки не активируется. Заданный хостом адрес нигде и никогда не фигурирует, но его значение можно узнать в регистре FNADDR.
- GET_DESCRIPTOR. Получателем этого требования может быть только устройство. При получении требования необходимо:
  - найти адрес соответствующего дескриптора;
  - записать адрес в указатель dptr для возврата в обработчик прерывания SUDAV;
  - установить признак передачи дескриптора flagGetDesc.

  Для обработчика данного требования введём программную переменную (var.asm):

```
flagGetDesc:          DBIT 1
```

- SET_DESCRIPTOR. Получателем этого требования также может быть только устройство. Данное требование является опциональным, поэтому его поддержка необязательна. О действиях, выполняемых при получении этого требования, можно судить по его названию:
  - получить данные для дескриптора;
  - сохранить новый дескриптор в памяти устройства;
  - при необходимости сделать переадресацию дескрипторов.
- GET_CONFIGURATION. Устройство должно вернуть один байт, в котором указан номер текущей конфигурации. Если номер равен нулю, то это значит, что устройство не сконфигурировано. Поскольку наше устройство имеет всего одну конфигурацию, то ограничимся битовой переменной, единичное состояние которой говорит о том, что активна конфигурация с номером один (устройство сконфигурировано):

```
flagCfgUsb:           DBIT 1
```

- SET_CONFIGURATION. Получатель требования – устройство. Если номер новой конфигурации соответствует нулю, то устройство переходит в адресованное состояние:
  - сбросить признак «устройство сконфигурировано» (flagCfgUsb);
  - перевести все точки в неработоспособное состояние.

  В противном случае устройство станет сконфигурированным, а все точки, доступные в текущем состоянии, станут работоспособными.

Для того чтобы обеспечить это, необходимо:
- установить признак «устройство сконфигурировано»;
- если в текущем состоянии имеются доступные точки, то перевести их в неработоспособное состояние (аналогично SET_FEATURE);
- установить первый интерфейс, доступный в данной конфигурации;
- для всех доступных точек в новом состоянии выполнить действия, аналогичные выполняемым для точки по требованию CLEAR_FEATURE.

Даже если номер конфигурации, заданный в команде, совпадает с номером текущей конфигурации, то перечисленную последовательность действий всё равно необходимо выполнить.

- GET_INTERFACE. Получателем требования может быть только интерфейс. В данном случае необходимо вернуть один байт, в котором хранится номер активной альтернативной установки для заданного интерфейса текущей конфигурации. Естественно, устройство должно быть сконфигурировано. Для хранения текущего номера альтернативной установки необходимо ввести следующую программную переменную (var.asm):

```
usbAltCur:              DS 1
```

- SET_INTERFACE. Получателем требования является интерфейс сконфигурированного устройства. В случае, если заданные интерфейс и альтернативная установка доступны в текущей конфигурации, необходимо:
  - при наличии доступных точек перевести их в неработоспособное состояние (аналогично SET_FEATURE);
  - установить заданный интерфейс;
  - для всех доступных точек в новом состоянии выполнить действия, аналогичные выполняемым для точки по требованию CLEAR_FEATURE.

  Данную последовательность необходимо выполнять всегда, даже при соответствии параметров команды текущему состоянию устройства.
- SYNCH_FRAME. Требование может быть адресовано только точке сконфигурированного устройства. Если в текущем состоянии устройства имеется точка с изохронным типом передачи данных, использующая неявную синхронизацию данных, то необходимо вернуть два байта, в которых указан номер фрейма синхронизации.

Рассмотрев подробно все стандартные команды, можно заметить, что часть действий некоторых требований совпадает, а некоторые требования устройство вообще может не поддерживать.

Представим, какие действия нам придётся выполнять наиболее часто и что мы можем придумать для упрощения их выполнения.

Часто встречается действие, связанное с проверкой доступности заданной точки в текущий момент. О работоспособности любой точки можно судить по биту VALID в регистре конфигурации соответствующей точки. В FX2LP имеются регистры конфигурации для следующих точек:
- EP1OUTCFG – точка с адресом 1;
- EP1INCFG – точка с адресом 81h;
- EP2CFG – точка с адресом 2 или 82h;
- EP4CFG – точка с адресом 4 или 84h;
- EP6CFG – точка с адресом 6 или 86h;
- EP8CFG – точка с адресом 8 или 88h.

Итак, для проверки доступности в текущий момент точки с заданным адресом напишем подпрограмму getValidEp, по результатам работы которой можно будет сделать вывод о доступности точки.

Следующее, почти так же часто встречающееся действие, связано со сбросом маркера данных в DATA0. В FX2LP для принудительной установки маркера данных используется регистр TOGCTL. При работе с ним требуется указать номер точки (1, 2, 4, 6, 8), направление передачи и устанавливаемый маркер (DATA0 или DATA1). Для сброса маркера данных заданной точки напишем подпрограмму clearToggleEp.

Что скрыто под фразой «установить заданный интерфейс»? Это следует понимать как необходимость включения некоторого набора точек и установки им определённого типа передачи данных в соответствии с дескриптором заданного интерфейса. Звучит устрашающе, но мы это решим следующим образом. Описанные здесь свойства точек определяются в уже знакомом нам регистре конфигурации EPxCFG, поэтому для каждой альтернативной установки создадим таблицы значений конфигурационных регистров. Количество таблиц должно соответствовать количеству альтернативных установок во всех интерфейсах и конфигурациях. В процессе работы по номерам конфигурации, интерфейса и альтернативной установки можно будет быстро отыскать нужную таблицу и скопировать из неё новые значения для регистров EP1OUTCFG, EP1INCFG, EP2CFG и т.д. Таким образом, произойдёт установка заданного интерфейса.

Как это воплотить в жизнь и не запутаться? Начнём с описания констант для регистров конфигурации соответствующих точек в топологии на рис. 8:

- *ep1out.asm*:

```
CFG1_IF0_ALT0_EP1OUT   EQU 0A0h
; включена, bulk
```

- *ep1in.asm*:

```
CFG1_IF0_ALT0_EP1IN    EQU 0A0h
; включена, bulk
```

- *ep2.asm*:

```
CFG1_IF0_ALT0_EP2      EQU 0
; выключена
```

- *ep4.asm, ep6.asm, ep8.asm* – аналогично точке 2;
- *ep0sd.asm*: создаём таблицу значений регистров EPxCFG для состояния CFG-1, IF-0, ALT-0:

```
tableCfg1If0Alt0:
DB CFG1_IF0_ALT0_EP1OUT
DB CFG1_IF0_ALT0_EP1IN
DB CFG1_IF0_ALT0_EP2
DB CFG1_IF0_ALT0_EP4
DB CFG1_IF0_ALT0_EP6
DB CFG1_IF0_ALT0_EP8
```

Если бы у нас была ещё одна альтернативная установка, то и для неё необходимо было бы создать подобную таблицу:

```
tableCfg1If0Alt1:
DB CFG1_IF0_ALT1_EP1OUT
DB CFG1_IF0_ALT1_EP1IN
DB CFG1_IF0_ALT1_EP2
DB CFG1_IF0_ALT1_EP4
DB CFG1_IF0_ALT1_EP6
DB CFG1_IF0_ALT1_EP8
```

Теперь создаём таблицу адресов таблиц конфигурирования точек для заданного интерфейса:

```
tableCfg1If0:
DW tableCfg1If0Alt0
```

и, если бы была вторая альтернативная установка:

```
DW tableCfg1If0Alt1
```

Таким образом, можно описать топологию любой сложности (см. рис. 10).

Поскольку топология нашего устройства достаточно проста, ограничимся созданием таблицы tableCfg1If0 и напишем подпрограмму setConfigEp, которая по заданному номеру альтернативной установки отыскивает нужную таблицу значений регистров и копирует её содержимое в реальные регистры.

Необходимо также вспомнить, что при манипуляциях с конфигурацией или интерфейсом надлежит вызывать некоторые подпрограммы для инициализации или деактивации функции соответствующей точки. Получается, что для каждой точки при изменении интерфейса необходимо вызвать подпрограмму деактивации её функции для текущего состояния, а затем вызвать подпрограмму инициализации функции точки для нового состояния. Как систематизировать все эти функции и обеспечить быстрый доступ к ним? Этот вопрос легко решается, если адреса функций объединить в соответствующие таблицы.

Итак, нам предстоит вызвать подпрограмму, ориентируясь на адрес точки. Всего в FX2LP возможно 10 адресов точек (1, 81h, 2 или 82h, 4 или 84h, 6 или 86h, 8 или 88h), поэтому создадим таблицу с десятью полями, соответствующими возможным адресам. В каждом поле будет храниться указатель на таблицу, в которой перечислены векторы подпрограмм, соответствующих точке для всех имеющихся альтернативных установок. Поскольку топология нашего устройства проста, то и вложенность таблиц минимальная. В случае более сложной топологии необходимо будет использовать вложенные таблицы наподобие рассмотренных на рис. 10.

Приступим к их созданию:

- *ep0sr.asm*:

```
tableInitFunctions: ; таблица
инициализации
  DW tableInitFuncEp1Out,
  tableInitFuncEp1In
  DW tableInitFuncEp2Out,
  tableInitFuncEp2In
  DW tableInitFuncEp4Out,
  tableInitFuncEp4In
  DW tableInitFuncEp6Out,
  tableInitFuncEp6In
  DW tableInitFuncEp8Out,
  tableInitFuncEp8In
tableHaltFunctions: ; таблица
деактивации
  DW tableHaltFuncEp1Out,
  tableHaltFuncEp1In
  DW tableHaltFuncEp2Out,
  tableHaltFuncEp2In
  DW tableHaltFuncEp4Out,
  tableHaltFuncEp4In
  DW tableHaltFuncEp6Out,
  tableHaltFuncEp6In
  DW tableHaltFuncEp8Out,
  tableHaltFuncEp8In
```

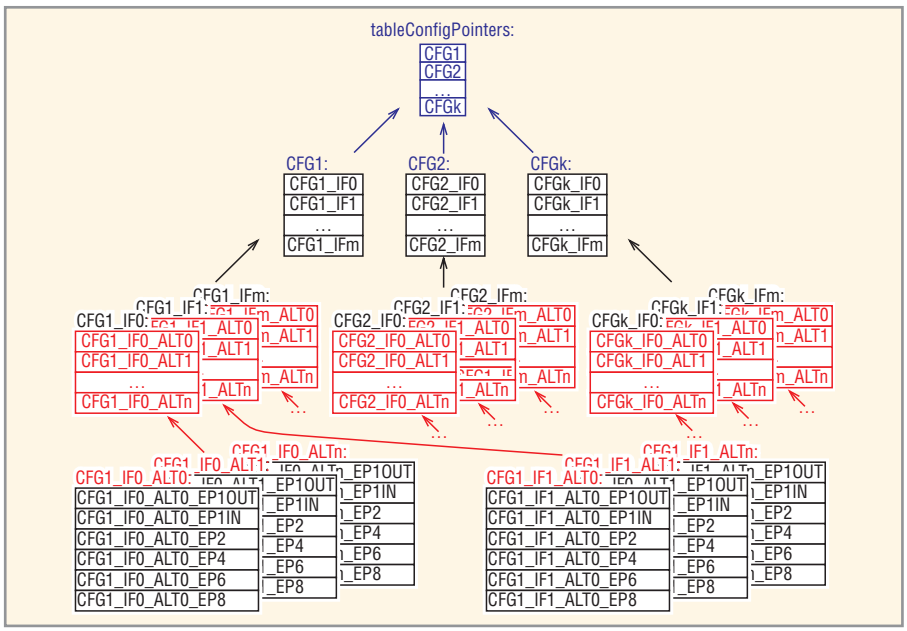

Рис. 10. Организация таблиц конфигурирования точек

- *ep1out.asm*: создаём объявленные таблицы:

```
tableInitFuncEp1Out:
  DW initFuncEp1Out
tableHaltFuncEp1Out:
  DW haltFuncEp1Out
```

Вместо подпрограмм initFuncEp1Out и haltFuncEp1Out создадим временные «заглушки»;

- *ep1in.asm* – аналогично файлу ep1out.asm;
- *ep2.asm*: здесь, казалось бы, необходимо создавать четыре таблицы, поскольку данная точка может иметь адрес 2 или 82h. Но эти адреса взаимно исключают друг друга, и, следовательно, на одну таблицу будет приходиться по два указателя:

```
tableInitFuncEp2Out:
tableInitFuncEp2In:
  DW emptyFunc
tableHaltFuncEp2Out:
tableHaltFuncEp2In:
  DW emptyFunc
```

В качестве подпрограмм укажем пустые функции – «заглушки»;

Таблица 3. Последовательность команд при регистрации USB-устройства в ОС Linux

N	bmRequest	bRequestType	wValue		wIndex		wLength		Пояснения
			wVH	wVL	wIH	wIL	wLH	wLL	
1	80h	06	01	00	00	00	00	08	Запрос дескриптора DEVICE. Задана длина меньше реальной
2	80h	06	01	00	00	00	00	12h	Запрос дескриптора DEVICE. Задана длина, соответствующая реальной
3	80h	06	02	00	00	00	00	09	Запрос дескриптора CONFIGURATION. Команда предназначена для выяснения реальной длины дескриптора
4	80h	06	02	00	00	00	00	20h	Запрос дескриптора CONFIGURATION. Задана длина, соответствующая реальной
5	00	09	00	01	00	00	00	00	Установка первой доступной конфигурации

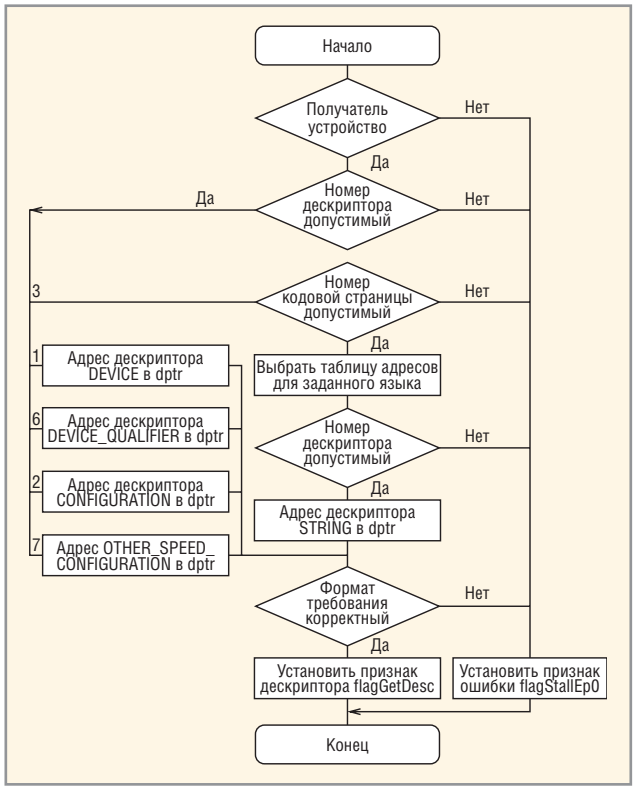

Рис. 11. Алгоритм обработки требования GET_DESCRIPTOR

- *ep4.asm, ep6.asm, ep8.asm* – аналогично точке 2.

Теперь, зная организацию векторов обработчиков, напишем подпрограмму, отыскивающую нужный вектор по адресу точки и номеру альтернативной установки. Поскольку строение таблиц (инициализации и деактивации) одинаковое, то при задании соответствующей базы эта подпрограмма сможет осуществлять запуск по вектору из любой таблицы. Точки входа в эту универсальную подпрограмму назовём соответственно initFunctionEp и haltFunctionEp.

Разработка обработчиков стандартных требований

Наверное, любому программисту интересно будет узнать, какой минимальный набор требований используется хостом для регистрации устройства на шине USB. В таком случае сразу можно было бы сосредоточиться на разработке соответствующих программ и приблизить момент первого включения.

Обратимся к таблицам 3 и 4. В них представлены реальные последовательности команд (повторяющиеся команды не показаны), выполняемые при регистрации USB-устройства в наиболее популярных операционных системах Linux и Windows. Можно заметить, что ключевыми требованиями являются GET_DESCRIPTOR и SET_CONFIGURATION, а в Windows ещё используется SET_INTERFACE. Поэтому в первую очередь приступим к разработке подпрограмм именно этих требований.

Алгоритм обработки требования GET_DESCRIPTOR показан на рис. 11. В соответствии с требованиями спецификации шины USB [6], при нарушении формата пакета следует сообщить об этом хосту маркером подтверждения STALL. Наша подпрограмма при выявлении ошибок устанавливает для обработчика прерывания SUDAV требование flagStallEp0, служащее сигналом для передачи маркера STALL.

Начинаем анализ пакета SETUP с проверки адресата в поле bmRequestType – он должен ссылаться на устройство. Далее проверяем корректность номера дескриптора в поле wValueH. Допустимыми номерами являются 1, 2, 3, 6, 7. Помним, что к дескрипторам интерфейсов и точек явный доступ невозможен, они всегда передаются как составная часть описания конфигурации.

Дескрипторы DEVICE(1) и DEVICE_QUALIFIER(6) всегда существуют в единственном числе, поэтому при их запросе соответствующий адрес сразу сохраняем в dptr. Количество дескрипторов CONFIGURATION(2) и OTHER_SPEED_CONFIGURATION(7) в USB-устройстве может быть больше одного. Но поскольку в топологии нашего устройства имеется всего одна конфигурация, при запросе этих дескрипторов будем считать допустимым только индекс 0. Следовательно, индекс специально не проверяем (проверим его далее на равенство нулю), а сразу задаём соответствующий адрес в dptr.

Для «простых» дескрипторов необходимые адреса вычислены. Теперь проверяем формат пакета, и если он корректный, то устанавливаем для обработчика прерывания SUDAV требование на передачу дескриптора – flagGetDesc.

Осталось рассмотреть логику выбора дескриптора STRING(3). Начинается анализ с поля wIndex, в котором задаётся идентификатор языка. Если значение поля соответствует какому-либо из идентификаторов, поддерживаемых нашим устройством языков, или равно 0 (кодовая страница

Таблица 4. Последовательность команд при регистрации USB-устройства в ОС Windows

N	bmRequest	bRequestType	wValue		wIndex		wLength		Пояснения
			wVH	wVL	wIH	wIL	wLH	wLL	
1	80h	06	01	00	00	00	00	40h	Запрос дескриптора DEVICE. Задана длина больше реальной
2	80h	06	01	00	00	00	00	12h	Запрос дескриптора DEVICE. Задана длина, соответствующая реальной
3	80h	06	02	00	00	00	00	08	Запрос дескриптора CONFIGURATION. Команда предназначена для выяснения реальной длины дескриптора
4	80h	06	02	00	00	00	00	FFh	Запрос дескриптора STRING с индексом 0 – идентификаторы поддерживаемых языков
5	80h	06	03	03	04	09	00	FFh	Запрос дескриптора STRING с индексом 3 – серийный номер
6	80h	06	02	00	00	00	00	FFh	Запрос дескриптора CONFIGURATION. Задана длина больше реальной
7	80h	06	06	00	00	00	00	0Ah	Запрос дескриптора DEVICE_QUALIFIER
8	80h	06	03	00	00	00	00	FFh	Повторный запрос дескриптора STRING с индексом 0
9	80h	06	03	02	04	09	00	FFh	Запрос дескриптора STRING с индексом 2 – название устройства
10	80h	06	01	00	00	00	00	12h	Повторный запрос дескриптора DEVICE
11	80h	06	02	00	00	00	04	00	Повторный запрос дескриптора CONFIGURATION. Задана длина больше реальной
12	00	09	00	01	00	00	00	00	Установка первой доступной конфигурации
13	01	0Bh	00	00	00	00	00	00	Установка интерфейса 0 и альтернативной установки

по умолчанию), то выбираем адрес соответствующей языковой таблицы. Эти таблицы были сформированы нами на этапе создания дескрипторов (файл ep0sd.asm). Далее анализируем индекс дескриптора в поле wValueL. Если в таблице имеется элемент с заданным номером, то копируем его значение в dptr. Завершаем обработку команды установкой требования flagGetDesc.

Как видим, корректная отработка требования GET_DESCRIPTOR зависит от правильности выполненных нами ранее действий:

- создания дескрипторов устройства;
- разрешения работы системы SDP;
- корректности работы обработчика прерывания SUDAV.

Если хотя бы на одном из этих этапов была допущена ошибка, то найти причину неверного обслуживания запроса будет весьма затруднительно.

Требования SET_CONFIGURATION и SET_INTERFACE по набору выполняемых действий очень похожи. Это мы уже заметили в предыдущей главе. Да и при установке конфигурации автоматически становятся активными интерфейс и альтернативная установка, определённые в ней по умолчанию. Таким образом, большая часть действий, выполняемых обработчиками этих требований, может быть объединена.

Начнём с требования SET_INTERFACE. Алгоритм его обработки представлен на рис. 12. В этом алгоритме, в соответствии с нашим проектом, предусмотрен только один интерфейс с множеством альтернативных установок. Если интерфейсов будет несколько, алгоритм следует доработать.

Итак, начинаем с проверки состояния устройства, получателя требования и корректности формата пакета. В случае ошибки, как обычно, устанавливаем требование flagStallEp0. В противном случае проверяем, допустимый ли номер альтернативной установки задан в поле wValue пакета SETUP.

Далее сбросим флаг flagAltUsb, который служит индикатором выполняемой фазы, – до установки нового интерфейса (0) или после (1). Теперь выбираем первую точку из имеющихся в FX2LP (ep1out, ep1in, ep2, ep4, ep6, ep8) и приступаем к последовательному выполнению всех необходимых действий.

Если точка с выбранным номером недоступна, то выбираем следующую. Когда будет обнаружена доступная точка, выполняем проверку текущей фазы – до установки нового интерфейса (flagAltUsb=0) и вызываем подпрограмму закрытия функции данной точки (haltFunctionEp). Фаза «до установки» закончится после перебора всех возможных точек. Тогда меняем фазу на «после установки», запоминаем новое значение альтернативной установки и копируем данные из таблицы значений регистров (tableCfg1If0) в конфигурационные регистры точек. В результате доступным становится другой набор точек. Возвращаемся к выбору первой точки и повторяем циклическую проверку доступности точек. На этот раз, если обнаружена доступная точка, а устройство является сконфигурированным (flagCfgUsb = 1), то выполняем её инициализацию.

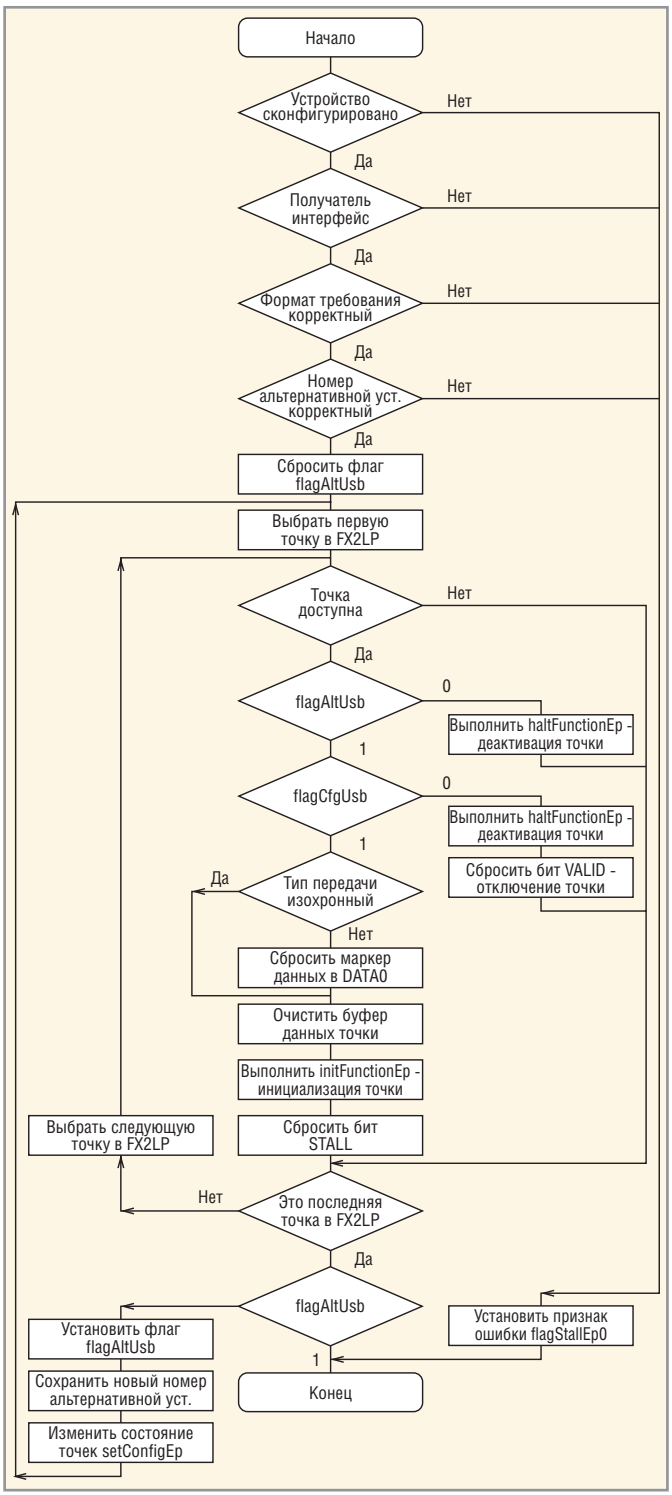

Рис. 12. Алгоритм обработки требования SET_INTERFACE

Вначале необходимо сбросить маркер данных в DATA0 для точек с типом передачи interrupt и bulk. Потом очистить буфер, который мог быть ранее отдан под управление модуля USB, и вызвать подпрограмму инициализации функции этой точки (initFunctionEp). Заканчивается подготовка к работе сбросом бита STALL в регистре управления соответствующей точки.

Перебор точек опять осуществляется последовательно. После достижения последней заканчиваем работу, поскольку флаг flagAltUsb сигнализирует о завершающей фазе установки.

У наиболее внимательных читателей может возникнуть вопрос: «Почему при работе в цикле мы выполняем проверку состояния устройства? Ведь это действие мы выполнили самым первым в алгоритме». Действительно, в самом начале мы выпол-

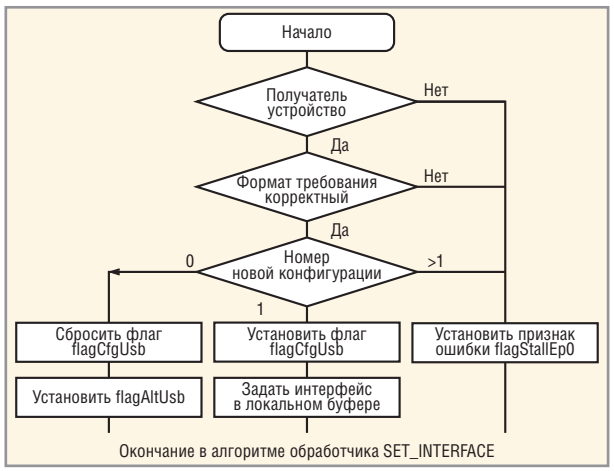

Рис. 13. Алгоритм обработки требования SET_CONFIGURATION

Рис. 14. USB-устройство в среде ОС Windows

няем переход на ошибку, если устройство не сконфигурировано. А данная ветвь в цикле введена для того, чтобы максимально использовать данную подпрограмму при обслуживании требования SET_CONFIGURATION.

Сейчас необходимо вспомнить последовательность действий для команды SET_CONFIGURATION. Начинается она с изменения признака состояния устройства, а далее происходит установка первого доступного интерфейса (для состояния «сконфигурированное») или деактивация всех точек (для состояния «адресованное»).

Алгоритм, представлённый на рис. 13, реализует выполнение начальных действий с последующим переходом в обработчик команды SET_INTERFACE.

При нарушении формата пакета или неверном получателе устанавливаем признак ошибки. Далее определяем новое состояние устройства по значению в поле wValue. Если задан номер существующей конфигурации, то устанавливаем признак того, что устройство сконфигурировано (flagCfgUsb = 1), и копируем номер первой доступной альтернативной установки в поле wValue локального буфера usbBufSetup. Теперь переходим на сброс флага flagAltUsb (рис. 12) в обработчике команды SET_INTERFACE. В результате получим сконфигурированное устройство с установленным по умолчанию интерфейсом и альтернативной установкой.

Если хост переводит устройство в состояние «адресованное», сбрасываем флаг flagCfgUsb (см. рис. 13) и устанавливаем последнюю фазу флагом flagAltUsb для обработчика SET_INTERFACE. Переходя на выбор первой точки в FX2LP этого обработчика (см. рис. 12), начнём выполнять цикл, в котором в соответствии с установленными флагами будет исполняться «лишняя» ветвь алгоритма. Здесь для каждой доступной точки последует вызов деактивирующей функции (haltFunctionEp) и принудительное отключение сбросом бита VALID в регистре EPxCFG. Таким образом, все точки будут выключены, а устройство примет адресованное состояние.

Вот мы и разработали алгоритмы для ключевых требований. Предлагаю попробовать наше устройство в работе.

А как же обработчики для оставшихся требований? Попробуйте реализовать их самостоятельно. Во всяком случае, пока в них нет необходимости, и поэтому вместо подпрограмм можно оставить «заглушки».

Первое включение устройства

Итак, трансляция программы успешно завершена и файл mydevice.hex готов. Место расположения файлов драйвера CyUSB известно [1], программа обслуживания CyConsole установлена. Всё готово к испытанию устройства.

Подключаем наше ядро и видим сообщение об обнаружении нового устройства. На приглашение установить для него драйвер отвечаем согласием и указываем путь к заранее заготовленному файлу CyUSB.inf. После окончания установки можно заглянуть в диспетчер устройств или сразу запустить CyConsole. В обоих случаях видим, что микроконтроллер обнаружен корректно и устройство фигурирует под именем «MCU CY7C68013x».

В списке устройств программы CyConsole выделим наше устройство и перейдём в режим доступа к функциям микроконтроллера (Options → → EZ–USB Interface). На рабочей форме (см. рис. 10 [1]) нас интересует кнопка <Download> – именно она предназначена для загрузки программы в ОЗУ микроконтроллера. Нажимаем её – появляется диалоговое окно выбора файла. Здесь указываем нашу программу – файл mydevice.hex. Прежде чем нажимать кнопку Ok, проанализируем, что мы ожидаем увидеть.

Сразу после загрузки нашей программы в ОЗУ микроконтроллера на ядро 8051 будет подан сигнал Reset, и программа начнёт выполняться. Следуя логике её работы (см. рис. 4), естественно ожидать отключение USB-устройства. Это можно контролировать по исчезновению значка с зелёной стрелкой (символ подключённых USB-устройств) в системном трее Windows. Далее последует небольшая задержка, после которой должно произойти подключение USB-устройства. Windows выдаст сообщение об обнаружении нового устройства и попытается автоматически найти для него драйвер. Так как в только что установленном нами драйвере CyUSB записано устройство с идентификаторами VID=3112 и PID=1973, то этот драйвер и должен быть найден операционной системой. Останется только проверить, под каким именем зарегистрировано устройство.

Итак, нажимаем <Ok> ... И вот сообщение Windows о том, что устройство подключено, настроено и готово к использованию. Открываем диспетчер устройств и видим долгожданный результат своей работы (см. рис. 14).

Литература

1. *Чекунов Д.* EZ-USB FX2LP – универсальное USB-решение. Современная электроника. 2005. № 4.
2. CY7C68013A/CY7C68015A EZ-USB FX2LP USB Microcontroller High-Speed USB Peripheral Controller. www.cypress.com.
3. EZ-USB FX2 Technical Reference Manual. www.cypress.com.
4. *Чекунов Д.* Стандартные требования USB. Современная электроника. 2004. № 2.
5. *Чекунов Д.* Стандартные дескрипторы USB. Современная электроника. 2005. № 1.
6. Universal Serial Bus Specification Revision 2.0. www.usb.org.

Санкт-Петербург, Ленэкспо,
февраль 2006 года

ElectronExpo

идеальный контакт

Второй форум для профессиональных разработчиков, производителей и поставщиков электронной аппаратуры и компонентов.

Ориентирован на промышленный рынок Северо-Западного региона России!

Мы делаем идеальные контакты возможными!

www.electronexpo.ru