

PIC USB Bootloader using Microchip firmware and C#

Contributed by Evan
 Wednesday, 23 August 2006
 Last Updated Sunday, 29 June 2008

Note: This article is kind of a mess at present, as I just recently did a significant overhaul of my PIC USB interfacing code, including the bootloader, so I need to update the article more to reflect the new code, when I have time. Presently, things that are crossed out are outdated info that I haven't yet updated or removed. Bear with me!

At the time I started this project, I was unable to find any decent information whatsoever about the PC side of bootloading. Over at piccoder.co.uk, there is a nice tutorial that gets you started with bootloading, with the microchip firmware on the PIC side, and an example C# GUI application that demonstrates the use of his custom bootloader DLL, which makes it incredibly easy. Mat has provided his bootloader DLL freely for non-commercial use, with a small nag screen popup.

However, I still wanted to do it myself, for two main reasons: First, I am very interested in knowing how it is done, and since there is nearly no information on the PC side of things, it seems the best way is to dive in and try to figure it out myself. Second, I would like to have working PC-side bootloader code to use in my own projects (personal or commercial), but with full access to (and understanding of) the source code so I could adapt it as I needed. And, of course, if I succeed I'd like to make the information available for others who are interested, to help combat some of the lack of information on the subject that has frustrated me so much.

Mat at piccoder said he basically reverse-engineered the necessary PC-side interface by starting with a look at the PIC firmware.

Download the software package for the PICDEM Full Speed USB evaluation kit . Install, and in the folder C:\MCHPFSUSB\fw\Boot\ will be the relevant source code of the PIC-side firmware. Now in the course of trying to get what I need from the firmware, I am focusing on only the things that are related to the PC interface; there are a bunch of functions that the PIC is doing for initializing and managing the USB connection, but the only things I need to know about for this are the things pertaining to the actual commands the PC software can send to the PIC.

```
In main.c, you will see that when the PIC is in bootload mode, it sits in this while loop after initialization: while(1)
{
  USBDriverService(); // See usbdrv.c
  BootService();      // See boot.c
} //end while
```

The function `USBDriverService()` seems to be related to the general USB operation of the PIC, not to the bootloading functionality. So, the function of interest is `BootService`, which is (as indicated) found in `\system\usb\class\boot\boot.c`. Of interest in that function is the switch statement that processes the possible commands. The options call the appropriate functions, which are seen in the same file, and all seem pretty simple and self-explanatory. The cases (seen in the switch statement, or as given here, from the enum that shows their numerical values as transmitted as well):

```
READ_VERSION   = 0x00,
READ_FLASH     = 0x01,
WRITE_FLASH    = 0x02,
ERASE_FLASH    = 0x03,
READ_EEDATA    = 0x04,
WRITE_EEDATA   = 0x05,
READ_CONFIG    = 0x06,
WRITE_CONFIG   = 0x07,
UPDATE_LED     = 0x32,
RESET          = 0xFF
```

Since that is all of the possible commands the PC can issue to the bootloader firmware, it seems as though it should be a pretty simple process, besides sorting out just how to format the data packets. Reading and writing the HEX files on the PC could prove to be similarly challenging.

Just a note, in the "C:\MCHPFSUSB\fw\Boot_output" folder you will find the HEX file for the bootloader, which is used with the rest of this article.

Next up: Monitoring PC-PIC communication using an existing bootloader application to figure out the protocol.
 {mos_sb_discuss:8}

{mospagebreak title=Page 2 - Monitoring Communication}

Monitoring Communication

I used a program called "USB Monitor" from HHD Software to monitor the communications between the PIC and the PC during bootloading using the Microchip-supplied demo program, PDFSUSB. This allowed me to see everything the PC-side program needed to do in order to perform the bootloading functions with the PIC, using the standard Microchip bootloader firmware. This particular USB monitor program did a good job of formatting the packets so they were easy to follow. Unfortunately, it's commercial software and costs \$65 which is a bit much for a hobbyist who will only be using it very occasionally, but they offer a 14-day (max 100 sessions) free trial, which should be enough to finish this project.

The hardware is the same barebones setup I used in my initial experimentation, seen in the first article , schematic shown below:

I just used the buttons to put the PIC in bootload mode, started up PDFSUSB.exe, and started playing around with it with the USB monitor running. Anyway, there are a lot of transactions, but here is what I picked out of it so far:

Packet Format:

Based on the communication I've monitored, I believe the packet format is:

<command><length><address (LSB)><address><address(MSB)>

The address is transmitted with the least significant byte first, followed by the next most significant byte, followed by the most significant byte. Note that for all program memory and EEPROM read/writes, the MSB is 0 (at least for the 18F4550) because these memories do not extend past address 0x7FFF. Only when reading/writing configuration bytes is the MSB non-zero.

See the below communication transcripts for examples.

Read Operation:

First, the program reads all the PIC program memory, using the command READ_FLASH (0x01)

```
PC>"01 3B 00 00 00"  
PIC>"01 3B 00 00 00 xx xx..." (59 data bytes)  
PC>"01 3B 3B 00 00"  
PIC>"01 3B 3B 00 00 xx xx..." (59 data bytes)  
etc...
```

Each response from the PIC is an echo of the 5 bytes from the PC, followed by 59 data bytes of program memory starting at the specified address. (59 + 5 = 64 bytes, which I believe is the maximum for one packet)

After the program finishes reading all the program memory, it reads the EEPROM, using READ_EEPROM (0x04), in exactly the same way it read the program memory:

```
PC>"04 3B 00 00 00"  
PIC>"04 3B 00 00 00 xx xx..." (59 data bytes)  
etc...
```

After reading the EEPROM, the program then reads the special ID locations, 0x200000 to 0x200007. For whatever reason, it does this by reading them one byte at a time:

```
PC>"06 01 00 00 20"  
PIC>"06 01 00 00 20 xx"  
etc...
```

Then it reads the actual configuration bytes, which reside in special locations 0x300000 to 0x30000D, 0x3FFFFE, and 0x3FFFFFF.

```
PC>"06 01 00 00 30"  
PIC>"06 01 00 00 30 xx"  
PC>"06 01 01 00 30"  
PIC>"06 01 01 00 30 xx"
```

...

```

PC>"06 01 0D 00 30"
PIC>"06 01 0D 00 30 xx"
PC>"06 01 FE FF 3F"
PIC>"06 01 FE FF 3F xx"
PC>"06 01 FF FF 3F"
PIC>"06 01 FF FF 3F xx"

```

Erase Operation: (command = 0x03)

```

PC>"03 01 00 08 00"
PIC>"03"
PC>"03 01 40 08 00"
PIC>"03"
PC>"03 01 80 08 00"
PIC>"03"
PC>"03 01 C0 08 00"
PIC>"03"
PC>"03 01 00 09 00"
etc...

```

Since the bootloader takes up the space before 0x800, it looks like each command erases 0x40 (64) memory locations, starting with 0x800. When it finishes erasing all the program memory, it verifies by simply reading it all; it's the same as the read operation listed above, except it only reads the program memory, not the EEPROM or the configuration.

Write Operation:

Well, this one is a pain, so bear with me. To give you an idea of how much USB traffic this involves, when I exported capture log from the USB monitor into simple HTML format, the file size was 3.52 megabytes, and contained 7661 lines of text, meaning somewhere around 1700 transactions to sort through.

The first thing that happens is that it performs an erase of the program memory, just as in the erase operation above, except it does not perform a full read to verify; instead it verifies as it programs in the new code.

Next, it repeats one process, in which it writes and then verifies one chunk of data. Each chunk is 16 bytes; I'm not 100% sure of the reasoning for this, but that also happens to be the number of data bytes seen on each line of a standard PIC HEX file, which may be the reason. It does not attempt to program any memory locations before 0x0800, even if they appear in the hex file, because that is reserved for the bootloader code.

```

PC>"02 10 00 08 00 xx xx..." (16 data bytes)
PIC>"02"
PC>"01 10 00 08 00"
PIC>"01 10 00 08 00 xx xx..." (16 data bytes)
PC>"02 10 10 08 00 xx xx..." (16 data bytes)
PIC>"02"
PC>"01 10 10 08 00"
PIC>"01 10 10 08 00 xx xx..." (16 data bytes)
etc...

```

After that, it does the same kind of write-verify process to write the EEPROM:

```

PC>"05 10 00 00 00 xx xx..." (16 data bytes)
PIC>"05"
PC>"04 10 00 00 00"
PIC>"04 10 00 00 00 xx xx..." (16 data bytes)

```

Then, it does one erase-write-verify to write the ID locations (starting at 0x200000):

```

PC>"03 01 00 00 20"
PIC>"03"
PC>"02 10 00 00 20 xx xx..." (16 data bytes)
PIC>"02"
PC>"01 10 00 00 20"
PIC>"01 10 00 00 20 xx xx..." (16 data bytes)

```

One confusing thing here is that when doing the read operation seen above, it used the command READ_CONFIG (0x06) to read the ID locations; here, it is using ERASE_FLASH, WRITE_FLASH, and READ_FLASH. I can understand

the erase, since there is no command to erase config locations, but why use read/write flash instead of read/write config? Perhaps the functions are interchangeable for these locations? I guess this is one of those things where I'll just have to keep my mouth shut and do it the way they did it, and then go back and test my theory later once I have a working program.

Finally, it does one write-verify on the configuration bytes, 0x300000-0x30000D (note that 0x3FFFFE-0x3FFFFF are not written as these are read-only values representing the PIC hardware device ID and revision number)

```
PC>"07 0E 00 00 30 xx xx..." (14 data bytes)
PIC>"07"
PC>"06 0E 00 00 30"
PIC>"06 0E 00 00 30 xx xx..." (14 data bytes)
```

Note that it only writes 14 data bytes this time, as opposed to the fixed 16-byte packets it used for everything else, for whatever reason.

```
Execute:
PC>"FF"
PIC resets
```

Just like a regular reset, the PIC goes into user mode and begins executing the user program, as long as the bootloader button isn't held down at the time of reset.

Next up: Starting to put together a windows bootloader application. {mos_sb_discuss:8}

{mospagebreak title=Page 3 - Dealing with HEX files}

The bootloader application on the PC side needs to be able to perform 2 main tasks: read and write HEX files, and perform bootloading tasks by communicating with the PIC. Since you need to have something to send or receive when attempting to implement the bootloader, I chose to start by first tackling the HEX file side.

Dealing with HEX files

Hex files are a reasonably easy format to deal with, but it still took some time for me to put together the necessary code. If you're very good with C#, you could probably do it a whole lot faster.

Rather than explain all about the hex file format, please visit <http://www.keil.com/support/docs/1584.htm> for information.

As you can see, the 2-byte address given on each line is only the lower 16 bits of the full address being referenced on the PIC. The higher bits are accessed using the "extended linear address" command, so often the first line of a hex file will be::020000040000FA

Which sets the upper 2 bytes of the full 32-bit address to 0x0000. Note that the 2 bytes set with this command apply to all subsequent regular data transmissions, until they are changed again. Since the PIC program memory is 0x000000 to 0x007FFF, this command is issued one time, and applies to all the data that goes in program memory. Some compilers/assemblers do not include that, I guess zero is assumed for the initial value. It will be issued again when the ID locations (starting at 0x200000) and configuration bits (starting at 0x300000) need to be accessed, if the hex file contains that information.

```
Following that, the rest of the hex file typically looks something like::0800000083011F308A00042F68
:103E080083010A168A152F2F83120313C8000C1E6C
:103E18000B2F48089900080083120313C801C90131
etc...
:103E580003180034142F0A308316031399002430F2
:103E68009800903083129800640010273B3A031999
:02400E00763FFB
:00000001FF
```

Where that last line (of transaction type 0x01) is the end of file marker.

I have implemented the HexInterpreter class to deal with reading and writing HEX files. This class is covered in the Bootloader Interface DLL code documentation, or you can dig into the code if you want to know more.

The OLD version of the example project is available here. I don't know why you'd want it (it's kind of a mess) but it's here

until I finish phasing it out.

DOWNLOAD OLD EXAMPLE PROJECT {mos_sb_discuss:8}

{mospagebreak title=Page 4 - Bootloader Interface Class}

I have implemented the required functionality and wrapped it up in a DLL so that it is pretty painless to simply use in your application. The complete documentation (MSDN-style) for the Bootloader Interface DLL is available online in HTML format , or as a downloadable compiled help module. From a user's standpoint, that documentation should cover all you need to be able to implement bootloader functionality in an application.

The Bootloader Interface class extends my PICUSB class, which in turn wraps around the USB functions provided in Microchip's MPUSBAPI.DLL. Essentially this class is a nice wrapper around all the 'dirty work' of dealing with HEX files and reading/writing/erasing the PIC, providing a number of very easy-to-use methods for all the necessary bootloader tasks.

Example Bootloader Application:

I originally implemented a simple bootloader application for this project, and while it worked fine, it was lacking some usability, so I have made this one which is a bit fancier, but a lot more usable for day-to-day use. Implementing a really simple version should be pretty straightforward based on the class documentation, if you wish to do so. Download Bootloader Interface Class Code Package

Includes C# project for building BootloaderInterface.DLL, as well as the DLL, and the compiled code documentation.

Last Updated 3/1/08

Download Bootloader Application Code Package

Includes C# project for bootloader app, including required DLL's.

Last updated 3/1/08 Download Bootloader Application (executable and required DLL's only)

Includes application executable and required DLL's (no code).

Last update 3/1/08

The "real" version of the USB bootloader app has a lot more features than the previous, oversimplified one, making it suitable for actual use rather than just as a clunky demo. Time-consuming read/write/erase tasks are done in a separate execution thread to prevent freezing up the GUI, and enough options are added to cover normal use.

This version also includes some support for a modified bootloader firmware done by Rob Edwards, which can be downloaded below. (His firmware has the ability to enter bootload mode on reset based on either the normal button presses, or on a 'flag' value written to a particular EEPROM location, which means your user program can write the value to EEPROM and reset, entering boot mode entirely through software) I've added a button to the bootloader software to send the command 0xFE to the PIC in user mode. This assumes that you've implemented the kind of generic-class device interface that I use in my other tutorials, and that in your user firmware you handle the command 0xFE by writing the value 0xFF to EEPROM location 0xFE (the bootload flag).

If you're using a different interface in your user code, this button in the software won't do you any good, but the EEPROM-flag bootloader firmware will still work just fine, you'll just have to code in your own method to order it into boot mode.

DOWNLOAD ROB EDWARDS' MODIFIED BOOTLOADER FIRMWARE (HEX)

(Last updated 3-28-07)