16FUSB - A USB 1.1 implementation for PIC16F628/628A

Author: Emanuel Paz

Contact: efspaz@gmail.com

16FUSB at Google Code

Changelog:

06/02/2012 version 1.2:

- Optional interrupt transfers support, with IN and OUT endpoints.
- Optional HID support.
- Bit stuffing removal fixed.
- Flash and RAM usage reduced.
- Added loop code call for functionalities.

05/08/2012 version 1.1:

- Decoupling of functionalities and core code.
- Data decode moved from ISR to MainLoop.

The text regards to current version: v1.2

Introduction

How many PIC developers did not use the PIC16F628 at least once in life? This was one of the most popular microcontrollers ever released by Microchip. Many PIC programmers had their first steps using this popular microcontroller. In fact, it's still in production and still widely used by applications that do not require advanced resources.

Nowadays, with the need to use communication via USB port, PICs as the 18F4550 and 18F2550 are becoming increasingly popular. Of course, not only for their USB port, but also for all other features present in the 18F family chips. However, in some simpler projects may be more interesting to use the old 16F628 which can cost up three times less than a PIC18F2550, for example.

Since PIC16F628/628A do not have any kind of USB interface, a project that requires communication with this bus, at first, completely cripples using this chip. But what if we have a firmware based implementation of the USB? That's exactly what the project presented here - 16FUSB - aims.

The 16FUSB is a software implementation of the USB low-speed for PIC16F628/628A microcontroller. All communication stuff is done by the firmware, completely discarding the need to use an additional chip. From the 16FUSB core is possible to add implementations of other protocols such as, for example, I2C, SPI or simple write data from the USB host (PC) directly to the PIC pins. A low-speed USB software implementation can be a great alternative for those who want lower cost projects and at the same time also doesn't need high speed data transfer.

The project is open source and is hosted on Google Code, with the address: http://code.google.com/p/16fusb. From there, you can download the source code directly from svn as well as releases, updates and track defects. To do this you just need a Google account.

Some of the ideas presented here have as inspiration a similar design for Atmel AVR microcontrollers: The IgorPlugUSB, found in http://www.cesko.host.sk/IgorPlugUSB/IgorPlug-USB%20(AVR)_eng.htm.

To understand the operation of the firmware, described below, it's essential to have knowledge about the USB protocol, which will not be treated in more details here. A direct and good description can be found at http://www.beyondlogic.org/usbnutshell/usb1.shtml. The text is easy to understand and covers all the fundamental concepts of low and high level protocol.

Firmware (core)

The development of such a firmware really isn't a trivial thing, especially taking into account the limitations of a simple microcontroller, such as the PIC16F628/628A, especially with regard to its speed.

A PIC16F628/628A can work with frequencies up to 20MHz. However, each instruction cycle takes four clock cycles. This means that, in fact, with a 20MHz crystal we have our PIC running on 5MHz (20 / 4 = 5). Doing a little overclock, with a 24MHz crystal, we can run programs on 6MHz (or 6Mips). Since the speed of the USB low-speed is 1.5 Mbps, we can obtain a total of four instructions (6 / 4 = 1.5) to treat each bit of data during transfer. That is, each bit of the USB bus takes the time of four instructions of our PIC.

As it's not hard to see, with only four instructions to encode/decode the NRZI, insert/remove the bit stuffing and even check the end of packet (EOP), the work becomes impossible. Fortunately using a few more tricks we can work around this problem, as we shall see.

The default endpoint, EP0, treats every control transfer messages. Although this transfer type is more used to the device setup, we can use it for general purposes too. Additionally, it's possible to use IN and OUT interrupt transfer enabling respectives endpoints on config file (def.inc). Interrupt endpoints are EP1 IN and EP1 OUT. If you're afraid about using a device driver (libusb in our case), you may enable HID option, write your own Report Descriptor or use the default one.

In general the firmware, which was written in assembly, can be divided into two parts: ISR and MainLoop.

The ISR (Interrupt Service Routine) performs the following operations:

- Waits for data transfer starts with the Sync Pattern;
- Receives and immediately save the package (still coded and bit stuffing) in an input buffer (RX_BUFFER);
- Checks in the address field if the package is really for device;
- Checks the packet type (Token or Data);
- If it's a OUT or SETUP token, saves the PID of the package to know the origin of the data that will come in the next packet;
- Sends acknowledgment packet (ACK) to the host;
- Sends a non-acknowledgment (NAK) if the device is not free and require a resend later.
- Copy data in RX_BUFFER to RXINPUT_BUFFER;
- Report MainLoop through ACTION_FLAG that there are data to be decoded in RXINPUT_BUFFER;
- If the packet is an IN Token, verifies through ACTION_FLAG if the answer is ready, encodes (in NRZI) and sends the entire contents of TX_BUFFER for control transfers or INT_TX_BUFFER for interrupt transfers, that must have been previously prepared (with bit stuff or CRC) for another routine inserted in MainLoop;
- Set ACTION_FLAG free when there's no more data to prepare/send;

And the MainLoop:

- Checks ACTION_FLAG and transfers the execution flow to proper treatment;
- Decodes data in RXINPUT_BUFFER to RXDATA_BUFFER.
- Calls VendorRequest (vendor/class), if it’s not a standard request, and ProcessOut to transfer control for the custom code (functionalities);
- Calls Main label to transfer control for custom code to do something periodically;
- Insert bit stuffing and CRC16 on device response (TX_BUFFER);
- Take care of all standard requests;

Data transfer process

Control Transfers:

The transfer process starts when the PIC receives the first positive going pulse of the Sync Pattern (coming from D+) on the external interrupt pin (INT/RB0) initializing the interrupt service (ISR). At the end of the Sync Pattern, ISR receives and immediately saves the bits sent by host in a buffer (RX_BUFFER) until the end of packet (EOP) is detected. Each bit is read in the receiving loop at the middle of the sample.

After the EOP detection, the type of packet that has just arrived is checked. If it's a token, its destination address is checked to guarantee if data is actually for the device. These two checks are done even before removing the NRZI encoding, because the device is subjected to a maximum time to send a response to the host. This maximum time, in our case, would be easily exceeded if we wait the completion of the decoding process. If the address doesn’t match, then the next incoming data packet is discarded.

Using the first pulse of Sync Pattern as reference, which is always a positive going pulse on the pin INT/RB0, is what makes possible these early findings (type and address of the packet), since the values ??of PIDs are fixed (obviously) and in the case of tokens, the first of the seven address bits immediately follows the last bit of the PID. Thus, we can make comparisons of data directly into NRZI in PID and ADDR fields on token packets, and in the PID field on data packet. Since the host can send new packets at anytime, the external interrupt of the PIC should always be ready to respond even if there is some internal processing in progress. When a processing is going on, the response to host shall be a NAK, indicating that the device is busy. Thus, host will send the packet again at a later time.

When the packet is a data packet, ISR copies RX_BUFFER to RXINPUT_BUFFER and ACTION_FLAG is filled with a value to inform MainLoop that there's data in RXINPUT_BUFFER to be decoded. Later, the decoding routine, on MainLoop, decodes (NRZI and bit stuffing removal) RXINPUT_BUFFER to RXDATA_BUFFER. So, RXDATA_BUFFER contains the decoded data. After decoding, ACTION_FLAG tells the MainLoop what to process depending on the type of token that data packet follows. For vendor/class requests MainLoop transfer control to custom code via VendorRequest or ProcessOut calling. MainLoop is also responsible for build answers for all standard requests.

If data available in RXDATA_BUFFER are from a SETUP token, the MainLoop (or custom code) must build the right response in TX_BUFFER if the request has been device-to-host, or perform some other operation on that basis if the request had been host-to-device. For OUT token data, processing is similar to host-to-device request. If it was a device-to-host request, then the host sends an IN token.

At this point a new interrupt starts, but this time after receiving the token, the ISR enters the sending loop of TX_BUFFER.

Interrupt Transfers:

The process is the same used in control transfers except that on interrupt transfers there are no setup stages, so no SETUP packets. IN requests are sent without any Setup before and answer for them must be on INT_TX_BUFFER.

The answer for interrupt transfer is often preprared by some code running in loop (Main label in main.asm), ever called by MainLoop. Out packets are also sent to ProcessOut, but ACTION_FLAG bit 2 will be setted for interrupt transfers. The data are avaliable in RXDATA_BUFFER as in control transfer.

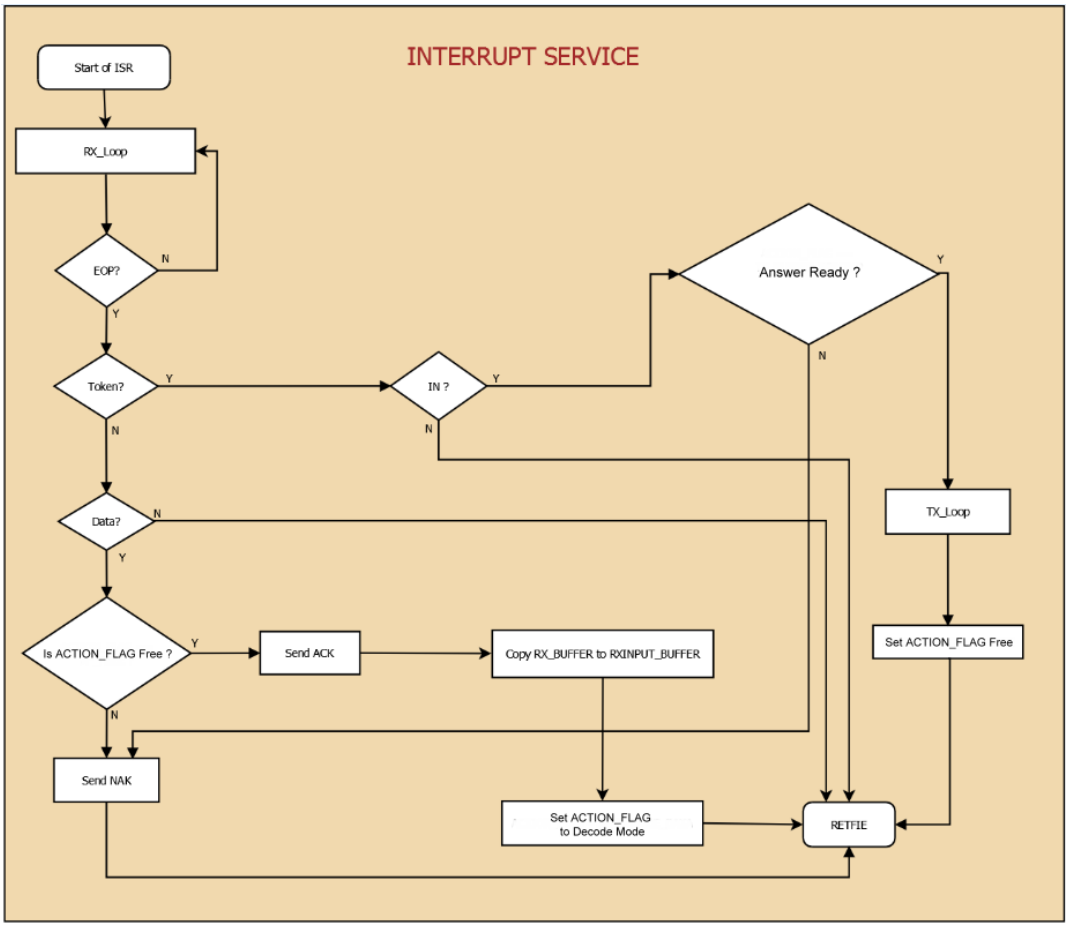

Image 1: Interrupt service routine flow diagram.

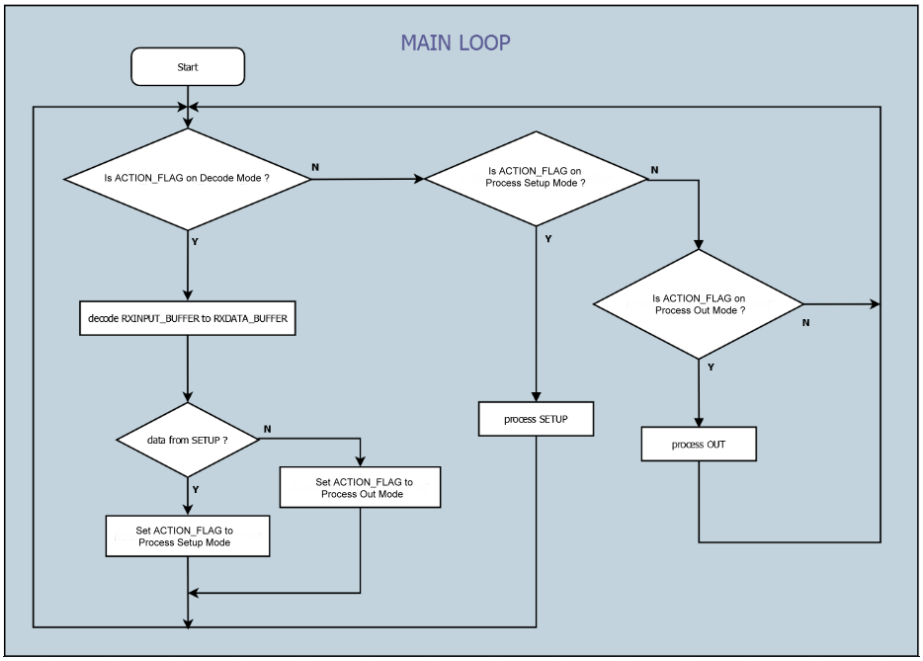

Image 2: MainLoop routine flow diagram.

Hardware

The basic circuit of 16FUSB consists in a typical PIC16F628/628A configuration using a crystal and a few more components as shown in the figure below. The circuit is feed by VCC (5V) from the USB port. The diodes D1 and D2 connected in the USB data lines are zeners that limit voltage coming from the PIC, since the USB specification states that these lines works with 3.3V signals. This should avoid potential problems with some more sensitive ports, although even without using zeners the circuit seems to work correctly in most cases. R3 and R4 limit the current in USB data lines. The resistor R1 is responsible for starting the device recognition by the host.

The external interrupt pin of the PIC (RB0) is connected to pin D+ USB port. This is where we receive the Sync Pattern (which initiates the interrupt routine) as well as all the others bits read by the PIC. D- is connected to pin RB1, being used almost exclusively for sending bits (along with RB0), unlike RB0 that receives and sends data bits. As the USB port works with differential pair, the lines D + and D- will always be in opposite logical levels, except on one occasion: the EOP (end of packet), where the two lines will be at logic zero (SE0). This ensures that the RB2 pin, connected to the differential pair through D3 and D4, will always be at logic one, except when the host informs the end of package. This trick allows us to check the EOP for every bit received just reading RB2 (btfss PORTB, 2). Remember that time is all we don't have here.

Since PIC's external interrupt input is connected through a Schmitt Trigger buffer, the input high voltage must be at least 0.8 x VDD on RB0. Some USB ports may not reach this value. To solve this, PIC VDD is connected through D5. It will reduce the VDD voltage in 0.6V and the 3.3V of USB port will be enough to reach the Schmitt Trigger threshold.

Is easy to see that, excepting RB0, RB1, RB2 and clock pins, all others I/O PIC pins will be free to be used in customized functionalities. The reference implementation Direct I/O make it clear.

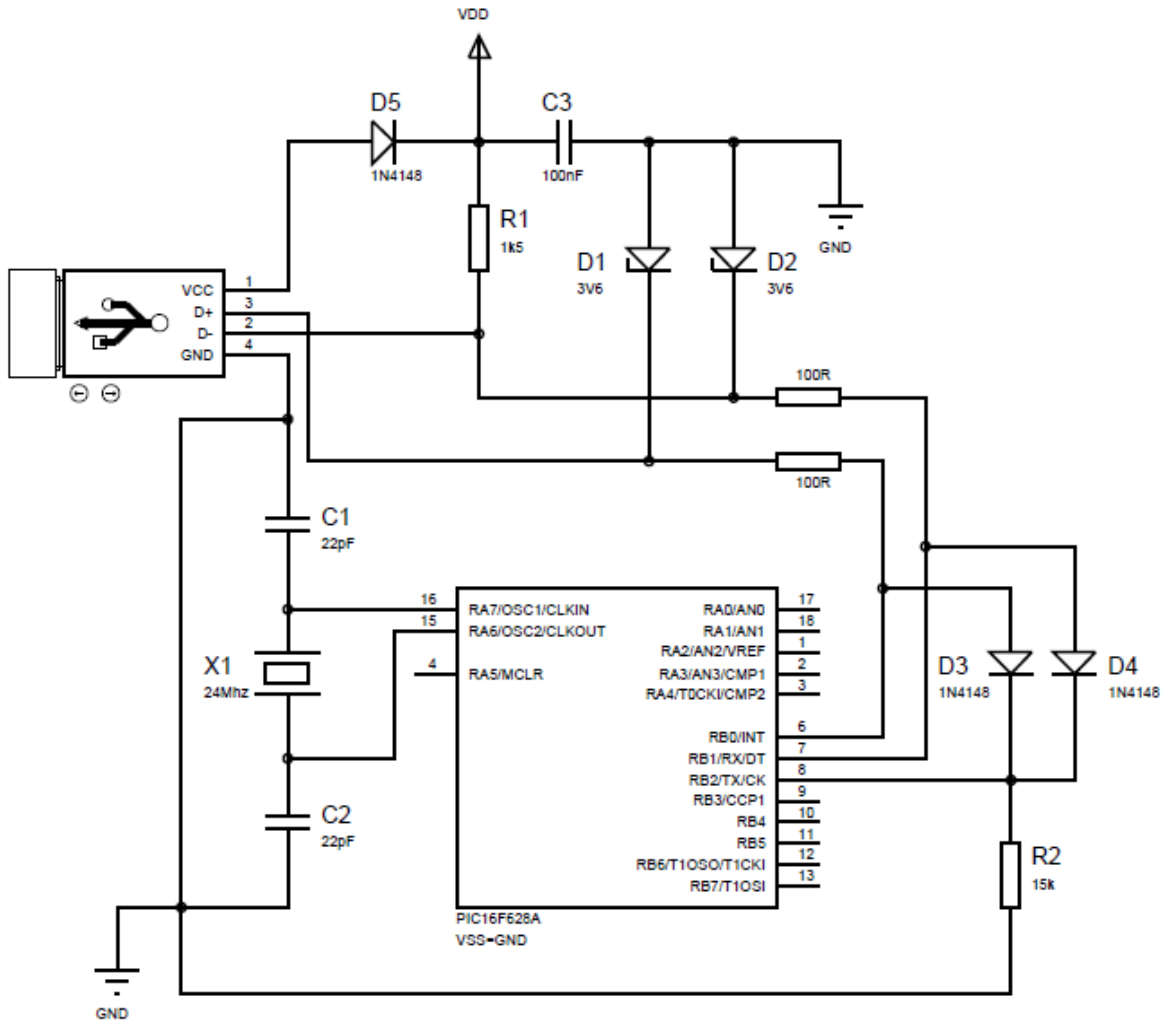

Image 3: Circuit diagram of 16FUSB.

Driver - libusb (HID disabled)

For communication between a USB device and the host computer, obviously, we need an application and a driver on the operating system to interface with this device. In the 16FUSB case, the driver used for both Windows and Linux is the libusb. The libusb is a driver/library that provides access to USB devices for user-level applications. With libusb is possible to exchange messages between your application and the USB hardware without the need to write a specific driver for your device. Its API is very straightforward and easy to use. Don't miss http://www.libusb.org/ to understand more about the project.

Image 4: Layers in communication via libusb.
USB hardware available in user mode.

Windows environment

For both runtime and development environment:

- Download the 16FUSB device driver:

  16FUSB_driver-libusb-win32-1.2.6.0.zip

- Install the device driver:

  Press **Win+R**, type '**hdwwiz**' and click '**Next**'. Choose '**Install the hardware that I manually select form a list (Advanced)**' and click '**Next**' again. Just leave the option '**Show All Devices**' and go '**Next**' button. Click '**Have Disk**' button and browse to the folder you extracted the driver then choose '**16FUSB**' and click '**Next**'. Windows will warn you about driver verification. Choose '**Install Driver Anyway**' to authorize driver install. Click on the '**Finish**' to close the wizard.

For development environment only:

- Copy the include file 'lusb0_usb.h' (found in the device driver package) into the Microsoft SDK's include directory.
- Copy the file 'libusb.lib' (also found in the device driver package) into Microsoft SDK's lib directory or into your IDE lib directory.
- Install the filter driver libusb-win32-filter-devel-xxxx.exe. The version of the filter driver must match the version of device driver. This step is optional.

HID support

16FUSB also supports HID communication. Just enable interrupt endpoints and HID option in def.inc file. Using HID we can discard libusb driver and use the operational system HID driver. There are many texts over Internet showing how to use HID through Windows and Linux. You can write you own Report Descriptor editing rpt_desc.inc. Report Descriptor size is adjusted in def.inc.

Just remember that HID can use both control and interrupt transfers, so it's important to know well the API you're using and how 16FUSB works with control and interrupt transfers.

Adding functionalities

The 16FUSB was developed as a core which, through an interface, is able to provide code that will support real functionality to the device. Without the code that add functionality, the firmware responds to all standards requests, fulfilling the entire protocol (including device enumeration), but with no practical application. Following we'll see how codes for custom requests can be added to the core.

First let's see how the source code is organized. The source files can be divided into two groups: core files and interface files.

Core Files:
It's the files that make possible all USB communication. No need to be modified to add features.

- **isr.asm:** File that contains the code responsible for interrupt service routine (ISR) described above.
- **usb.asm:** Contains the MainLoop code, also already described above. It makes the integration of all interface files with the firmware core.
- **func.asm:** It has general functions used by core.
- **stdreq.asm:** Implements the answer for all mandatory standard requests.

Interface files:

These files are the interface between the core and the user code. We shall edit them to add new functionalities.

- **main.asm:** This file allows you to declare initial settings and run some task in loop. On Init label you can put anything to do after PIC reset and before it starts accepting interruptions. On Main label you can run a periodic task (eg. put in INT_TX_BUFFER the state of PIC pins).
- **setup.asm:** All non standard requests are redirected to here via a “call” to the label VendorRequest. This is where we insert the code that handles custom control transfers (vendor/class). In short, the code flow will be on the label VendorRequest whenever we have control transfers, with data from the SETUP stage or data requested during DATA stage, ie, on device-to-host direction (IN).
- **out.asm:** When receiving data from an OUT token, flow is transferred to this file at ProcessOut label. We may understand this point as a callback function for OUT packets. An OUT packet will be available here in a control transfer DATA stage of a host-to-device request as well as OUT packets coming from interrupt transfers.
- **rpt_desc.asm**: Contains the HID Report Descriptor. Change this to customize the reports.

Working With Control Transfers (EP0):

Every control transfer starts with a Setup stage, composed by a SETUP token packet and a DATA0 data packet (see Image 5). On Table 1 we can see the Setup request format. Whenever a non standard request happens, the MainLoop calls VendorRequest. If you look at offset 0 description (Table 1), you’ll notice that VendorRequest will be called if value composed by bit 5 and 6 of bmRequestType field is not zero. These two bit defines if a request is standard, class or vendor request.

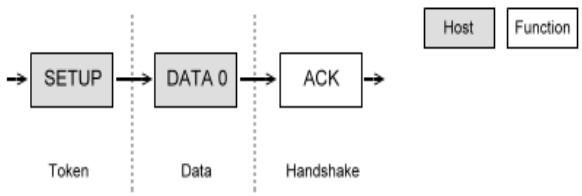

Image 5: Setup stage of control transfer

Offset	Field	Size	Value	Description
0	bmRequestType	1	Bit-Map	**D7 Data Phase Transfer Direction** 0 = Host to Device 1 = Device to Host **D6..5 Type** 0 = Standard 1 = Class 2 = Vendor 3 = Reserved **D4..0 Recipient** 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	bRequest	1	Value	Request
2	wValue	2	Value	Value
4	wIndex	2	Index or Offset	Index
6	wLength	2	Count	Number of bytes to transfer if there is a data phase

Table 1: Setup request format

In VendorRequest routine, setup data can be read in RXDATA_BUFFER, how we can see in picture below. We can read the offsets just using the form RXDATA_BUFFER+offset, ex: RXDATA_BUFFER+2 reads wValue low. The values in message fields is part of the developer's imagination.

OFFSET								
0	1	2	3	4	5	6	7	RXDATA_BUFFER
bmRequestType	bRequest	wValue low	wValue high	wIndex low	wIndex high	wLength low	wLength high	Setup Packet
byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7	Out Packet

Image 6: RXDATA_BUFFER map

All kind of requests always have a transfer direction: Device-to-Host - host expects get data from device; Host-to-Device - roughly, host sends data to device

- Device-to-Host request:

Transfer direction can be checked on bit 7 of bmRequestType field. Once we're on device-to-host request, we need to compose the answer, because host will send IN requests to get the device response. Thus, MainLoop delegates this to VendorRequest procedure. The answer shall fill 1 to 8 offsets of TX_BUFFER (TX_BUFFER+1 ... TX_BUFFER+8). We do not need fill the three others offsets, MainLoop will do it automatically for us.

OFFSET				
0	1 ... 8	9	10	TX_BUFFER
PID	data	CRC16 low	CRC16 high	Data Packet

Image 7: TX_BUFFER map

Low speed devices are limited to a maximum of 8 bytes packet size. If data stage have more than 8 bytes (wLength > 8), the transaction will be divided in multiple packets. In this case you shall use the FRAME_NUMBER register for check what part of answer host is asking.

Example:

wLength = 20

Host ask for first 8 bytes FRAME_NUMBER = 0
Host ask for second 8 bytes FRAME_NUMBER = 1
Host ask for last 4 bytes FRAME_NUMBER = 2

```
DeviceToHostRequest:
  movwf  FRAME_NUMBER,W
  xorlw  0x01
  btfsc  STATUS,Z
  goto   Answer_Frame1

  movwf  FRAME_NUMBER,W
  xorlw  0x02
  btfsc  STATUS,Z
  goto   Answer_Frame2

Answer_Frame0:
 movlw   0x55
 movwf   TX_BUFFER+1

 movlw   0xAA
 movwf   TX_BUFFER+2
 ...
 movlw  0x55
 movwf   TX_BUFFER+8

 return

Answer_Frame1:
 ...
Answer_Frame2:

 ...
```

- Host-to-Device request:

Non standard Host-to-Device request are also always treated firstly by VendorRequest, and only by it if we do not have data stage. For request with data stage, after Setup, host will send OUT packet and MainLoop will transfer control to ProcessOut (out.asm). At this point, RXDATA_BUFFER will reflect data stage content.

For transaction with more than 8 bytes, maybe you need to know which Setup request comes OUT packets. Any Setup information will only be available in VendorRequest. This is a good chance to save some information to be used later. For example, if our requests are based in bRequest field, at VendorRequest we can save it in other register and make a query for its value later in ProcessOut to know how to proceed with the data in RXDATA_BUFFER.

Working With Interrupt Transfers (EP1):

To handle interrupt transfer is a little different than handle control transfers. Interrupt transfers sends IN and OUT requests directly, without use a SETUP stage. Thus, when host wants to get some data from device it simple send a IN packet to EP1. If device have no data to send, a NAK is sent to host. Host may try again according to a defined timeout. When host needs to send data to device it simple send a OUT packet followed by data. As we have either SETUP nor STATUS stage in interrupt transfers, we can say that it's faster than control transfers.

- IN Interrupt Transfer:

The answer for IN interrupt transfers is made using the buffer INT_TX_BUFFER, the same way we do in TX_BUFFER (INT_TX_BUFFER+1 ... INT_TX_BUFFER+8). After build the message the code must call PrepareIntTxBuffer and put in INT_TX_LEN the message length. This routine will adjust data toggle, calculate CRC16, insert bit stuffing and set the ACTION_FLAG bit 5 (AF_BIT_INT_TX_READY) to inform there are bytes to send in EP1 interrupt endpoint. Once the message is ready, on the next host's poll (IN packet), the ISR sends the INT_TX_BUFFER and just after clear AF_BIT_INT_TX_READY. Thus, checking the AF_BIT_INT_TX_READY device knows if the buffer was sent and if a new message can be queued on buffer.

(main.asm)

```
; Always get pins state and put in buffer

Main:
   call    GetPinsState                ; Returns in W the PORTA pins state.
   movwf   INT_TX_BUFFER+1     ; Put pins state in buffer
   movlw   0x01                        ; Send one byte.
   movwf   INT_TX_LEN
   call    PrepareIntTxBuffer          ; Prepare buffer
   return
```

(main.asm)

```
; Get pins state and queue in buffer. Don't put new pins state in buffer until host receive the queued message.

Main:
   btfsc   AF_BIT_INT_TX_READY ; If there are pending message in buffer, return.
   return
   call    GetPinsState                ; Returns in W the PORTA pins state.
   movwf   INT_TX_BUFFER+1     ; Put pins state in buffer
   movlw   0x01                        ; Send one byte.
   movwf   INT_TX_LEN
   call    PrepareIntTxBuffer          ; Prepare buffer
   return
```

- OUT Interrupt Transfer:

Data from OUT packets on interrupt transfers arrives the ProcessOut label with data avaliable in RXDATA_BUFFER, just like in control transfers. So, how to know whether the packet is from control or interrupt transfer? All we need to do is checking ACTION_FLAG bit 2 (AF_BIT_INTERRUPT). It's cleared for control transfers and setted for interrupt transfers.

Working With HID:

HID specs defines use of control and interrupt transfers. IN interrupt transfer is mandatory, so one IN interrupt endpoint must be implemented. OUT interrupt endpoint is opitional. To enable HID and the EP1 IN, you must edit def.inc in the application folder. If you want to use OUT interrupt transfer with HID, just enable the EP1 OUT in the same file. Without the EP1 OUT, all HID reports sent by host to device will be via control transfer.

To send message for device via control transfer, HID uses Set_Report. To get messages from device, Get_Report. Depending on the API you're using, you may invoke functions thats guarantee a specific transfer type. On Windows 7, for example, you may use HidD_GetInputReport and HidD_SetOutputReport to generate Get_Report and Set_Report, respectively. The ReadFile function always retrives a buffered message obtained by host via an IN interrupt transfer. WriteFile function send message for device using interrupt transfer if EP1 OUT is available, otherwise it use control transfer.

To handle interrupt transfers with HID is same as described above. Control transfers with HID will be avaliable in VendorRequest (setup.asm) like any other class/vendor request. On host-to-device messages case (eg. Set_Report), the OUT packet will be avaliable in ProcessOut like any other OUT packet.

When HID is enabled, we have only messages defined by HID specs. Thus, when a control transfer arrives VendorRequest label, we know thats is a HID message. All we have to check is the kind of HID request and/or the report ID. See below the main fields for Get_Report and Set_report messages. To configure the Report Descriptor edit the rpt_desc.inc file. The length of the descriptor must be configured in def.inc.

Request	bRequest	wValue (high byte, low byte)	wIndex	wLength	Data Stage
Get_Report	0x01	report type, report ID	interface	report length	report
Set_Report	0x09	report type, report ID	interface	report length	report

16FUSB RAM use:

The core of 16FUSB only uses Bank 0 including some positions of shared area (mirrored). On Bank 0, you can use free positions of shared area and LOCAL_OVERLAY section. Using LOCAL_OVERLAY you have the advantage of don't worry about memory banks.

Bank 1 and 2 are totally free and can be all used by the functionality. Don't forget to use "banksel" directive to select the right bank for your register if you use other banks than Bank 0. Whenever you need to save information to query later, you may save it in an overlay section. Notice that in this case you cannot use LOCAL_OVERLAY as it is a temporary area, ever overwritten by core.

Example of an overlay section:

```
(setup.asm)
;Bank 1 registers
MYAPP_OVERLAY   UDATA_OVR      0xA0
MYREG         RES    1
(out.asm)
;Bank 1 registers
MYAPP_OVERLAY   UDATA_OVR      0xA0
;MYREG here will access the same content that MYREG in vreq.asm
MYREG         RES    1
```

You also may simple use an udata section and just leave the linker decide the registers addresses. It still can be accessible by other objects if you use global directive. Anyway, banksel directive still must be used as you don't know where linker will put your variable.

Example:

```
MYAPP_VARIABLES UDATA
MYREG         RES    1
```

Reference Implementations: Direct I/O and Direct I/O HID

Direct I/O

As a reference functionality I chose to implement a simple application that sends and receives bytes directly into the PIC pins via control transfer. This implementation selects 8 pins to send/receive 1byte, two control pins with read/write access and a read only status pin.

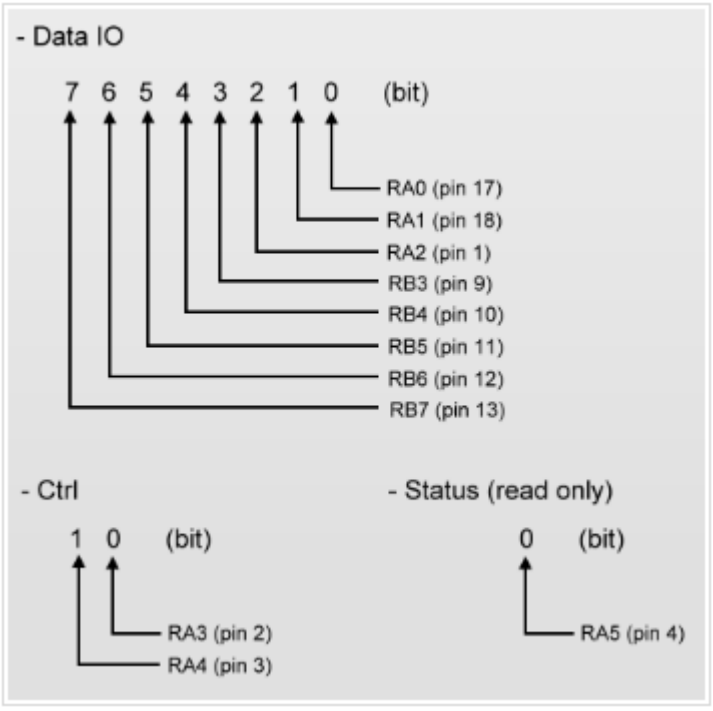

Image 8: Data, status and control pins in Direct I/O implementation.

The table below shows the requests of the Direct I/O, their respectives bytes in the Setup packet and what should be present in the data stage if it happens.

REQUEST	DESCRIPTION	bmRequestType	bRequest	wValue	wIndex	wLength	Data
DirectIO -> WriteByte	Write 1 byte into data port	01000000B	0x01	0x0001	Byte	Zero	No data stage
DirectIO -> ReadByte	Read 1 byte from data port	11000000B	0x01	0x0001	Zero	1	Byte Read from data port
DirectIO -> WriteLowNibble	Write 4 bits from low nibble of data port	01000000B	0x01	0x0002	Byte	Zero	No data stage
DirectIO -> WriteHighNibble	Write 4 bits from high nibble of data port	01000000B	0x01	0x0004	Byte	Zero	No data stage
DirectIO -> ReadLowNibble	Read 4 bits from low nibble of data port	11000000B	0x01	0x0002	Zero	1	Low nibble from data port
DirectIO -> ReadHighNibble	Read 4 bits from high nibble of data port	11000000B	0x01	0x0004	Zero	1	High nibble from data port
DirectIO -> WriteCtrl	Write 2 bits into control port	01000000B	0x01	0x0008	Byte	Zero	No data stage
DirectIO -> ReadCtrl	Read 2 bits from control port	11000000B	0x01	0x0008	Zero	1	Two bits read from control port
DirecIO -> ReadStatus	Read status pin	11000000B	0x01	0x0010	Zero	1	Bit read from status port

Table 2: Direct I/O messages

Reading the source code of the Direct I/O is the best way to understand how to make a custom implementation. If you are a PIC developer, a good analysis of the code as well as a reasonable study of the USB protocol should be enough to start developing your own functionality for the 16FUSB.

Anyway, I draw attention here to some important points:

RXDATA_BUFFER contains the data sent from the host and using offset we can access all 8 bytes sent by the Host, if this is the case. We see this in line 112 and 121 of the code snippet below. According to the table of requests, WriteByte sends the byte in wIndex to be written in PIC pins, ie, at offset 4, RXDATA_BUFFER+4. If we want to know the total size of bytes in the data stage, that is, read the value of wLength, for example, we would do it through RXDATA_BUFFER+6.

```
98   DIO_WriteByte:
99       ;Put data port in output mode
100      bsf     STATUS,RP0
101      movlw   B'00000111'
102      andwf   TRISB,F
103      movlw   B'11111000'
104      andwf   TRISA,F
105      bcf     STATUS,RP0
106
107      ;Put value in RB3-RB7
108      movlw   B'00000111'
109      andwf   PORTB,W
110      movwf   TMP
111      movlw   B'11111000'
112      andwf   RXDATA_BUFFER+4,W       ;wIndex Lo
113      iorwf   TMP,W
114      movwf   PORTB
115
116      ;Put the last bits values in RA0-RA2
117      movlw   B'11111000'
118      andwf   PORTA,W
119      movwf   TMP
120      movlw   B'00000111'
121      andwf   RXDATA_BUFFER+4,W       ;wIndex Lo
122      iorwf   TMP,W
123      movwf   PORTA
124
125      return
```

Image 9: Direct I/O WriteByte request code.

Finally, it's important to understand the use of TX_BUFFER to assemble the message. Is easy to imagine that the bytes must be placed using offsets in form TX_BUFFER (TX_BUFFER +0), TX_BUFFER+1, TX_BUFFER+2, TX_BUFFER+8. For the custom code we must observe that the first byte to be sent should be placed in TX_BUFFER+1, the second in TX_BUFFER+2 and so on. This is because in TX_BUFFER+0 we have the PID of the request, which is adjusted by the PreInitTxBuffer subroutine in core. In the case of Direct I/O, as we have only one byte to be sent, we use only TX_BUFFER+1.

```
288     ;Prapare TX_BUFFER
289     ;TMP must have the byte with answer.
290     DIO_PrepareAnswer:
291         movf    TMP,W
292         movwf   TX_BUFFER+1
293
294         return
```

Image 10: Answer code for ReadByte, ReadLowNibble, ReadHighNibble, ReadCtrl and ReadStatus requests.

The application that runs on the host was written in C for Windows using the 'Microsoft Visual C++ 2010 Express'. It explores all read and write commands present in the Direct I/O. To exchange messages with the firmware, the application basically uses the libusb's 'usb_control_msg' function, since this sample uses only control transfer. The API version used by Windows libusb (libusb-win32) is 0.1. It's not compatible with version 1.0. See the [libusb-win32 documentation].

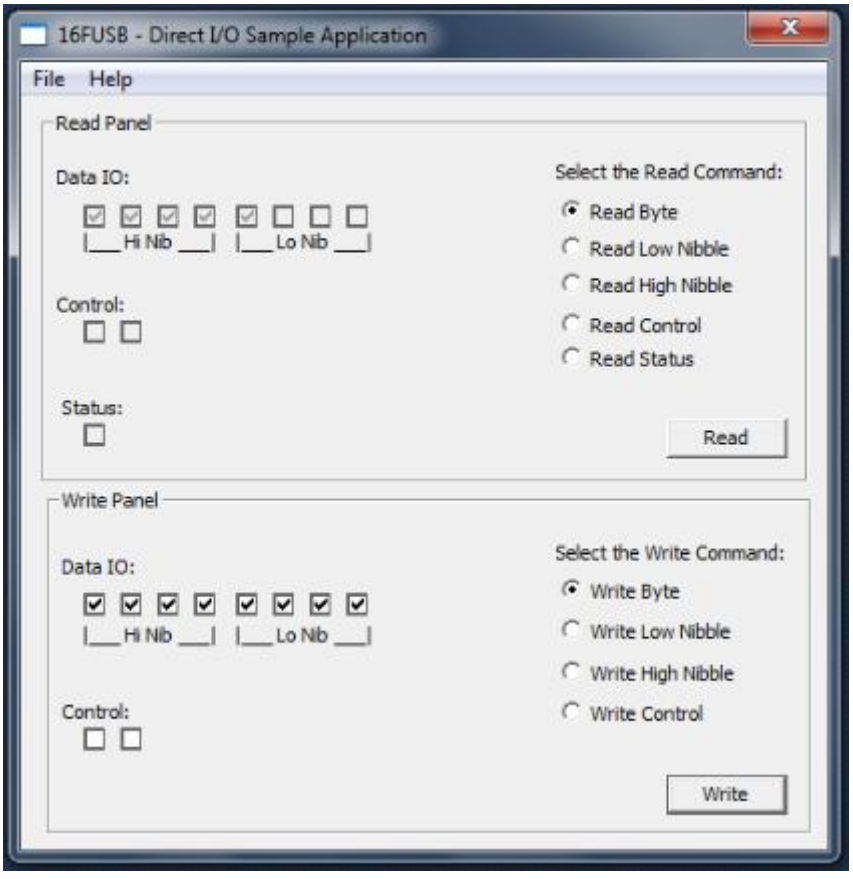

Image 11: Sample application that explores Direct I/O commands.

Direct I/O HID

This version of Direct I/O uses HID with IN and OUT interrupts endpoints. Of course, libusb is not used here. Remove libusb associated with 16FUSB device if you have installed it before. You may do it using Windows device manager.

The application set RB4-RB7 pins by user click, and read a switch state connected trhough RA0 whenever it changes. Connect the switch between RA0 and VDD.

The read and write operation are made using ReadFile and WriteFile functions, respectively. Only default report was used (no reportID) by this example, so address is omitted in out packet. Thus, the byte sent by host will be avaliable in RXDATA_BUFFER+0. The ReadFile is called by a separeted thread that capture input report whenever device send it.

This application version was written in C++/Cli also using the 'Microsoft Visual C++ 2010 Express'.

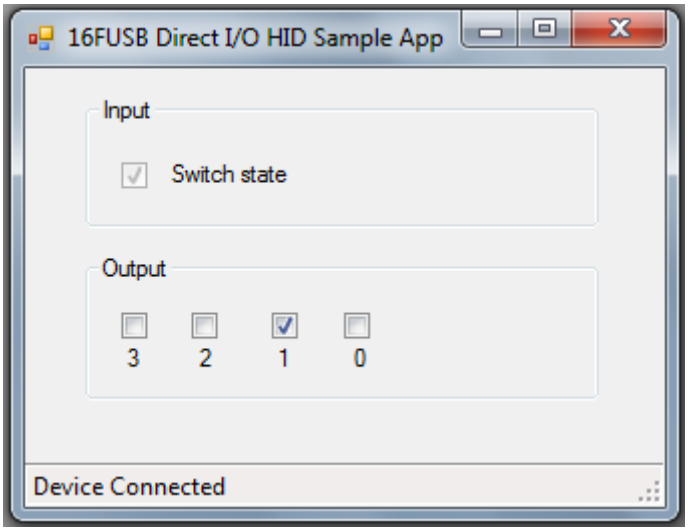

Image 12: Sample application of Direct I/O HID commands.

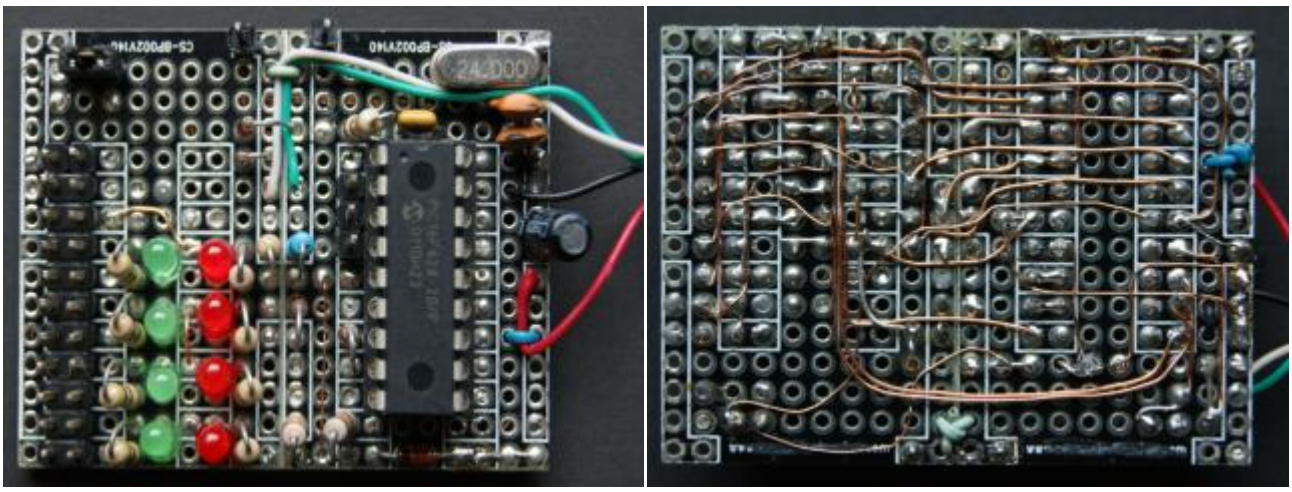

Photo 1: Test board of Direct I/O

Downloads

Firmware

- 16FUSB-bare-src-1.2.tar.gz - Core source code, base to new functionalities

- 16FUSB-dio-src-1.2.tar.gz - Direct I/O reference implementation source code
- 16FUSB-dio-1.2.hex - Direct I/O reference implementation binary

- 16FUSB-dio_hid-src-1.2.tar.gz - Direct I/O HID reference implementation source code
- 16FUSB-dio_hid-1.2.hex - Direct I/O HID reference implementation binary

Schematic

- 16FUSB_schematic-1.2.pdf - Basic circuit

Driver (win32)

- 16FUSB_driver-libusb-win32-1.2.6.0.zip - libusb device driver for 16FUSB
- libusb-win32-devel-filter-1.2.6.0.exe - libsusb filter driver

Direct I/O Sample Application (win32)

- 16FUSB_dio_sample_application_win32-src-1.0.zip - 'Visual C++ 2010 Express' project of Direct I/O sample application.
- 16FUSB_dio_sample_application_win32-bin-1.0.zip - Direct I/O sample application binary.

Direct I/O HID Sample Application (win32)

- 16FUSB_dio_hid_sample_application_win32-src-1.0.zip - 'Visual C++ 2010 Express' project of Direct I/O HID sample application.
- 16FUSB_dio_hid_sample_application_win32-bin-1.0.zip - Direct I/O HID sample application binary.