Using the Cypress tools effectively to harness the power of the EZ-USB FX

Kyumsung Lee
Nikhil Jayakumar

Univ of Colorado at Boulder

Table of Contents:

Introduction

What is USB:

USB (Universal Serial Bus) has revolutionized the way we connect to and talk to peripherals. It solves many problems that computer users have when installing a new peripheral on their PC. It allows a user to plug in a device (or unplug it) while the PC is running. Installing USB based hardware is usually very simple and there is no need nay more to worry about things like DMA and IRQ conflicts or switching jumper settings. Another useful feature is the speed. While a USB host controller can support up to 12Mbps serial data rates, it can also operate at a lower rate of 1.5Mbps. This reduces the cost (incurred from needing to make special cables and connectors to support high speed data transmission) of devices that do not require such high speeds. USB hubs can be added to expand the USB port and allow adding numerous devices to one machine. Another feature that makes USB a good serial protocol is that it uses differential signaling. This also makes the USB cables cheaper since we only need 4 wires – 2 for the differential data lines and 2 for power supply and ground.

Cypress EZ-USB FX:

The EZ-USB FX from Cypress Semiconductor Corporation takes care of all the USB communication and the complications of the communication protocol used and makes writing code for a device simple. It is a soft (RAM based) solution that allows implementation of a variety of USB peripherals. It operates at 3.3V which can be easily derived from the USB bus which is at 5V.

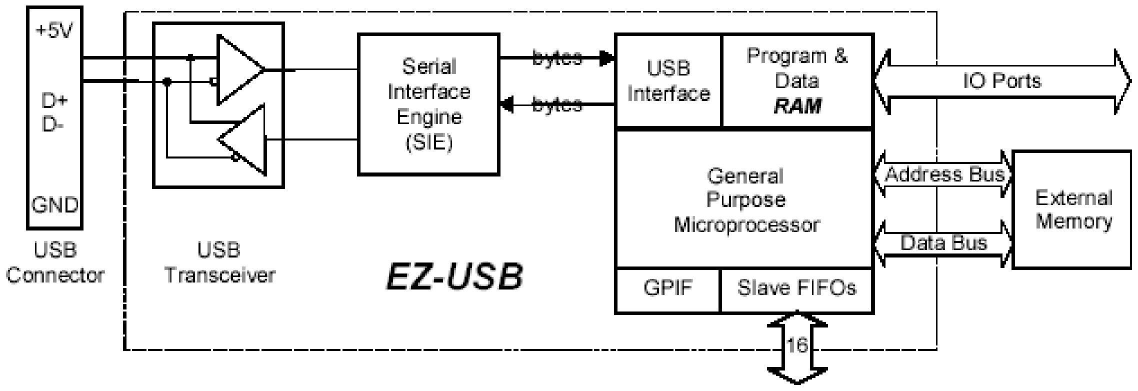


Figure 1: EZ-USB FX block diagram

The figure above shows the block diagram of the EZ-USB FX (from the Cypress EZ-USB FX Technical Reference manual).

The EZ_USB FX has an 8051 uP at its core with a Serial Interface Engine (SIE) to handle the USB communication. This makes it simple to use since we only have to program the 8051 inside (using assembly). Also there are tools such as the Keil development tools that simplify the coding aspect.

The 8051uP used in the EZ-USB FX is an enhanced 8051. It has the following feature(as mentioned in the Cypress Technical Reference Manual):

- 4clocks/cycle as compared to 12 clocks/cycle for the standard 8051.
- 48-Mhz clock
- DMA for 48MB/s memory-to-memory data transfers and dual data pointers for improved XDATA access.
- 2 UARTS.
- 3 counter-timers.
- Expanded interrupt system.
- 256 bytes of internal register RAM.
- Standard 8051 instruction set (makes coding easy for someone familiar with 8051).

Since the EZ-USB FX is soft, it can identify itself to a host computer as any kind of USB device that is desired. At first, the FX loads the 8051 code and USB descriptor tables (that define what kind of device is being plugged in) into the RAM onboard the chip (usually all this is stored on a EEPROM). Once the code and descriptor tables have been downloaded, the FX disconnects itself and comes back on as the new device defined in the descriptor tables. This process is called ReNumeration.

All USB devices have “endpoints” which are actually FIFOs which sequentially empty/fill with USB bytes. USB can uniquely address 32 endpoints (16 IN and 16 OUT). There are 4 USB endpoint types: Bulk, Control, Interrupt and Isochronous.

Bulk endpoints are unidirectional and so the IN and OUT endpoints have different addresses. The EZ-USB FX has 14 Bulk endpoints (1-IN through 7-IN and 1-OUT through 7-OUT). Each endpoint has a 64-byte buffer.

Control endpoints are bi-directional which transfer control information (such as setup data) to and from the USB device. The EZ-USB FX has one control endpoint at endpoint zero.

Interrupt endpoints are almost the same as bulk endpoints. The only difference is that they have a polling interval byte in their descriptor so that the host knows how often they should be serviced. The EZ-USB FX has 14 such endpoints (EP1-IN through EP2-IN and EP1-OUT through EP2-OUT).

Isochronous endpoints are used to deliver high bandwidth, time critical data over USB (for example streaming voice or video). The EZ-USB FX has 16 such endpoints. 1024 bytes of memory are available to these endpoints and this memory should be divided among them. But the EZ-USB FX actually has double that memory available to allow the use of double buffering.

In addition to the standard interrupts on the 8051, the FX adds seven interrupt sources. Three of them (INT4, INT5# and INT6) are available on device pins. The other four are used internally. The USB core automatically supplies the jump vectors for these interrupts.

The EZ-USB FX has 4 resets. (Power-On Reset, USB Bus Reset, 8051 Reset, USB Disconnect/Re-connect).

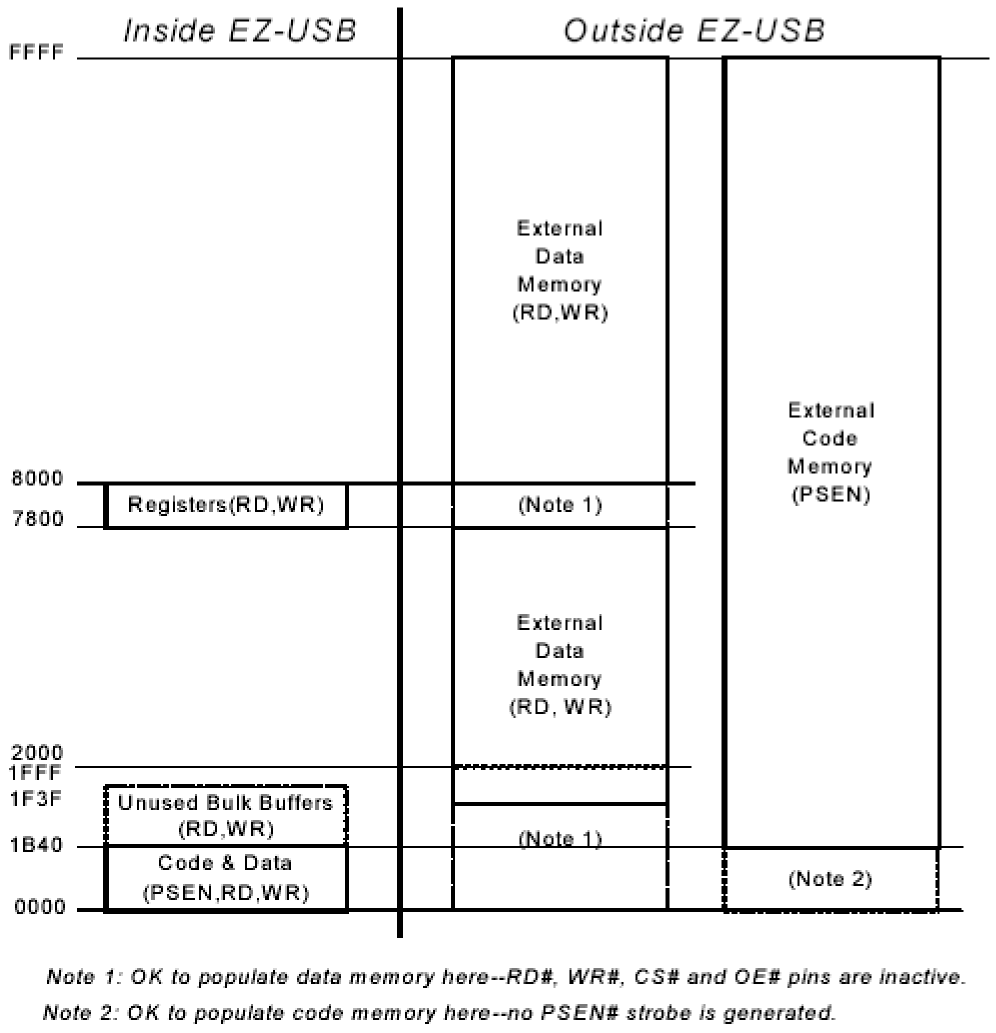
The FX chip has four 64-byte FIFOs and a General Programmable Interface (GPIF) that provide a very flexible and high-speed interface to numerous peripherals.

The EZ-USB FX chip comes in 3 packages, 128-pin, 80-pin and 52-pin. The 128-pin package 40 I/O pins, a 16-bit address bus and an 8-bit data bus. In the smaller packages the address and data line and some of the I/O pins are not brought out. The 80-pin package has 32 I/O pins and the 52-pin package has 18 I/O pins.

An important aspect of the EZ-USB FX chip that has to be kept in mind while programming is the memory map used. The EA (External Access) pin control placement of the code. If the EA pin is tied low, the USB core internally Ors the PSEN and RD signals of the 8051 so that the code and data now share the 0x0000-0x1B3F memory space. If EA=1, all the code is stored in external memory. The internal block 0x7B40-0x7FFF is reserved for the EZ-USB FX buffers and registers and this is aliased at the memory space 0x1B40-0x1FFF. Since future versions may not have this aliasing, Cypress advises user to uses only the space at 0x7B40-0x7FFF to access the buffers and registers of the EZ-USB FX.

NOTE: *The 52-pin version internally ties the EA pin low.*

When EA = 0, the code/data memory is internal at 0x0000-0x1B3F. External memory can be added in the entire region from 0x0000-0xFFFF, but it appears in the memory map only at 0x1B40-0xFFFF. The USB core inhibits any #PSEN strobes to external memory in the space from 0x0000 to 0x1B3F. Thus external memory can be easily added from 0x0000-0xFFFF without requiring any decoding to disable it between 0x0000 and 0x1B3F. It is automatically done by the EZ-USB FX.



Inside EZ-USB
Outside EZ-USB
FFFF
8000
7800
2000
1FFF
1F3F
1B40
0000
Registers(RD,WR)
Unused Bulk Buffers (RD,WR)
Code & Data (PSEN,RD,WR)
External Data Memory (RD,WR)
(Note 1)
External Data Memory (RD, WR)
(Note 1)
External Code Memory (PSEN)
(Note 2)
Note 1: OK to populate data memory here--RD#, WR#, CS# and OE# pins are inactive.
Note 2: OK to populate code memory here--no PSEN# strobe is generated.

Figure 2: EZ-USB FX memory map with EA=0

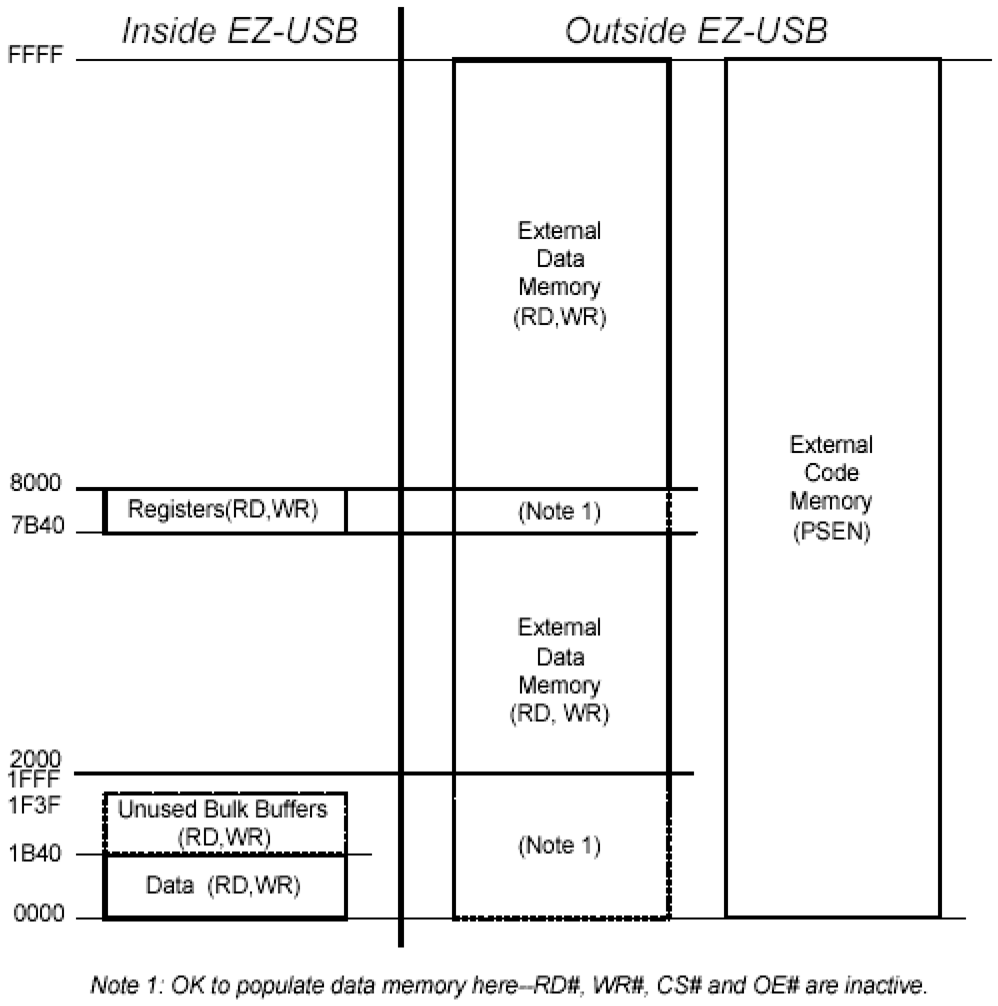

Inside EZ-USB
Outside EZ-USB
FFFF
8000
7B40
2000
1FFF
1F3F
1B40
0000
External Data Memory (RD,WR)
Registers(RD,WR)
(Note 1)
External Data Memory (RD, WR)
Unused Bulk Buffers (RD,WR)
Data (RD,WR)
(Note 1)
External Code Memory (PSEN)
Note 1: OK to populate data memory here--RD#, WR#, CS# and OE# are inactive.

Figure 3: EZ-USB FX memory map with EA = 1

With EA = 1, all code memory is external and the internal RAM on board the chip is used as data memory.

Using the Tools

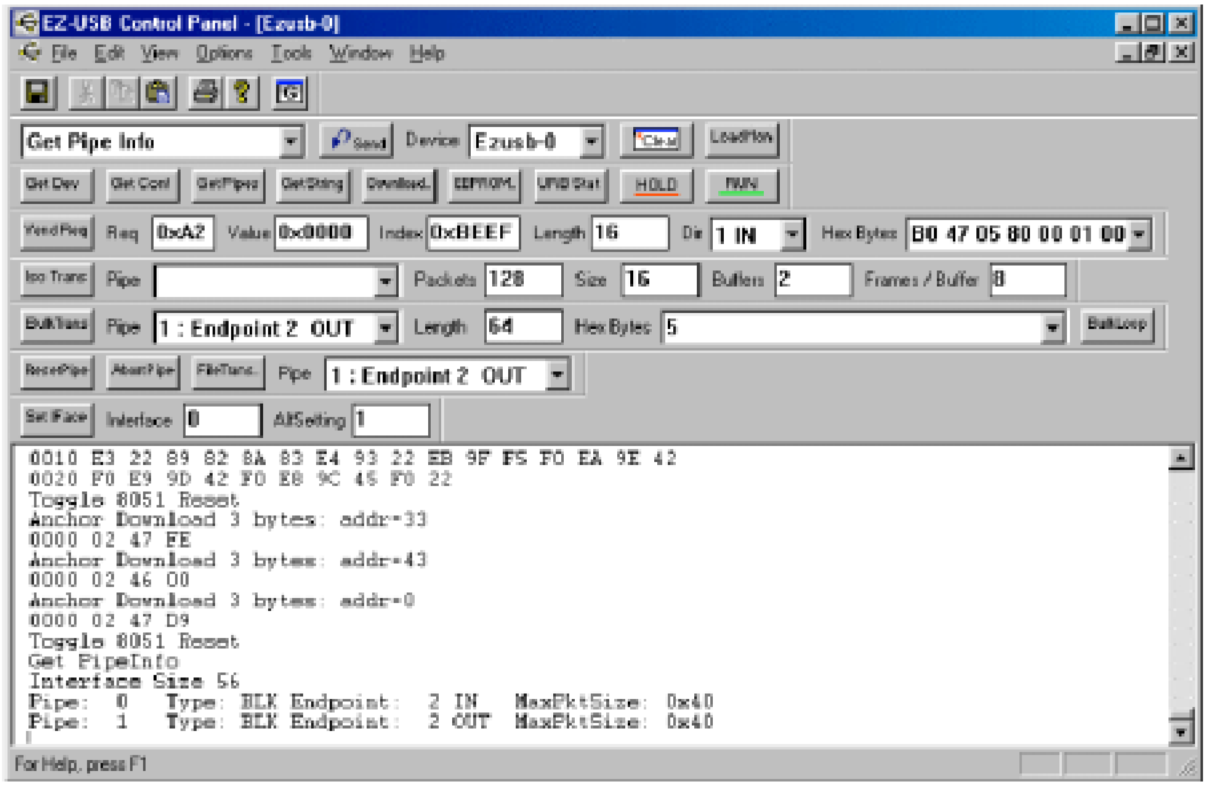
The EZ-USB Control Panel:

The EZ-USB control panel provides an interface that helps one perform basic operations such as:

- Downloading the 8051 code to the on-chip external RAM on the EZ-USB chip over USB.
- Programming an EEPROM on an I2C bus. (More on this later)

- Getting Descriptors.
- Sending and Receiving Bulk data from the screen or a file.
- Sending and Receiving Isochronous data.
- Performing loop back tests.

A screen shot of the EZ-USB Control Panel is shown below.



Get Pipe Info
Device Ezusb-0
Req 0xA2 Value 0x0000 Index 0xBEEF Length 16 Dir 1 IN Hex Bytes B0 47 05 80 00 01 00
Pipe Packets 128 Size 16 Buffers 2 Frames / Buffer 8
Pipe 1 : Endpoint 2 OUT Length 64 Hex Bytes 5
Pipe 1 : Endpoint 2 OUT
Interface 0 AltSetting 1

```
0010 E3 22 89 82 8A 83 E4 93 22 EB 9F F5 F0 EA 9E 42
0020 F0 E9 9D 42 F0 E8 9C 45 F0 22
Toggle 8051 Reset
Anchor Download 3 bytes: addr=33
0000 02 47 FE
Anchor Download 3 bytes: addr=43
0000 02 46 00
Anchor Download 3 bytes: addr=0
0000 02 47 D9
Toggle 8051 Reset
Get PipeInfo
Interface Size 56
Pipe:  0   Type: BLK Endpoint:  2 IN   MaxPktSize: 0x40
Pipe:  1   Type: BLK Endpoint:  2 OUT  MaxPktSize: 0x40
```

For Help, press F1

Figure 4: EZ-USB Control Panel

The first time that the control panel is launched, the user is first asked to choose a target (EZ-USB & FX/ FX2/ SX2). In our case of course we choose the first option. This setting (target device) can be changed even after the control panel is launched, by selecting a device from the dropdown menu next to “Target”. At startup the control panel first checks to see if there are any devices on the USB bus that it can communicate with (by checking for the correct VID/PID combination). If it can’t find any such devices it presents the user with only the menu bar and one application tool bar. To get the interface as shown in the screen shot above, the user should first hook up the device to the USB hub/port and click on the “Open All” button in the application toolbar or select “File/Open All devices” from the menu bar. If for some reason this doesn’t work, first check if the board is getting power (either external or bus as the case may be), then check if there is an EEPROM on the I2C bus. This EEPROM may be automatically uploading its own VID/PID. Remove this EEPROM and try again (disconnect and reconnect).

There are Unary operations (operations requiring no parameter except for perhaps a target file) available to the user as buttons on the toolbar. Given below is a summary of those operations:

- Get Device Descriptor: Get Device Descriptor standard call.
- Get Configuration Descriptor: Get Configuration Descriptor standard call.
- Get Pipes: Uses "Get Pipe Info" IOCTL to get the pipe/endpoint configuration information from the driver. The driver maintains this information in memory, so no USB traffic is actually generated from this command.
- Load Mon: Load monitor code into the USB device for use with a debugger later on.
- Clear: Clear the contents of the output buffer.
- Get String: gets string descriptors (it is hard coded to get the strings with index 1 and 2). This is normally the Manufacturer string index and Product string index (which is seen when you plug in the device).
- Download: Download a target (*.hex) file.
- Re-Load: Re-Load the last target file.
- EEPROM: Select EEPROM file to download file contents to EEPROM.
- URB Stat: gets the most recent USB Error status reported. Multiple USB errors (Indicated by the generic “Endpoint Error”) map to a single IOCTL Error. The IOCTL errors are normally reported directly by the Control Panel, but such a USB error will now indicate “Endpoint Error” instead. Pressing URB stat will manually request the last URB (USB Request Block) error. Please be aware that some types of errors (like a bad parameter in the IOCTL) would fail before getting to USB, so the error code would be meaningless. Pressing “URB Stat” will give you the value of the last USB error, and will decode the error if possible.

- Hold: Put 8051 Reset into Hold state.
- Run: Put 8051 Reset into Release state.

The Standard USB requests are also contained in a dropdown menu next to the "Send" button. This allows users to select a USB request and send it by clicking the "Send" button. *Note: It was noticed that this didn't work function in a version of the control panel that we used.*

Clicking on "Help/Control Panel User Guide…" provides a useful document about the control panel explaining each of these functions and how to use the control panel. It is probably the best document available to help one explore all the features of the control panel. However the essential functions that will be used most often are explained below.

For downloading code (8051 code) into the RAM onboard the EZ-USB FX chip, click the "Download" button. This lets you browse through folders and select a hex file. Once loaded the program downloaded is automatically run. If you want to halt the execution of the program, you can click "Hold". Clicking "Run" will resume execution of the code.

Note that once your code has finished executing the program counter continues to increment till it reaches the end of the address space after which it loops back and executes the code again. Use 'while' loops or 'goto' statements to prevent this from happening and to have control over the location from which your next instruction is fetched.

Some code that you write can use bulk/ isochronous transfers. The control panel provides two buttons "Bulk Trans" and Iso Trans" that can help debug such code. Before using these buttons, the user has to first click "Set Interface" then click "Get Pipes". The user can now select the from the endpoints available, choose the length of data (in bytes) that he/she wants to send/receive and the data (if data is being sent) and then click "Bulk Trans"/"Iso Trans". When receiving data, if there is no data at the output endpoint buffer or it is busy or stalled, the control panel can sometimes hang. In such cases simply unplug the device and close the control panel. Bulk transfers are limited to 64 bytes at a time. There is also a "File Trans" button that allows the user to send/receive data from/to a file. The user can select either isochronous or bulk endpoints as targets for the file. If an OUT pipe is selected, the user is prompted for a file, which is then sent out through the selected endpoint.

Such a function is useful when trying to simulate large chunks of data being sent. If an IN pipe is selected, the file will be created and filled with data coming in the pipe.

Code and descriptor tables can be downloaded onto a EEPROM on a I2C bus if the address lines are set up correctly (see section on "Using an EEPROM for boot loading"). This is done with the EEPROM button, which prompts the user to select an EEPROM image file, and then sends the data across the I2C bus and burns the EEPROM.

Data can also be written into the EEPROM using the "Vend Req" button. Before using the button, the program Vend_Ax.hex (usually in C:\Cypress\USB\Examples\EzUsb\Vend_Ax\) should be downloaded into the device. If the EEPROM uses 8-bit addressing, set the request field to 0xA2 and set it to 0xA9 if the EEPROM uses 16-bit addressing. The location to write into/read from is given in the value field. The data length can be provided in the length field and the data entered in the dropdown menu box next to "hex bytes". If the direction is set to OUT the data is sent on the I2C bus and written into the EEPROM. Similarly, the data in the EEPROM can be read back by setting the direction to IN.

The operations discussed above are typically enough to interface with the EZ-USB FX and act as a debugging tool. However when downloading your own descriptor tables with a different VID/PID, the control panel cannot be used after that since the control panel can no longer identify the device as a EZ-USB FX.

The GPIF tool:

The GPIF tool can be launched by clicking on the GPIF button in the EZ-USB Control Panel or by running the executable C:\Cypress\USB\Bin\gpif.exe. The GPIF tool allows users to generate GPIF waveforms and code for the EZ-USB FX and FX2. The target processor can be set by clicking on "Proj/Create New Project" and then clicking on the correct radio button. The file name and location can also be set in this box. In the GPIF tool, the user is presented with a C program that has some sections commented. In these commented sections there are some hot spots, which are in blue text. Clicking on these hotspots brings up pop-up dialogs or pull-down menus that allow us to change the values in those hot spots. Changing these values changes the code in the C code. This code can then be directly used in your program to interface with peripheral devices. Functions are available to do byte reads, byte writes, word reads and word writes using both GPIF pins and FIFOs too.

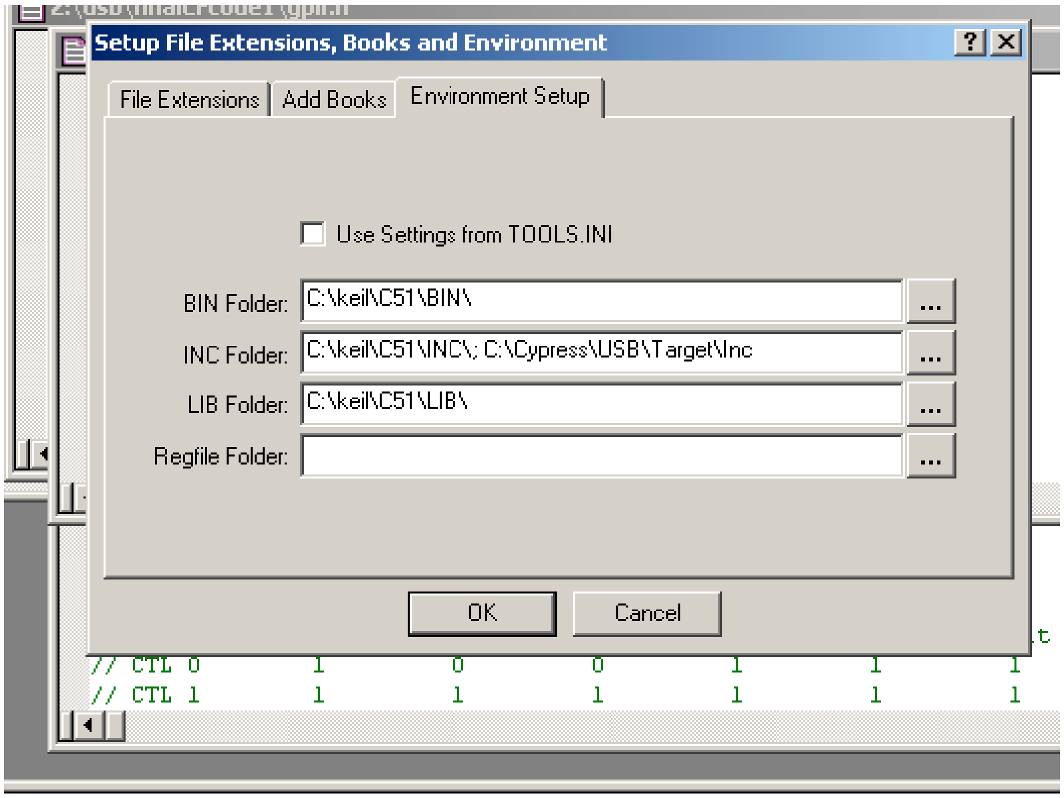
A main function is also there with sample read and write instructions. Thus this C file generated can be used by itself without the need for writing any more code.

The GPIF tool lets the user associate each of the read and write functions with one of four different waveforms. The waveforms have a maximum of 7 intervals. The duration of these intervals can be made fixed by setting the number of wait (clock) cycles or can be made to depend on one of the GPIF Rdy inputs. In each waveform the user can also set the intervals in which the data is available and the intervals in which the address is available. Next Data for the FIFO mode is also set here. Thus the GPIF tool provides an easy way to harness the power of GPIF pins on the FX and FX2 processors.

Using Keil tools(Tips specific to programming the EZ-USB FX):

Code for the EZ-USB FX can be written easily using Keil's uVision2. Once a new project is created, the user can select the target processor by first selecting "Target" in the Files window (so that the necessary menu options are available) and then clicking "Project/Select Device for Target". The dialog box that pops up allows the user to select the device, which in our case would be the FX chip by Cypress Semiconductor Corporation. The user can also instead right-click on "Target" and select

In order to use some of the header files, a few changes may have to be made to the Environment setup. This can be done by selecting "Project/File Extensions, Books and Environment". In the dialog box that pops up, the Environment setup Tab has to selected. The check box next to "Use settings from Tools.ini" has to be unchecked and the following details as shown in screenshot below have to be filled up. The directory paths and names may be different depending on installation.


Setup File Extensions, Books and Environment

File Extensions | Add Books | Environment Setup

[ ] Use Settings from TOOLS.INI

BIN Folder: C:\keil\C51\BIN\

INC Folder: C:\keil\C51\INC\; C:\Cypress\USB\Target\Inc

LIB Folder: C:\keil\C51\LIB\

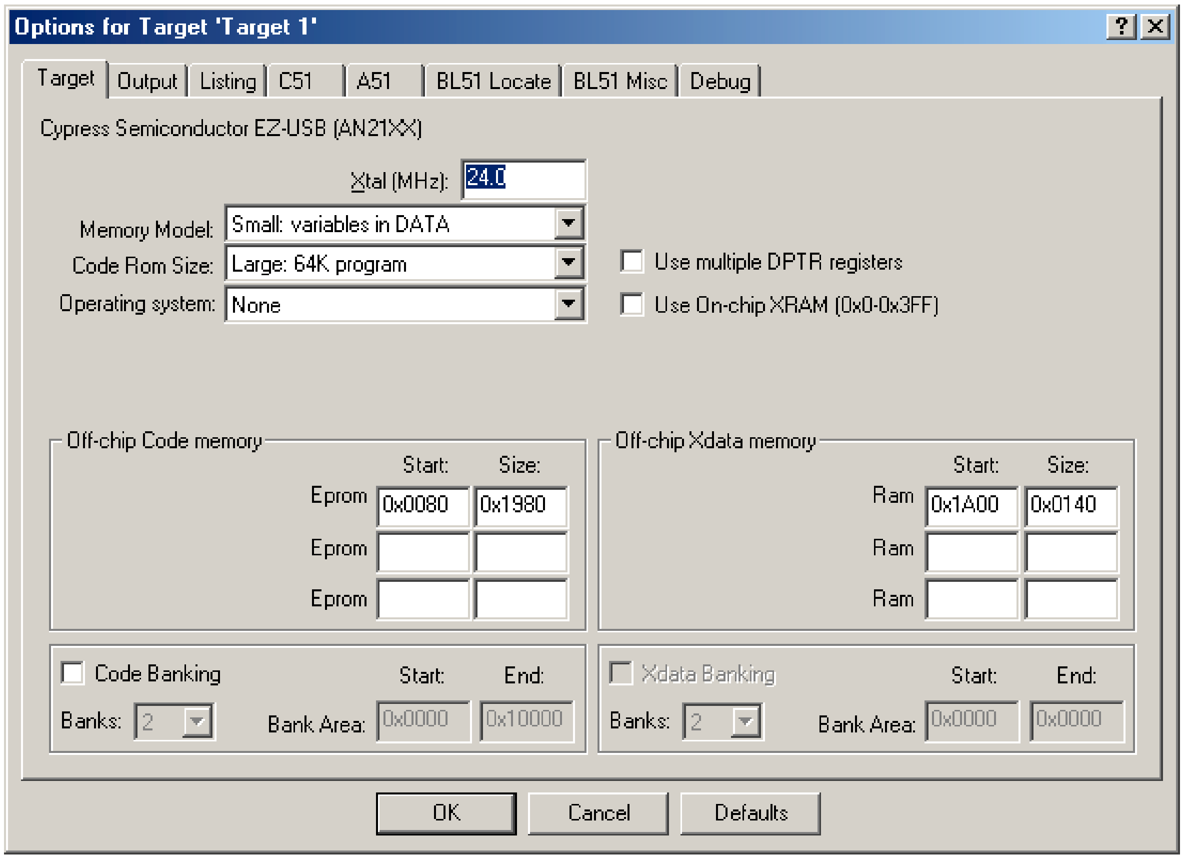
Regfile Folder:

```
// CTL 0    1    0    0    1    1    1
// CTL 1    1    1    1    1    1    1
```

Figure 5: Screen-shot showing the settings for Environment setup

There are several library functions available for use with the EZ-USB FX. These functions are available in the library file Ezusb.lib (typically available in the following directory, C:\Cypress\USB\Target\Lib\Ezusb\). To use the library, this file (Ezusb.lib) should be added to source group. The source code for this library is also available in the same directory as the library file.

The header files ezusb.h, ezregs.h and fx.h are useful header files containing names and definitions of almost all registers that are available. Definitions of some simple functions (such as suspend, resume, delay etc.) are also contained in these header files.


Options for Target 'Target 1'

Target | Output | Listing | C51 | A51 | BL51 Locate | BL51 Misc | Debug

Cypress Semiconductor EZ-USB (AN21XX)

Xtal (MHz): 24.0

Memory Model: Small: variables in DATA

Code Rom Size: Large: 64K program

Operating system: None

☐ Use multiple DPTR registers

☐ Use On-chip XRAM (0x0-0x3FF)

Off-chip Code memory

	Start:	Size:
Eprom	0x0080	0x1980
Eprom		
Eprom		

Off-chip Xdata memory

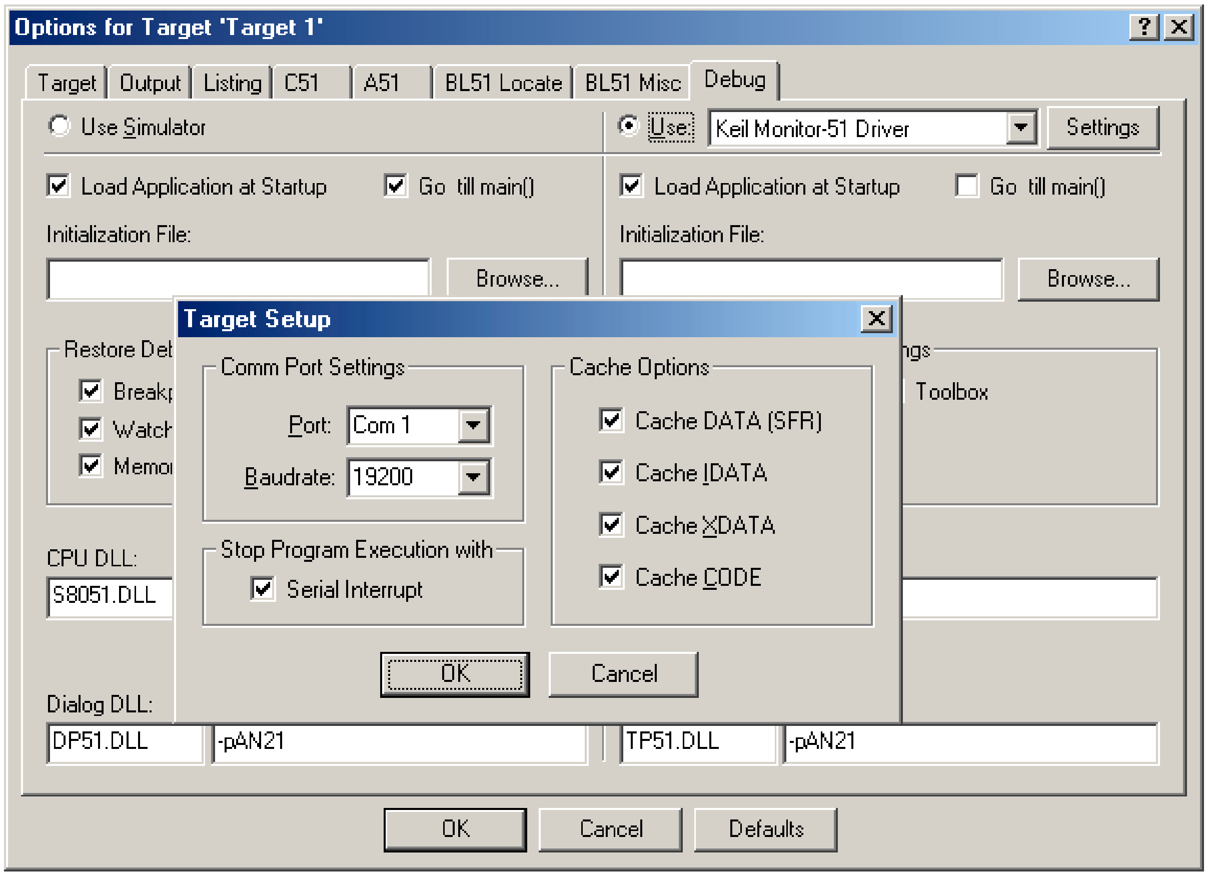
	Start:	Size:
Ram	0x1A00	0x0140
Ram		
Ram		

☐ Code Banking — Banks: 2 — Bank Area: Start: 0x0000 End: 0x10000

☐ Xdata Banking — Banks: 2 — Bank Area: Start: 0x0000 End: 0x0000

The memory map has to be set up correctly too so that there is space for the interrupt vectors (which can be used to add code for revectoring) and for separating the code and data space. To setup all this the user has to first right-click on "Target" and then select "Options for Target" or he/she can click on target and select "Project/Options for Target". In the dialog box that pops up, the target tab must be selected. Here the user can select the memory model (suggested model: small), the Code ROM size(choose based on estimated program size) and the operating system (none in our case). Also the code memory and data memory space can be specified as shown in the screen shot above. The total address space available for code and data on the EZ-USB FX spans from 0x0000 to 0x1B3F. (0x1B40 – 0x1FFF is used for the EZ-USB buffers and registers). Of this space 0x0080 should be reserved for interrupt vectors. The code starting address can thus be set to 0x0080 and the size can be adjusted as required. Once the code size has been set, the starting address for the data can be calculated (0x0080 + code size). The size of data should be set such that no data is written into space reserved for the EZ-USB buffers and registers. This size can be easily calculated since the starting address is known and the ending address (0x1B3E) is also known (data size =

0x1B3F - data starting address + 1). After this memory space allocation has been done this information has to carried over to the linker. For this BL51 tab has to be selected and here the checkbox next to “Use memory layout from Target Dialog” has to be checked. If the user wishes to choose between using the Keil debugger and the simulator, this choice can be made by clicking the “Debug” tab and selecting the appropriate radio button.



Use Simulator [✓ Use: Keil Monitor-51 Driver]

Simulator: Load Application at Startup [✓]; Go till main() [✓]; Initialization File:

Keil Monitor-51 Driver: Load Application at Startup [✓]; Go till main() [ ]; Initialization File:

Breakp [✓]; Watch [✓]; Memo [✓]

CPU DLL: S8051.DLL

Dialog DLL: DP51.DLL -pAN21 | TP51.DLL -pAN21

Comm Port Settings: Port: Com 1; Baudrate: 19200

Stop Program Execution with: Serial Interrupt [✓]

Cache Options: Cache DATA (SFR) [✓]; Cache IDATA [✓]; Cache XDATA [✓]; Cache CODE [✓]

If the Keil Mon-51 debugger is used, the serial port settings have to be set up correctly as shown in the screen shot above.

The optimization level and the kind of optimizing (for size or for speed) of the compiler can be changed by clicking on the “C51” tab. The highest level of optimization is 9.

After all the required options for the target have been set, the user can click “OK” and then save and compile the project.

Tip: In some cases it may be a good idea to compile more than once to make sure the linking has been done properly. Infact the Keil tool itself sometimes prompts the user to recompile. This is usually the case when a high level of optimization is used.

If the code compiles with no errors, a hex file should have been created. If no such file has been created, then the user must first select the “Output” tab in the “Options for Target” dialog box and then select the “Create executable” radio button and check the checkbox against “Create HEX file”. The HEX format should be HEX-80.

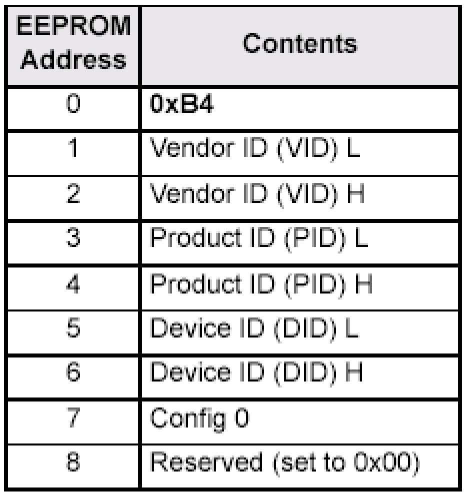
There is a lot more that can be done with Keil, but the information above should be enough to get one started at writing code for the EZ-USB FX.

Using a EEPROM for boot-loading

The EZ-USB FX is a soft RAM based solution. So the information that enables ith to come on as another device has to be stored somewhere. This information can be stored on a EEPROM on the I2C-compatible bus. When the EZ-USB FX comes out of reset, the boot loader first checks if there is an EEPROM on its I2C bus. If it detects a EEPROM, the loader reads the first EEPROM byte to determine how to enumerate.

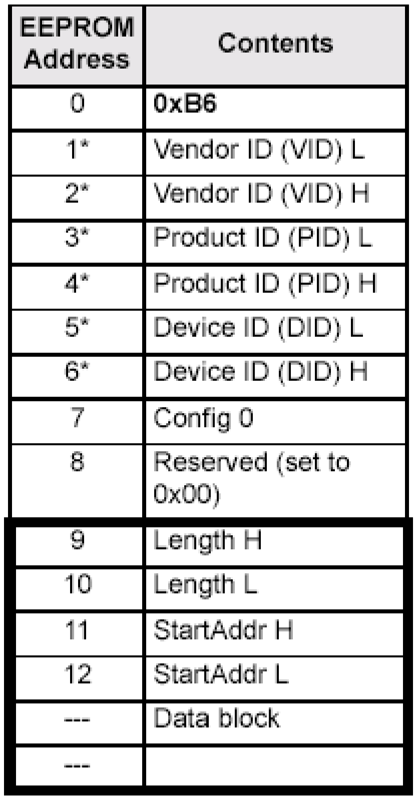
If the first byte is 0xB4, the EZ-USB FX copies the Vendor ID (VID), Product ID(PID) and Device ID (DID) from the EEPROM into its internal RAM. Only the VID/PID/DID bytes in the default USB device descriptor are replaced. The rest of the descriptor data is unchanged. So now, after initial enumeration, the driver download 8051 code and the USB descriptor data into the internal of the EZ-USB FX and starts the 8051. The code then ReNumerates and comes on as a new device based on the VID/PID combination.

Table 1: EEPROM data format for 0xB4 load



EEPROM Address	Contents
0	**0xB4**
1	Vendor ID (VID) L
2	Vendor ID (VID) H
3	Product ID (PID) L
4	Product ID (PID) H
5	Device ID (DID) L
6	Device ID (DID) H
7	Config 0
8	Reserved (set to 0x00)

Table 2: EEPROM Data format for 0xB6 load



EEPROM Address	Contents
0	**0xB6**
1*	Vendor ID (VID) L
2*	Vendor ID (VID) H
3*	Product ID (PID) L
4*	Product ID (PID) H
5*	Device ID (DID) L
6*	Device ID (DID) H
7	Config 0
8	Reserved (set to 0x00)
9	Length H
10	Length L
11	StartAddr H
12	StartAddr L
---	Data block

If the first byte is 0xB6, the EZ-USB FX load s all the EEPROM data into the internal RAM (not just the VID/PID/DID bytes). The RENUM bit is set to 1, which causes device requests to be fielded by the 8051 instead of the USB core. The 6 bytes after the first one in the EEPROM can contain the VIS/PID/DID bytes if it is desired at some point to run the 8051 code with RENUM bit set to 0. In this case the USB core will handle device requests and the VID/PID/DID is taken from the EEPROM. Following the 6-bytes of VID/PID/DID the data records follow from address 9 onwards. Each data record consists of a length (specified by Length H and Length L in Table 2), a starting address, and a block of data bytes. The maximum value of length H is 0x03 which limits the length of a record to 1023 bytes. The last data record must have the MSB of its length H byte set to 1 (length = 0x80 or greater). This last record loads the CPUCS register with a single byte (of which only the LSB is significant). Setting the LSB of this byte to 0 brings the 8051 out of reset.

If the first byte is neither 0xB4 or 0xB6, the boot loader just assumes that there is no boot EEPROM and enumerates as the EZ-USB FX with no serial EEPROM.

Hardware Issues:

The EZ-USB FX boot loader supports 2 I2C compatible EEPROM types, EEPROMs that use 8-bit addressing (24LC00 to 24LC16) and EEPROMs that use 16-bit addressing (24LC32 and greater). The data sheets of the EEPROM should be checked before connecting it to the board. EEPROMs that use 8-bit addressing should be wired such that A2, A1 and A0 are tied to ground (except for the 24LC00 which does not have address pins). If the EEPROM uses 16-bit addressing, the A2 and A1 address lines of the EEPROM should be tied low and A0 should be tied high. The EZ-USB FX boot loader identifies what kind of EEPROM is used from the way these lines are tied. So this has to be got right if the EEPROM is to be programmed correctly and made to act as a boot-up EEPROM

Software Issues:

A hex file containing the code cannot be directly downloaded into the EEPROM. It has to be converted into the correct format. This can be done with the "hex2bix" program (typically at C:\Cypress\USB\Bin\hex2bis.exe). This file can be copied to the directory where the code and the hex file are. The hex2bix program can be run from the DOS command prompt. To see the options available for hex2bix, at a command prompt, type hex2bix –h. The –F option sets the first byte of the EEPROM. The –V option lets you set the Vendor ID. The –C option lets you set the Config0 byte. Typically the command a user would use is the following:

```
prompt>Hex2bix –I –F 0xB6 –O <output file name> <input hex file name>
```

This sets the first byte to 0xB6 that make the program auto load from the EEPROM. The output file generated is in a format that can be downloaded to the EEPROM when the EEPROM button in the control panel is clicked.

Using the frameworks (fw) code from Cypress

Cypress provides some code that lets one get started with developing peripherals very quickly. It provides mainly three file fw.c, dscr.a51 and periph.c. There is also a Keil uVision2 project file (fw.uv2) that allows you to straight away open this project file and start writing the code for a USB device. The "fw" code for the EZ-USB FX is typically located (depends on the choice at the time of installation of course) at C:\Cypress\USB\Target\Fw\Ezusb.

Periph.c:

This code contains hooks for mainly two functions of significance, TD_Init() and TD_Poll(). These functions are called by fw.c when the USB device enumerates as a new device. Hardware initialization routines can be included in the TD_Init() function. Once the device has ReNumerated, the TD_Poll() function is called repeatedly. It is this function (or in functions called from TD_Poll()) that the user must write most of the code. Since this function is called repeatedly, as the function name suggests, this can be used to poll endpoint buffers and registers of the USB device and then perform the tasks required. There are also hooks for the functions TD_Suspend(), which is called when the device goes into suspend mode and TD_Resume(), which is called when the device resumes. Hooks are also provided for all the ISRs.

Dscr.a51:

This file which is in assembly contains the descriptor table that the frameworks code uses to ReNumerate. The device descriptors are all stored here. If the user needs to change the VID, PID, DID, the configuration descriptor , the interface descriptor or the string descriptor, it can be done easily in this file.

Fw.c:

This file contains the main() function and when the device ReNumerates it boots up using this code. The main function calls TD_Init() and TD_Poll() in periph.c. Fw.c has a function SetupCommand() to handle the setup data (device requests). It decodes the data requests and sends out the descriptor data which is in the dscr.a51 file. The code in fw.c almost never has to be touched. Most of the coding is to be done in periph.c. Only when making code for a HID (Human Interface Device) would you have to add some code for parsing the report descriptor.

Conclusion

Along with the EZ-USB FX chip, Cypress also provides useful tools (such as Keil's uVision2, the EZ-USB Control Panel, the GPIF tool) and software code to get you started (the frameworks code). All this put together makes a very good development system for a USB device. Some tools like the Control Panel have bugs in them (at least 2 different versions that we tried had 2 different bugs as discussed earlier), but overall the tools certainly make things much easier for the USB device developer and as Cypress claim “accelerate the learning curve”.

References