

# Understanding the Microchip USB Firmware

Contributed by Evan  
Saturday, 23 September 2006  
Last Updated Tuesday, 30 January 2007

(in progress)

As a precursor for understanding the workings of the Microchip USB firmware, it's a good idea to have a basic grasp of the USB protocol itself. Jan Axelson's book *USB Complete* is a good resource, and the web article "USB in a Nutshell " is a good free resource.

The Microchip USB firmware (for the PICDEM full-speed USB demo board, but used frequently for DIY projects as well) is written and organized in a manner that it is not very easy to just take a quick look and see exactly what's going on. Everything is arranged across 9 source files, and 15 header files. There are enough subroutines to make your head spin. Things are broken down into pretty logical chunks, with an ample amount of commenting accompanying each subroutine, however it's still hard to pick out what is being run, and when. First off, you need to get an idea of the "core" execution path during normal operation, with the device already enumerated and configured and sitting there running user code and servicing requests. This description is assuming that the device is running the 'generic' class firmware, as in the Microchip firmware I started from. {nomultithumb}

That simple flowchart shows the basic operation, where the black loop is the top level (in main.c), and blue and red are 2 levels of subroutines executed from within the functions in the top loop. First I'll mention that the standard firmware is using the "generic" device class, which includes a set of functions for formatting/interpreting the generic command+data packets used for communication.

The function blocks shown can be described as follows: (Note: a lot of this is just the function overviews from the comments in the code)

USBTasks (main.c)

Executes USBCheckBusStatus and USBDriverService. (Service loop for USB Tasks)

USBCheckBusStatus (usbdrv.c)

This routine enables/disables the USB module by monitoring the USB power signal.

USBDriverService (usbdrv.c)

This routine is the heart of this firmware. It manages all USB interrupts. Determines the state of the device.

Device state transitions through the following stages:  
DETACHED -> ATTACHED -> POWERED -> DEFAULT ->  
ADDRESS\_PENDING -> ADDRESSED -> CONFIGURED -> READY

ProcessIO (user.c)

This is the entry point into user routines

BlinkUSBStatus (user.c)

This is responsible for flashing the USB status LEDs, to indicate the current state of the device.

ServiceRequests (user.c)

This is where any incoming commands/data are handled, and returned data (if any) prepared and sent. This function (and the corresponding enum defining all the possible commands, found in user.h) need to be edited if you add or modify

any commands.

USBGenRead (defined in usbgen.c)

This performs a USB packet read; as evidenced by the "Gen" in the name, this is part of the "generic" device class code.

(handle command)

Not a subroutine per se, but if a packet is received with USBGenRead, this is where it is handled, with a switch statement with a case for each defined command from the enum in user.h.

USBGenWrite (defined in usbgen.c)

After a read and handling the command, if anything needs to be sent back to the host, that happens here, transmitting the data loaded into the buffer in the previous step.

{mospagebreak title=Page 2 - Descriptors and Endpoints}

If you want to add or modify endpoints on the device, you need to know of several places to make changes. First, there are a couple of good sections of comments in the code; check out the comments in usbmmap.c, and usbdsc.c. In each case, the existing endpoints and descriptors should be enough of a guide as to what exactly to add or change. The changes you need to make are as follows:

usbdsc.c

This is where the descriptor information is, which is reported to the PC. The important parts here are setting the number of endpoints in the interface descriptor, and adding or modifying the endpoint descriptors.

usbdsc.h

Here you need to add or modify the endpoint descriptor(s) in the configuration descriptor.

usbmmap.c

This file maps the memory space used by the endpoints. Here you need to add or modify the buffer space declared for each endpoint you plan to have.

usbmmap.h

Here, you have to declare those buffers as externs.

usbcfg.h

Here you will find a number of #define statements mapping intelligible names to certain endpoint parameters. These will generally contain the name of the device class; for example, the #define's for the 'generic' device class start with 'USBGEN\_'. These #define's are referenced in several places throughout the firmware.

usbgen.c

This is where the functions important to your user-controlled endpoints lie. Technically, this file contains the functions for the 'generic' class, and thus you should probably keep your own functions in a separate file (especially if you're ditching the 'generic' class entirely) but for simple experimentation it's probably easiest to just add or modify functions here. The functions here are for initializing the endpoint(s), reading, and writing. For any endpoints you add, you'd need to put some initialization code in the first function, modeling what's already in there. And for each IN or OUT endpoint, you need a corresponding read or write function. The existing 'generic' class functions should be a good enough guide for a basic case.