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

Окончание. Начало см. СЭ № 1, 2006

Дмитрий Чекунов (г. Ижевск)

В продолжении статьи рассказывается об использовании «малых» точек и модуля I²C для реализации функции обновления встраиваемого ПО. Разработанное ядро получает статус самостоятельного USB-устройства.

Использование «малых» точек для расширения функций ядра

«Хорошим тоном» для любого современного микропроцессорного устройства является поддержка обновлений (update) встраиваемого программного обеспечения.

В схеме нашего ядра [2, рис. 2] для хранения исполняемого кода зарезервирована микросхема AT24C128. Используем «малые» точки для обслуживания AT24C128 и реализуем функцию обновления встраиваемого ПО.

Чтение данных из AT24C128 реализуем с помощью точки EP1IN. С логикой работы точки IN мы уже хорошо знакомы (рис. 4), поэтому, казалось бы, можно сразу перейти к практической части. Однако у данной точки есть один недостаток, который не позволяет организовать её работу по схеме, показанной на рис. 4. Заключается он в том, что буфер EP1INBUF *невозможно очистить программно.*

Когда этот недостаток может проявиться? Давайте представим, что хост начал чтение данных из EP1IN, но все 16 Кб считать не успел. Допустим, последний пакет размером 64 байта остался в EP1INBUF. После этого хост записал новые данные в AT24C128, инициализировал точку EP1IN (установил счётчик адреса AT24C128 в 0) и приступил к чтению данных, чтобы проконтролировать корректность записанной информации. Поскольку буфер EP1INBUF невозможно очистить программно, первый пакет, переданный хосту, будет содержать устаревшую информацию. Естественно, записанные и считанные данные не совпадут. Как же быть в таком случае? Для этого организуем работу точки EP1IN по схеме, представленной на рис. 12.

Во вновь показанной схеме точка формирует данные только в случае реального запроса хоста. То есть все прерывания EP1IN, возникающие после передачи хосту данных из буфера EP1INBUF, игнорируются. А следующий запрос, обращённый к пустому буферу, заставляет FX2LP готовить новый пакет данных. Даже пакеты нулевой длины мы будет формировать по прерыванию IBN. Получается, что для чтения одного пакета данных потребуется передать два запроса IN (IN – NAK + IN – DATA – ACK). В таком случае в буфере EP1INBUF всегда будут актуальные данные и логика работы всех точек FX2LP (и «больших», и «малых»), с точки зрения хоста, будет одинакова.

Итак, для получения работоспособной точки EP1IN нам предстоит внести в проект следующие изменения:

- написать общий обработчик прерывания IBN;
- написать функцию инициализации точки EP1IN;
- написать функцию деинициализации точки;
- написать обработчик прерывания IBN для EP1IN.

Файл system.asm. В подпрограмме initSystem добавим общее разрешение прерываний IBN:

```
mov dptr,#NAKIE
mov a,#1
movx @dptr,a
```

Файл intusb.asm. Здесь напишем общий обработчик для прерывания IBN. Логику обработчика сделаем предельно простой: получить перечень запросов из IBNIRQ и для каждой точки, имеющей установленный запрос и разрешение на прерывание, вызвать подпрограмму обслуживания, а в конце очистить все запросы. Замечу только, что реализация такой логики обработки может избавить нас от редактирования обработчика в будущем:

```
        anl exif,#0EFh
        _MOVX_R_ IBNIRQ
; считали регистр прерываний IBN
```

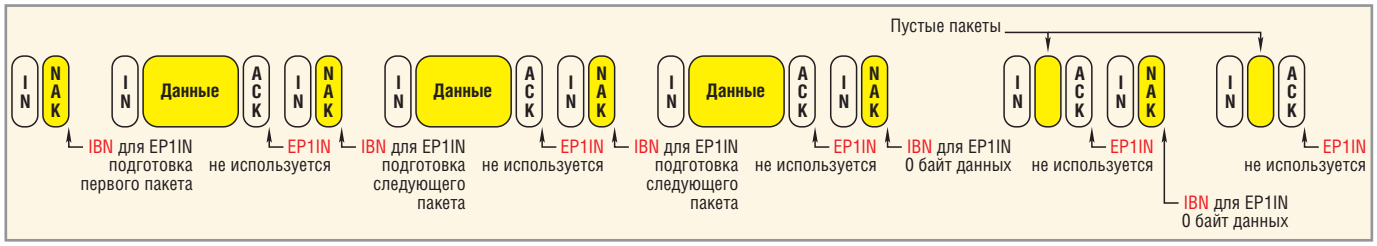

Рис. 12. Взаимодействие хоста и точки EP1IN

Далее сохраним его значения для анализа:

```
        mov b,a
        jnb b.0,isrIbn_1
```

Директивами IFDEF и ENDIF ограничиваем блок команд, который будет включен в текст программы только в том случае, если имеется метка ibnEp0In, то есть нами написан соответствующий обработчик прерывания IBN:

```
        IFDEF ibnEp0In
        _MOVX_R_ IBNIE
        jnb acc.0,isrIbn_1
        lcall ibnEp0In
```

Для контроля обнаруженных обработчиков добавляем следующее сообщение:

```
        $WARNING(MESSAGE: found
IBN for EP0IN)
        ENDIF
isrIbn_1:
        jnb b.1,isrIbn_2
```

Ограничительные директивы используем для всех возможных точек:

```
        IFDEF ibnEp1In
        _MOVX_R_ IBNIE
        jnb acc.1,isrIbn_2
        lcall ibnEp1In
        $WARNING(MESSAGE: found
IBN for EP1IN)
        ENDIF
isrIbn_2:
. . .

isrIbn_6:
```

В конце очищаем все запросы и глобальный запрос IBN в регистре NAKIRQ:

```
        _MOVX_W_ IBNIRQ,b
        _MOVX_W_ NAKIRQ,#1
```

Использование директив позволяет легко добавить обработчик прерывания IBN для любой точки. Если же какой-либо обработчик в тексте отсутствует, то соответствующий ему блок команд не будет включен в программу, что приведёт к уменьшению её размера.

Файл var.asm. Введём программные переменные для работы с шиной I²C:

```
i2cWrAdrHi:     DS      1
i2cWrAdrLo:     DS      1
i2cRdAdrHi:     DS      1
i2cRdAdrLo:     DS      1
; двухбайтные счётчики адреса
для операций записи и чтения со-
ответственно
```

Поскольку любая операция на шине I²C начинается с передачи нескольких байтов служебной информации, то выделим для такого заголовка буфер, в котором будем указывать параметры, необходимые для выполнения той или иной операции. Буфер будет использоваться операциями чтения и записи с разделением по времени:

```
i2cHead:        DS      5
i2cWrDev        EQU     i2cHead
; обращение к устройству для за-
писи
i2cAdrHi        EQU     i2cHead+1
i2cAdrLo        EQU     i2cHead+2
; старший и младший байты адреса
соответственно
```

Следующие два байта имеют разное назначение для операций записи и чтения, что обусловлено необходимостью указания размера буфера записываемых данных и подачи дополнительной команды в подпрограмме чтения:

```
i2cWrPage       EQU     i2cHead+3
; размер записываемого буфера
i2cWrPtr        EQU     i2cHead+4
; указатель на буфер для записи
i2cRdDev        EQU     i2cHead+3
; обращение к устройству для
чтения
```

Длина страницы для чтения не указывается, поскольку она всегда равна 64 байтам (размер буфера EP1INBUF).

```
i2cRdPtr        EQU     i2cHead+4
; указатель на буфер для чтения
i2cData:        DS      1
; временные данные, передаваемые
непосредственно на шину I²C
i2cDeal:        DS      1
; эта переменная хранит количе-
ство байт в промежуточном буфере
i2cCnt:         DS      1
; локальный счетчик байт
используется при взаимодействии
с шиной
```

В битовой области выделим флаги для контроля ошибок:

```
flagAckI2c:     DBIT    1
; ответ микросхемы на шине I²C:
flagErrI2c:     DBIT    1
; конфликт на шине I²C.
```

Данные в операциях записи и чтения будут размещаться в промежуточном буфере, который выделим в верхней части внутреннего ОЗУ микроконтроллера:

```
        ISEG    AT      0C0h
bufEp1:         DS      64
```

Файл ep1in.asm. В функции initFuncEp1In устанавливаем счётчик адреса чтения в 0 и разрешаем прерывание IBN:

```
mov i2cRdAdrHi,#0
mov i2cRdAdrLo,#0
mov dptr,#IBNIE
movx a,@dptr
setb acc.1
movx @dptr,a
```

В функции haltFuncEp1In (деактивация точки) запрещаем прерывание IBN:

```
mov dptr,#IBNIE
movx a,@dptr
clr acc.1
movx @dptr,a
```

Создадим подпрограмму обслуживания запроса IBN – ibnEp1In (соответствующий блок команд в обработчике прерывания IBN будет включен автоматически). Алгоритм для неё представлен на рис. 13. Рассмотрим его подробнее. На первом этапе сохраняем регистры в стек. Далее счётчик байтов, считанных из микросхемы, сбросим в 0. Прежде чем приступить к чтению данных, проверяем счётчик адреса чтения на достижение максимального адреса AT24C128. Если адрес достигнут, то данных для чтения больше нет, о чём следует сообщить хосту. В этом случае мы записываем в регистр EP1INBC значение 0, и хост получит пустой пакет данных. Если данные для чтения ещё имеются, то формируем заголовок служебной информации для микросхемы I²C и вызываем подпрограмму чтения страницы данных из EEPROM. Далее анализируем результат завершения чтения, и если на шине I²C возникла ошибка, то установим STALL для точки EP1IN (ошибка при работе

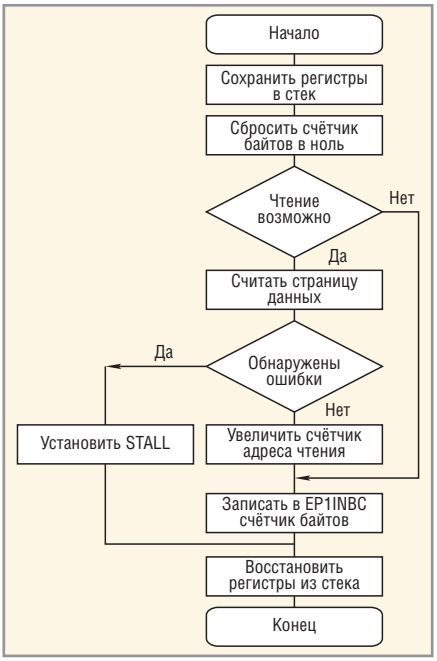

Рис. 13. Алгоритм обслуживания прерывания IBN для EP1IN

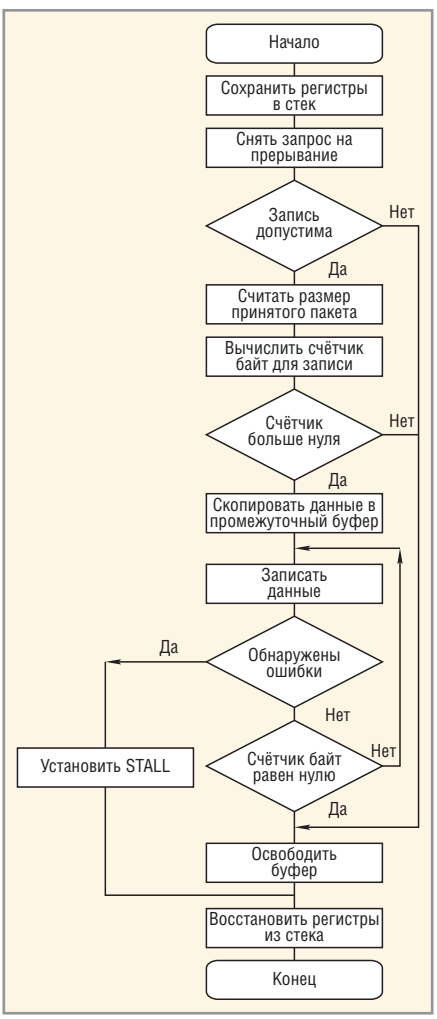

Рис. 14. Алгоритм обслуживания прерывания EP1OUT

с EEPROM). При успешном завершении чтения копируем данные из промежуточного буфера в буфер EP1INBUF и увеличиваем счётчик адреса. Завершаем обслуживание точки EP1IN записью размера считанной страницы в регистр EP1INBC. В завершение восстанавливаем регистры. Подпрограмму чтения данных из EEPROM напишем позднее, а пока сделаем для нее «заглушку». На этом работу с точкой EP1IN закончим.

Перейдём к точке EP1OUT, которая используется для записи данных в микросхему AT24C128. В логике работы точек с направлением OUT нет никаких хитростей, поэтому сразу приступаем к редактированию файла.

Файл ep1out.asm. В функции initFuncEp1Out сбрасываем счётчик адреса для записи в 0 и разрешаем прерывание EP1OUT:

```
mov i2cWrAdrHi,#0
mov i2cWrAdrLo,#0
mov dptr,#EPIE
movx a,@dptr
setb acc.3
movx @dptr,a
```

В функции haltEp1Out всего лишь запрещаем прерывание:

```
mov dptr,#EPIE
movx a,@dptr
clr acc.3
movx @dptr,a
```

Завершаем «оживление» точки EP1OUT написанием обработчика прерывания isrEp1Out. Алгоритм данного обработчика представлен на рис. 14. Начинаем работу с сохранения регистров в стек и снятия запроса на прерывание (здесь, в отличие от IBN, прерывание индивидуальное). Далее проверяем счётчик адреса записи на превышение адресного пространства AT24C128. Если адрес записи больше допустимого, то переходим на освобождение буфера. В противном случае считываем размер принятого пакета и вычисляем допустимое количество байтов для записи. Если счётчик байтов равен нулю – освобождаем буфер EP1OUTBUF, иначе – копируем данные для записи в промежуточный буфер bufEp1. Перед тем как вызвать подпрограмму записи, формируем заголовок со служебной информацией. После каждой операции записи необходимо проверять отсутствие ошибок на шине I²C. При возникновении ошибки устанавливаем STALL для точки EP1OUT, сообщая хосту об отказе микросхемы EEPROM. Если же ошибок нет, то пересчитываем оставшееся количество байтов и повторяем запись, пока счётчик байтов больше нуля. Когда все допустимые данные будут записаны, освобождаем буфер EP1OUTBUF. Завершаем обработчик восстановлением регистров из стека. Подпрограммы страничной и побайтной записи данных в AT24C128 пока оформим в виде «заглушек»; мы напишем их после знакомства с модулем I²C.

Файл util.asm. Для манипуляций с адресными счётчиками чтения и записи, а также для вычислений допустимого для записи количества байтов потребуются функции сложения и вычитания слова и байта. Первая подпрограмма – byte2AddByte1 – будет суммировать слово и байт, вторая – byte2SubByte1 – будет вычитать байт из слова.

Для копирования данных из промежуточного буфера в буфер внешнего ОЗУ добавим подпрограмму movInto2Ext.

Модуль I²C

Встроенный в FX2LP модуль I²C освобождает программистов от рутинных операций на физическом уровне шины I²C. Все передающие и принимающие действия реализованы в модуле аппаратно. Кроме того, модуль I²C может функционировать на одной из двух возможных скоростей обмена (100 или 400 кГц), устанавливаемых программно.

Ресурсы FX2LP для управления модулем I²C

Для взаимодействия с модулем I²C в МК семейства FX2LP имеется три регистра:

- I2CS – регистр управления и статуса:
  - бит DONE (D0) – признак завершения передачи/приёма байта на шине I²C. При обращении к регистру I2DAT бит DONE аппаратно сбрасывается в 0 и устанавливается в 1 только после окончания действия. Данный бит является одним из источников прерывания модуля I²C;
  - бит ACK (D1) – признак готовности абонента на шине. Бит ACK актуален только в операциях записи. На каждом девятом такте линии SCL инверсное значение с

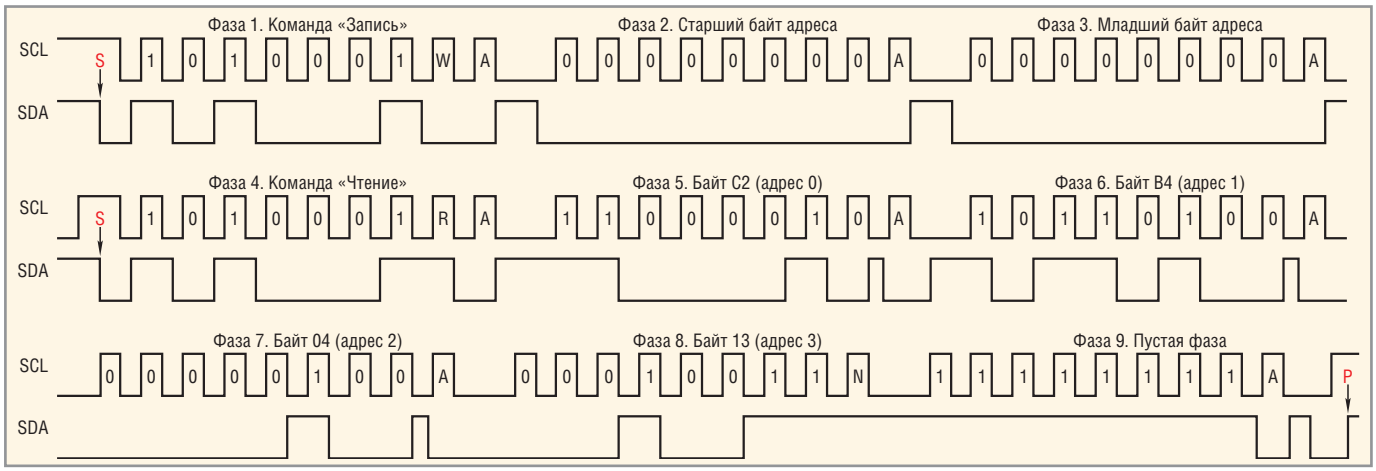

Рис. 15. I²C-транзакция

линии SDA копируется в бит ACK, что позволяет делать выводы о готовности абонента. Значение 0 соответствует ответу NAK, 1 – ACK;

– бит BERR (D2) – признак конфликта на шине I²C (наличие двух ведущих устройств). BERR аппаратно сбрасывается в 0 при обращении к I2DAT. В случае обнаружения конфликта работа модуля I²C прекращается до появления события «СТОП»;

– биты ID1, ID0 (D4, D3) – идентификаторы наличия загрузочной микросхемы на шине I²C, обнаруженной в момент запуска FX2LP. Соответствие типа микросхемы состоянию битов показано в табл. 3;

– бит LASTRD (D5) – сигнал завершения пакетного чтения. Значение данного бита передаётся на шину I²C на девятом такте SCL, сигнализируя о продолжении или завершении пакетного чтения;

– бит STOP (D6) – требование формирования события «СТОП» на шине. Для активизации требования необходимо установить бит в 1. Модуль I²C сформирует событие «СТОП» сразу после завершения такта ACK и сбросит бит STOP в 0;

– бит START (D7) – требование формирования события «СТАРТ» на шине. Данное требование будет выполнено перед следующей передачей данных на шину. После формирования события бит будет аппаратно сброшен в 0;

- I2DAT – регистр данных. При обращении к регистру модуль I²C аппаратно формирует последовательность необходимых сигналов на шине I²C;
- I2CTL – регистр настройки шины:
  – с помощью бита 400KHZ (D0) производится выбор тактовой частоты модуля I²C. Если бит сброшен в 0, то обмен осуществляется на частоте примерно 100 кГц;
  – бит STOPIE (D1) – флаг разрешения ещё одного источника прерывания модуля I²C. Если STOPIE установлен в 1, то событие «СТОП» является таким же источником прерывания, как и бит DONE.

Управление модулем I²C

Наличие аппаратной поддержки протокола I²C в МК упрощает работу программиста. Однако при переходе от «ручного» управления линиями SCL, SDA к автоматизированному (модуль I²C) следует познакомиться с логикой работы аппаратного модуля.

На рис. 15 представлена операция чтения четырёх байт, начиная с адреса 0, из микросхемы AT24C128, имеющей адрес 1. Рассмотрим последовательность действий, необходимых для осуществления этой операции:

1. Установим бит START в регистре I2CS;
2. Сформируем байт команды – A2h (тип микросхемы + адрес микросхемы + команда на запись). Запишем байт в регистр I2DAT;

Модуль I²C начинает выполнять фазу 1 (см. рис. 15), где формирует событие «СТАРТ» и передаёт байт данных. На девятом такте модуль I²C сохраняет состояние шины SDA (ответ микросхемы) в бит ACK в инверсном виде.

3. Контролируем состояние бита DONE в регистре I2CS и ждем его установки. Проверяем состояние бита ACK, чтобы оценить готовность AT24C128;
4. Записываем старший байт адреса – 00 в регистр I2DAT;

Модуль I²C формирует фазу 2, но на этот раз события «СТАРТ» нет, поскольку бит START был аппаратно сброшен в предыдущей фазе.

5. Контролируем состояние бита DONE (I2CS) и ждём завершения транзакции;
6. Запишем младший байт адреса – 00 в регистр I2DAT;
7. Контролируем завершение операции по состоянию бита DONE;

Действия, соответствующие пп. 6 и 7, показаны в фазе 3 (рис. 15). После завершения этой фазы внутренний счётчик AT24C128 примет значение 0. Теперь переходим к чтению данных.

8. Установим бит START в регистре I2CS;
9. Сформируем байт команды – A3h (тип микросхемы + адрес микросхемы + команда на чтение). Запишем байт в регистр I2DAT;

Модуль I²C формирует последовательность, начинающуюся с события «СТАРТ», – фаза 4.

10. Ждём завершения транзакции (DONE);

Таблица 3. Идентификаторы загрузочной микросхемы на шине I²C

ID1	ID0	Микросхема I²C
0	0	Нет микросхемы
0	1	Микросхема с однобайтной адресацией
1	0	Микросхема с двухбайтной адресацией
1	1	Не используется

11. Выполняем чтение из регистра I2DAT;

Операция чтения заставляет модуль сформировать последовательность импульсов и принять данные (фаза 5). Однако сама операция чтения не ждёт завершения транзакции и получает из I2DAT случайные данные. Достоверные же данные окажутся в I2DAT только после установки DONE, где их и можно будет забрать в следующий раз.

12. Ждём завершения транзакции (DONE);
13. Выполняем чтение из регистра I2DAT. Получаем байт со значением C2h из ячейки с адресом 0;

В это время на шине I²C выполняется следующая транзакция (фаза 6) – идёт чтение байта из ячейки с адресом 1.

14. Контролируем DONE – ждём завершения транзакции;
15. Следующее чтение из регистра I2DAT. Получаем байт со значением B4h из ячейки с адресом 1;
16. Ждём окончания транзакции (DONE);

Транзакция, соответствующая фазе 7, завершена. Мы уже считали два байта, и третий байт находится в I2DAT. Поскольку мы решили считать всего 4 байта, то необходимо сообщить об этом микросхеме AT24C128.

17. Установим бит LASTRD в регистре I2CS;
18. Выполняем следующее чтение из регистра I2DAT. Получаем байт со значением 04 из ячейки с адресом 2;

В это время модуль I²C выполняет транзакцию, в которой на девятом такте SCL будет установлен признак конца пакетного чтения – фаза 8.

19. Ждём окончания транзакции (DONE);
20. Выполняем последнее чтение из I2DAT – получаем байт 13h из ячейки с адресом 3;

Модуль I²C генерирует последовательность сигналов SCL, но AT24C128 уже не отвечает – фаза 9.

21. Установим требование STOP.

Событие «СТОП» формируется модулем I²C сразу после такта подтверждения в фазе 9.

Итак, считываемые данные доходят от модуля I²C с задержкой на одну транзакцию, а для завершения транзакции необходимо установить бит LASTRD за одну транзакцию до реального завершения обмена.

Разработка функций обслуживания микросхемы I²C

Добавим в проект два новых файла. В одном из них будут описаны константы используемой микросхемы, а в другом функции для работы с шиной I²C.

Редактируем *файл config.asm*:

```
$INCLUDE(at24c128.asm)
$INCLUDE(i2c.asm)
```

Файл at24c128.asm. Здесь укаажем специфические параметры, присущие микросхеме AT24C128:

```
SIZE_PAGE_I2C  EQU     64
; размер записываемой страницы;
SIZE_MEMORY_I2CEQU     4000h
; объём памяти микросхемы;
WORD_WRITE_I2C EQU     0A0h
; команда "запись" без указания
адреса микросхемы;
WORD_READ_I2C  EQU     0A1h
; команда "чтение" без указания
адреса микросхемы.
```

Использование файла с константами позволяет легко выполнить адаптацию программы в случае изменения аппаратной части. Например, если в нашем ядре будет применена другая микросхема EEPROM, то достаточно заменить файл at24c128.asm на файл с параметрами, соответствующими используемой микросхеме, и изменения в коде программы вносить не потребуется.

Теперь перейдём к подпрограммам работы с шиной I²C. У каждого программиста, работавшего с шиной I²C, наверняка имеются отлаженные подпрограммы чтения и записи данных. Как правило, эти подпрограммы реализуют алгоритмы пакетной записи или чтения данных, не манипулируя сигналами шины. Для управления линиями SCL и SDA на физическом уровне используются короткие подпрограммы, позволяющие скрыть рутинные переключения тактовых сигналов, анализ и опрос шины данных. Набор подпрограмм обычно включает в себя:

- outs – формирует событие «СТАРТ», передаёт байт данных и сохраняет ответ подчинённой микросхемы;
- out – передаёт байт данных и сохраняет ответ подчинённой микросхемы;
- in – принимает байт данных;
- next – формирует ACK (9-й такт в 0), сообщая подчинённой микросхеме о продолжении пакетного чтения;
- down – формирует NACK (9-й такт в 1), сообщая подчинённой микросхеме о завершении пакетного чтения;
- stop – формирует событие «СТОП».

Файл i2c.asm. Начнём редактирование с создания подпрограмм физического управления шиной.

Действия подпрограмм outs и out во многом схожи, различие состоит лишь в том, что outs предварительно формирует событие «СТАРТ». Поэтому напишем подпрограмму outs и в ней создадим лишь вход для подпрограммы out:

```
outs:   mov a,#80h
; устанавливаем данные для собы-
тия "старт"
        mov dptr,#I2CS
        movx @dptr,a
```

Формирование события «СТАРТ» подготовили, далее следует передача данных, то есть здесь начинается подпрограмма out:

```
out:    mov a,i2cData
; загружаем данные
        mov dptr,#I2DAT
        movx @dptr,a
; передали данные
```

Контролируем завершение транзакции – ждём установки бита DONE:

```
        mov dptr,#I2CS
o_1:    movx a,@dptr
        jnb acc.0,o_1
```

Сохраняем возможные ошибки на шине I²C для последующего анализа:

```
        mov c,acc.1
        mov flagAckI2c,c
; сохраняем бит ACK
        mov c,acc.2
        mov flagErrI2c,c
; сохраняем бит BERR
        ret
```

Теперь перейдём к подпрограмме in. Из практических исследований

помним, что чтение из регистра данных I2DAT формирует служебные сигналы для приёма байта, но сам байт становится доступен лишь при следующем обращении к регистру. Поэтому чтение начинается с контроля завершения транзакции на шине I²C – бит DONE установлен в 1:

```
in:     mov dptr,#I2CS
i_1:    movx a,@dptr
        jnb acc.0,i_1
```

Сохраняем бит BERR для дальнейшего анализа (бит ACK при чтении не является ответом подчиненной микросхемы):

```
        mov c,acc.2
        mov flagErrI2c,c
; сохраняем бит BERR
```

Считываем данные, которые были получены в предыдущей фазе чтения:

```
        mov dptr,#I2DAT
        movx a,@dptr
        ret
```

Практически в подпрограмме next нет необходимости. Однажды установленный бит LASTRD будет аппаратно сброшен в 0 после завершения приёма очередного байта. Однако наличие данной функции обеспечит совместимость подпрограмм высокого уровня и нового набора подпрограмм физического управления шиной:

```
next:   clr a
        mov dptr,#I2CS
        movx @dptr,a
        ret
```

Подпрограмма down для FX2LP является более актуальной. Помним, что данную команду необходимо подавать за одну фазу до реального завершения чтения:

```
down:   mov a,#20h
        mov dptr,#I2CS
        movx @dptr,a
        ret
```

Формирование события «СТОП» происходит после выдачи девятого такта любой транзакции. Признаком завершения операции является состояние бита STOP, равное нулю. Сброс данного бита происходит аппаратно:

```
stop:   mov a,#40h
; загрузили флаг STOP
        mov dptr,#I2CS
; загрузили адрес контрольного
регистра
        movx @dptr,a
; записали данные
```

Ждём аппаратного сброса бита STOP:

```
s_1:    movx a,@dptr
        jb acc.6,s_1
; ждём события "стоп"
        ret
```

Итак, подпрограммы физического управления шиной написаны. Осталось разработать подпрограммы высокого уровня, где будет происходить обмен данными и сохранение ошибок.

Для точки EP1IN потребуется написать подпрограмму страничного чтения данных. В этой подпрограмме необходимо реализовать:

- чтение данных с заданного адреса, хранящегося в счётчике адреса чтения (i2cRdAdrHi, i2cRdAdrLo);
- контроль наличия микросхемы при записи начального адреса чтения в EEPROM.

Входным параметром для подпрограммы будет указатель на буфер заголовка со служебной информацией следующего формата:

- командное слово записи;
- 2 байта – адрес ячейки;
- командное слово чтения;
- указатель на приёмник данных в промежуточном буфере.

Попробуйте написать данную подпрограмму самостоятельно.

Мы же перейдём к записи данных в EEPROM, поскольку такая функция будет востребована уже в ближайшем будущем. Здесь нам предстоит написать две подпрограммы. Одна – для записи байта, другая – для записи страницы данных.

Как известно, микросхемы I²C поддерживают страничную запись данных. Допустимый размер страницы для записи зависит от микросхемы. Так, например, для AT24C128 и AT24C256 допустимый размер составляет 64 байта, для AT24C64 – 32 байта. Запись неполной страницы в микросхему может привести к сохранению данных во временном буфере и переход к ожиданию его полного заполнения. Для того чтобы исключить подобную ситуацию, будем осуществлять побайтную запись, если остаток данных меньше размера страницы.

Итак, подпрограмма записи одного байта – writeByteI2c. Для данной подпрограммы входным параметром является указатель _ptrSrc, указывающий на буфер заголовка со служебной информацией следующего формата:

- командное слово;
- 2 байта – адрес ячейки;
- байт данных для записи.

Подпрограмма writeByteI2c будет выглядеть следующим образом:

```
writeByteI2c:
        mov i2cData,@_ptrSrc
; любые данные, физически
передаваемые на шину I²C,
записываем в переменную i2cData
        mov i2cCnt,#10
; устанавливаем счётчик
обращений к микросхеме равным 10
wBI2c_2:
        lcall outs
; посылаем событие "старт" и
командное слово
        jb flagAckI2c,wBI2c_1
; если микросхема готова, то
переходим на продолжение записи,
иначе повторяем запрос
        lcall stop
        djnz i2cCnt,wBI2c_2
        sjmp wBI2c_3
```

Если в течение десяти запросов не был получен ответ ACK, то завершаем операцию записи с ошибкой:

```
wBI2c_1:
        mov i2cCnt,#3
; устанавливаем новое значение
для счётчика, теперь нам оста-
лось записать двухбайтный адрес
ячейки и байт данных - счётчик
равен 3
wBI2c_4:
        inc _ptrSrc
        mov i2cData,@_ptrSrc
        lcall out
; посылаем адрес и данные
        djnz i2cCnt,wBI2c_4
        lcall stop
; посылаем событие "стоп"
```

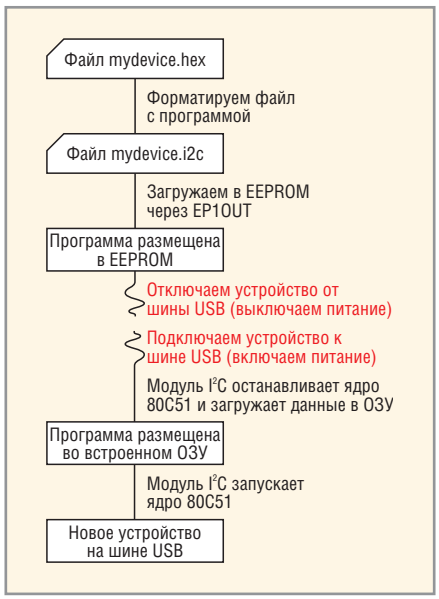

Рис. 16. Загрузка исполняемого кода через I²C

```
wBI2c_3:
        ret
```

Подпрограмма страничной записи – writePageI2c. Входным параметром также является указатель _ptrSrc, указывающий на буфер заголовка, однако формат служебной информации немного отличается:

- командное слово;
- 2 байта – начальный адрес записи;
- размер записываемой страницы;
- указатель на начало данных в промежуточном буфере.

Подпрограмма страничной записи немногим отличается от предыдущей подпрограммы, поэтому здесь её рассматривать не будем.

Итак, мы разработали подпрограммы записи в микросхему EEPROM. Данные подпрограммы совместно с подпрограммами обслуживания «малых» точек МК реализуют функцию обновления встраиваемого программного обеспечения.

Таблица 4. Формат загрузочной записи C0h

Адрес в EEPROM	Значение
0	C0h
1	Младший байт VID
2	Старший байт VID
3	Младший байт PID
4	Старший байт PID
5	Младший байт DID
6	Старший байт DID
7	Конфигурационный байт

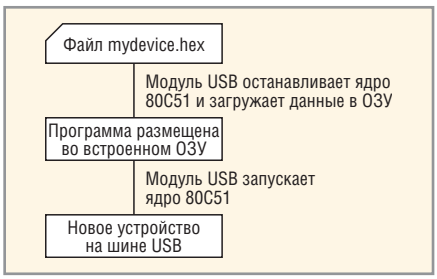

Рис. 17. Загрузка исполняемого кода через USB

Создание самостоятельного устройства

Подготовка загрузочной программы

Поскольку в схеме нашего ядра имеется микросхема для хранения встраиваемого программного обеспечения, а в проекте реализована поддержка записи в названную микросхему, то мы имеем возможность перевести ядро в разряд самостоятельных устройств. В таком случае наше ядро при подключении к шине USB будет сразу определяться как разработанное нами устройство, а не как EZ-USB FX2LP. Для этого нам всего лишь необходимо закодировать исполняемую программу определённым образом и записать её в загрузочную микросхему (AT24C128), как показано в верхней части рис. 16.

Форматы загрузочных записей

Модуль I²C, имеющийся в FX2LP, после прохождения внешнего сигнала RESET ищет в загрузочной микросхеме запись одного из двух рассмотренных ниже форматов.

Формат «C0h». Запись данного вида предназначена для установки определённых значений идентификаторам VID, PID, DID, с которыми FX2LP будет работать до отключения питания. Формат записи показан в табл. 4. Первый байт – идентификатор C0h. Далее по два байта занимают идентификаторы VID, PID, и DID. Завершает запись конфигурационный байт, назначение которого рассмотрим позднее.

Обнаружив такую запись, модуль I²C загружает и сохраняет идентификаторы для модуля USB. Ядро 80C51 остаётся выключенным, а модуль USB самостоятельно регистрируется на шине с установленными идентификаторами. Далее, по сценарию фирмы Cypress, должна последовать загрузка исполняемого кода через USB (см. рис. 17).

Формат «C2h». Запись такого вида предназначена для загрузки исполняемого кода через интерфейс I²C. Формат записи показан в табл. 5. Идентификатор записи (первый байт) имеет значение C2h. Последующие семь байт имеют назначение, аналогичное соответствующим байтам записи формата C0h. Далее следуют пакеты исполняемого кода. Каждый пакет имеет заголовок, в котором указан размер пакета (первые два байта) и начальный адрес в ОЗУ FX2LP (следующие два байта), с которого будут загружены данные. Далее следует исполняемый код. Максимальный размер пакета составляет 1023 байта, поэтому вся программа разбивается на несколько пакетов. За последним пакетом следует служебная запись вида «80h 01h E6h 00h 00h». Обнаружив эту запись, модуль I²C запускает ядро 80C51, и ядро начинает выполнять загруженную программу.

После загрузки записи вида «C2h» модуль I²C аппаратно устанавливает бит RENUM (регистр USBCS) в 1, что приводит к отключению аппаратного обслуживания стандартных требований модулем USB. В этом случае стандартные требования должна обслуживать программа пользователя или следует программно сбросить RENUM в 0.

Конфигурационный байт, находящийся в обеих записях на седьмой позиции, позволяет управлять подключением к шине USB и выбрать скорость обмена на шине I²C. Так, если бит D0 (400KHZ) будет установлен в 1, то тактовая частота на шине I²C составит 400 кГц. Бит D6 (DISCON) определяет подключение модуля USB к шине. В нашем случае D6 должен быть установлен в 1, тогда на момент загрузки кода из AT24C128 модуль USB будет отключен от шины. Подключение к шине произойдет программно после запуска ядра 80C51.

Инструменты для кодирования исполняемого кода

В дополнительном материале к статье имеется программа hex2i2c. Данная программа транслирует файл hex в файл с расширением i2c. Полученный файл представляет собой запись вида «C2h», в которой изначально модуль USB отключен, а быстродействие

Таблица 5. Формат загрузочной записи C2h

Адрес в EEPROM	Значение
0	C2h
1	Младший байт VID
2	Старший байт VID
3	Младший байт PID
4	Старший байт PID
5	Младший байт DID
6	Старший байт DID
7	Конфигурационный байт
8	Старший байт размера блока данных
9	Младший байт размера блока данных
10	Старший байт адреса загружаемой области в FX2
11	Младший байт адреса загружаемой области в FX2
12	Первый байт блока данных
…	…
…	Последний байт блока данных
N	Старший байт размера блока данных
N+1	Младший байт размера блока данных
N+2	Старший байт адреса загружаемой области в FX2
N+3	Младший байт адреса загружаемой области в FX2
N+4	Первый байт блока данных
…	…
…	Последний байт блока данных
K	80h
K+1	1
K+2	E6h
K+3	0
K+4	0

модуля I²C оставлено по умолчанию – 100 кГц.

В Интернете можно найти универсальную программу кодирования от фирмы Cypress. Программа называется hex2bix и позволяет создавать записи любого формата для всех микроконтроллеров фирмы Cypress.

Кодирование рабочей программы

Подготовка к созданию самостоятельного устройства начинается с кодирования программы (см. рис. 16). Выполним трансляцию исполняемого кода командой hex2i2c mydevice.hex, в результате чего получим файл mydevice.i2c, представляющий собой запись формата C2h, готовый для записи в загрузочную микросхему.

Запись загрузочной программы

Для выполнения всех дальнейших действий необходимо, чтобы ядро уже функционировало по программе mydevice.hex.

Теперь запускаем программу CyConsole и переходим на закладку Other Endpt Xfers. Выбираем точку 0x01 (OUT), в поле Bytes of data вводим число, равное размеру файла mydevice.i2c или превышающее его, и нажимаем кнопку <File Transfer>. В диалоговом окне указываем подготовленный файл (mydevice.i2c) и нажимаем <Ok> – начинается передача данных в точку EP1OUT. Обработчик точки записывает принятые данные в микросхему AT24C128. После завершения записи можно назвать наше ядро самостоятельным устройством.

Ядро – самостоятельное устройство

Отключим кабель USB и подключим его вновь. События начинают развиваться, как показано в нижней части рис. 16. После завершения сигнала RESET модуль I²C удерживает ядро 80C51 в выключенном состоянии и начинает поиск загрузочной микросхемы и записи в ней. При обнаружении записи «C2h» модуль загружает исполняемый код по заданному адресу в ОЗУ и запускает ядро 80C51. Далее происходит программное подключение устройства к шине USB, и мы можем видеть признаки его обнаружения операционной системой.

На этом развитие нашего ядра закончено, теперь на его базе возможна разработка более сложных устройств.

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.