

18F4550 USB Interface

Contributed by Evan
Sunday, 06 August 2006
Last Updated Monday, 02 June 2008

This article contains information about getting a USB interface working between an 18F4550 PIC and a PC using C#.NET 2.0, using the Microchip-supplied driver mpusbapi.dll. In this article, the PIC is configured using some (slightly modified) demo code originally for the PICDEM FS USB demo board, in which it is configured as a generic class device, with one IN + OUT interrupt endpoint. The device is set up with a bootloader, allowing the PIC to be programmed through software after the initial programming. For this article, it should not be necessary for the reader to know much about the details of USB in general, but I would still highly recommend Jan Axelson's book "USB Complete " to anyone interested in going further with USB. I owe a big thanks to Mat at PICCoder.co.uk for his own PIC USB tutorial, which is what got me started in the first place. The Microchip PIC 18F4550 microcontroller is a popular device for interfacing with USB. Aside from being a very full-featured PIC (by the current standards) it of course has a full-speed USB 2.0 interface built in. Since much of the existing USB example code is for this device, it makes a good starting point. The information in this article is expected to be usable on other similar USB-capable PICs (such as the lower-pin-count 18F2550, or the cheaper versions of both chips with less memory space), but conversion is up to the reader. Microchip produces a full-speed USB demo board. The amount of hardware required to get a working USB interface going is very minimal. Here is a small prototyping board that I made; there are a few surface-mount parts (decoupling capacitors and LED resistors) on the underside, but in all, you can see that it is quite simple. The board includes the PIC, reset/bootload buttons, USB connector, 4 USB LEDs, a power LED, a spot for an optional voltage regulator for self-powered operation, and the obligatory decoupling capacitors. You can download the schematic and PCB layout of the board below, made in Eagle 4.16 by cadsoft.de. One thing of note is that you don't have to use a 20MHz crystal; you can use a number of speeds (4, 8, 12, 16, 20, or 24 MHz) by choosing the right division ratio in the configuration settings later on. [DOWNLOAD PCB + SCHEMATIC FILES](#) (Cadsoft Eagle 4.16 format)

For the purposes of getting started, this circuit can also very easily be built on a breadboard, as you can see. The USB connector is connected through some short wires to a 4-pin header to plug into the breadboard. I also included an ICSP connector as you can see, this isn't shown on the schematic because it was only used for the initial programming of the bootloader, and thus isn't very necessary; just pull the chip and program it externally. There are a few things you need to pay attention to. First, you MUST include the capacitor on pin 18 (Vusb). This is required by the internal USB voltage regulator. The actual value is not critical, I believe the actual recommended value is 0.33uF, so use what you've got that's close. Second, make sure you keep the wires between the USB socket and the PIC pins as short as you can, and make sure to put a decoupling capacitor across the power pins of the USB socket as close to it as you can; I used 0.1uF. Once you have that constructed, it's time to get some firmware on it. In the interest of convenience, it is tremendously beneficial to use a USB bootloader to upload code to the PIC. Once the bootloader is programmed onto the chip, no separate programmer is needed for development, and the PIC can be programmed quickly and with minimal effort. So, it's time to get the bootloader set up! {mospagebreak title=Page 2 - Burning in the Bootloader}The bootloader firmware used in this example is the standard Microchip bootloader, as a precompiled HEX. For the PC software, I have included my own bootloader application (which has its own article with more information), but it can also be used with the Microchip standard demo bootloader application. **IMPORTANT** You must download Microchip's code package for their FS USB demo board, MCHPFSUSB, which is accessible from the FS USB Demo Board site. The file you need is an executable installer, which (by default) installs to C:\MCHPFSUSB. The link name changes occasionally as they update the code, presently the file you want is titled "MCHPFSUSB v1.2 USB Framework and Examples". It contains a lot of important stuff: the code and hex for the standard bootloader firmware, their PC demo bootloader application, the windows device driver, mpusbapi.dll, and a number of example programs for the demo board, which form the basis of the demo code in this article. [DOWNLOAD CODE PACKAGE FOR THIS ARTICLE](#) (last updated 2/16/08 - Updated code to implement new USB interface DLL with 'safe' code) [DOWNLOAD CODE FOR PICUSB.DLL](#)

This is the code for the DLL file used in the above demo app, in case you want to see the details of how it works. Last updated 3/1/08Now, you need the bootloader firmware. I have included it in the code package as bootloader.hex - but FYI, it is exactly the same file as the official microchip bootloader firmware, normally located at C:\MCHPFSUSB\fw\Boot_output\MCHPUSB.hex, from the MCHPFSUSB demo code package linked above. Getting the bootloader set up is relatively easy, however it does require that you have a working PIC programmer that is compatible with the 18F4550. I used a JDM-style cheap serial programmer, with the software WinPIC800. You need to make sure you get the right configuration settings. The default oscillator is 20MHz, so if you choose a different speed make sure to choose the right clock divider when doing the config. The necessary settings are shown here, taken from the bootloader code project in MPLAB.Also, here are the same settings in WinPIC800. The easiest way of verifying you have the right settings in any given programming application is to compare all the hex values to those from MPLAB. Once you have successfully programmed the bootloader firmware onto the PIC, you can do your first test of the USB connectivity. When you plug the PIC in to your computer via USB, not much will happen, but if you hold the bootload button and press the reset button, two of the LEDs should begin flashing, and the computer should detect it as a new USB device, and ask you for a driver; direct it to the MCHPFSUSB driver and all should be OK, and you will see it in device manager.The next step is to open up the bootloader software. This is the program you will use to download application firmware to your PIC.

Open up the bootloader application, and when the PIC is ready it should read "PIC detected in BOOT mode". From here you can read what's on the PIC, erase it, or issue an execute command, which just resets the PIC so it goes into user code. Next, it's time to put some useful code on it! {mospagebreak title=Page 3 - USBDemo PIC Firmware and C# Software} The demo firmware I have put together for this article is built from the microchip example code from MCHPFSUSB. For the purposes of this article, you don't necessarily need to examine and understand the code; there's certainly a lot there to understand. For more information on it, I suggest you later check out my Understanding the Microchip USB Firmware article. I will give more useful information regarding the firmware later in the article. Right now, all you need to do is use the bootloader software to load the demo firmware into the PIC. Open the hex file, which is in the PIC Firmware folder of the code package for this article, and press "Program". If all is well, and you see a 'Write succeeded', then you can move on. If the write fails, reset the board and try again. If all else fails, try the bootloader application PDFSUSB, included in the MCHPFSUSB package from Microchip, it may be more reliable than mine. Once you have the firmware loaded, you can reset the board to begin executing it. When you do this, windows is going to see it as a different device, so it's going to ask you for drivers again the first time. Just give it the same driver as before. Now that the device is running the demo code and has been recognized by windows, you're ready to open up the demo software and test it out. The executable file is in C# Demo Software\USBDemo\bin\Debug in the code package. Run it, and you should see the following: Checking the boxes and pressing the first button will allow you to set the states of LEDs #3 and 4 connected to the PIC (the two that aren't constantly flashing). The other example, the counter, counts a value from 0 to 255 by sending it to the PIC, which increments it and sends it back, demonstrating useful two-way communication. Please note that the speed at which it counts is not representative of the real maximum possible throughput rate, such as the software overhead with such small packets, and the default interrupt transfer interval of the PIC. {mospagebreak title=Page 4 - Understanding the Demo Firmware} Just to clarify, the firmware here sets up the PIC as a custom device using the Microchip driver, and configures it as what they refer to as a "generic" class device. In addition to the required endpoint 0, it also has endpoint 1 enabled as an IN and OUT interrupt endpoint, which is the endpoint that it uses for communication with the functions USBGenRead and USBGenWrite. These functions, and the initialization function USBGenInitEP, are found in usbgen.c. By default, endpoint 1 is configured with a bInterval value of 32, or a max latency of 32ms between transfers. Packet Format Each packet consists of a command byte (corresponding to any of the commands in the enum in user.h) followed by 0 or more data bytes, depending on the command. <COMMAND><DATA><DATA>...etc... Looking at the switch statement cases in ServiceRequests in user.c, you can see the rather simple method by which received packets are accessed; in general, it is done as a simple byte array, where dataPacket._byte[0] is the command byte, and subsequent indices represent the data bytes (if any). To transmit data back to the PC in response to a command, the data is loaded into the same buffer (datPacket._byte[]), the value of the byte variable 'counter' is set to the total number of bytes to be transmitted (including the command and all data bytes) and at the termination of the switch statement, the following code will perform the transmission via USBGenWrite if there is data to be sent:

```
if(counter != 0)
{
    if(!mUSBGenTxIsBusy())        USBGenWrite((byte*)&dataPacket,counter);
```

```

}

//end ifOne question you might ask is "what if I want to send data from the PIC to the PC without the PC having to deliberately send it a command to request it?" (in other words, asynchronously sending data from PIC to PC) Well, this isn't something covered by this demo firmware and software, but is covered in another of my articles. Suffice it to say that it's not extraordinarily complex, but it can require some work that is beyond the scope of a plain introductory article (such as adding another endpoint, modifying descriptors, and on the PC side, running the asynchronous receive methods in a separate execution thread) Software SideWell, now that I have covered what is happening on the firmware side, I'll give some insight into the software side. Interfacing with the PIC would be pretty complicated if we had to make low-level access to the device driver, but thankfully microchip supplies a DLL that simplifies this interface, mpushapi.dll. Anyone who is competent in C++ should take a look at the source code of it, which is included as part of the MCHPFSUSB software package from Microchip. The heart of the interface in my USB Demo C# program is the usb_interface class in the PICUSB.DLL file. It basically imports the mpushapi.dll functions, and provides a few methods as a simpler wrapper around them. The complete documentation (MSDN-style) of the PICUSB DLL file is available in HTML format , or in a single compiled help file if you prefer to download. In the interest of making the interface as C#-friendly as possible, I made the function EasyCommand, which accepts and returns more "normal" data types for C#, instead of the pointers and DWORDs of the imported DLL functions. public uint EasyCommand(byte Command, int rlength, byte[] data, out byte[] dataout) You pass in the command, data (if any), and it attempts to do a transmit/recieve, returning any response in 'dataout'. The function return value indicates success or failure. Note that what is returned in 'dataout' is an array containing the whole packet, including both command and data. As far as what is going on in the demo program, you can see that it is quite simple. When you click button 1, it sends two packets, with the command UPDATE_LED (0x32), followed by the number of one of the LEDs (3 or 4), followed by the desired state, read from the corresponding checkbox. The resulting code is as simple as:private void button1_Click(object sender, EventArgs e)
```

```

{
    temp = new byte[2];
    byte[] temp2;
    if(checkBox1.Checked){temp[0]=1;}
    if(checkBox2.Checked){temp[1]=1;}
}
```

```
usbi.EasyCommand(0x32, 1, new byte[] { 3, temp[0] }, out temp2);  
usbi.EasyCommand(0x32, 1, new byte[] { 4, temp[1] }, out temp2);
```

```
}
```

The code for the counter feature is about as simple, albeit slightly more spread out because it's using a timer and a button, but in essence it sends a packet with the command 0x33, which the PIC increments and sends back; it then takes the second byte of the received packet (the data byte) and stores it back into the variable to be sent next time. The relevant code, located in the timer1_Tick method, is as follows: (see the full code for more detail if you wish)

```
byte[]  
temp;   
usbi.EasyCommand(0x33, 2, temp, out temprx);  
temp[0] = temprx[1];
```

Note that to compile a project using the usb_interface class, you have to enable "allow unsafe code" in the build options. If anyone more advanced in the use of C# has ideas about any ways to change this (if it is even possible, given the nature of the DLL functions being used), please let me know!

I have now updated the code to use marshalling when importing the Microchip DLL functions, so no unsafe code is necessary. {mos_sb_discuss:8}