CY3674/CY3684

EZ-USB® Development Kit User Guide

Doc. # 001-66390 Rev. *F

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone (USA): 800.858.1810
Phone (Intnl): 408.943.2600
www.cypress.com

Copyrights

© Cypress Semiconductor Corporation, 2011-2014. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.

All trademarks or registered trademarks referenced herein are property of the respective corporations.

Contents

1. Introduction 7

1.1 Development Kit 7
1.2 Kit Contents 7
1.2.1 Hardware 7
1.2.2 Software 8
1.2.3 Other Tools 9
1.3 Documentation Conventions 10

2. Getting Started 11

2.1 Prepare the FX2LP Development Kit 11
2.2 Kit Software Installation 12
2.3 Install Hardware 12

3. My First USB 2.0 Transfer using FX2LP 13

3.1 Introduction 13
3.2 Application Summary 13
3.3 Procedure 13
3.4 Summary 16

4. EZ-USB Advanced Development Board (FX2LP DVK) 17

4.1 Introduction 17
4.2 Schematic Summary 17
4.3 Jumpers 18
4.4 EEPROM Select and Enable Switches SW1 and SW2 19
4.5 Interface Connectors 21
4.6 ATA Connector P8 25
4.7 U2 – 22v10 Gate Array Logic 25
4.8 Memory Maps 26
4.9 I2C Expanders 27
4.10 Indicators – Power and Breakpoint 28
4.11 General-Purpose Indicators 28

5. Development Kit Files 31

5.1 Bin 31
5.2 Documentation 31
5.3 Drivers 32
5.4 Firmware 32
5.4.1 Bulksrc 32
5.4.2 CyStreamer 32
5.4.3 Dev_io 32
5.4.4 Bulkloop Firmware 32
5.4.5 EP_Interrupts 32

5.4.6 extr_intr 32
5.4.7 hid_kb 32
5.4.8 Ibn 33
5.4.9 iMemtest 33
5.4.10 LEDCycle 33
5.4.11 Pingnak 33
5.4.12 Vend_ax 33
5.5 GPIF_Designer 33
5.6 Hardware 34
5.7 Target 34
5.8 Utilities 35
5.9 uV2_4k 36
5.10 Windows Applications 36
5.10.1 BulkLoop 36
5.10.2 Streamer 37
5.10.3 USB Control Center 38

6. FX2LP Code Development Using the EZ-USB Framework 39

6.1 Structure of an FX2LP Application 39
6.1.1 TD_Init 41
6.1.2 TD_Poll 41
6.1.3 Interrupt Service Routines 42
6.2 Build the Bulkloop Project 43
6.3 Framework 44
6.4 Framework Functions 45
6.4.1 Task Dispatcher Functions 45
6.4.2 Device Request Functions 45
6.4.3 ISR Functions 47
6.5 EZ-USB Library 48
6.5.1 Building the Library 49
6.5.2 Library Functions 49

7. Firmware Examples in Detail 51

7.1 hid_kb Firmware Example 51
7.1.1 Building Firmware Example Code for EZ-USB Internal RAM and External EEPROM. 53
7.1.2 Method to Download Firmware Image to EZ-USB Internal RAM 54
7.1.3 Method to Download Firmware Image to External I2C EEPROM 54
7.1.4 Binding Cypress USB Driver for the Downloaded Firmware Image 55
7.1.5 Testing the hid_kb Firmware Example Functionality 55
7.2 IBN Firmware Example 56
7.2.1 Description 56
7.2.2 Building Firmware Example Code for EZ-USB RAM and EEPROM 58
7.2.3 Method to Download Firmware Image to EZ-USB Internal RAM and External EEPROM 58
7.2.4 Binding Cypress USB Driver for the Downloaded Firmware Image 59
7.2.5 Testing the IBN Firmware Functionality 60
7.3 Pingnak Firmware Example 60
7.3.1 Description 60
7.3.2 Building Firmware Example Code for EZ-USB RAM and EEPROM 62
7.3.3 Method to Download Firmware Image to EZ-USB Internal RAM and External

EEPROM......62
7.3.4 Binding Cypress USB Driver for the Downloaded Firmware Image......63
7.3.5 Testing the pingnak Firmware Functionality......63
7.4 Bulkloop Example......63
7.4.1 Description......63
7.4.2 Building Bulkloop Firmware Example Code for EZ-USB RAM and EEPROM......65
7.4.3 Method to Download Bulkloop Firmware Image to Internal RAM or EEPROM......66
7.4.4 Binding Cypress USB Driver for the Downloaded Firmware Image......66
7.4.5 Testing the Bulkloop Firmware Functionality......66
7.5 Bulksrc Firwmare Example......68
7.5.1 Description......68
7.5.2 Building Bulksrc Firmware Example Code for EZ-USB RAM and EEPROM......70
7.5.3 Method to Download Firmware Image to EZ-USB Internal RAM and EEPROM......70
7.5.4 Binding Cypress USB Driver for the Downloaded Firmware Image......70
7.5.5 Testing Bulksrc Firmware Functionality......71
7.6 EP_Interrupts Example......72
7.6.1 Description......72
7.6.2 Building EP_Interrupts Firmware Example Code for EZ-USB RAM and EEPROM......72
7.6.3 Method to Program EP_Interrupts Firmware Image to EZ-USB Internal RAM and EEPROM......72
7.6.4 Binding Cypress USB Driver for the Downloaded Firmware Image......72
7.6.5 Testing EP_Interrupts Firmware Functionality......72
7.7 iMemtest Firmware Example......73
7.8 LEDcycle Firmware Example......73
7.9 Dev_IO Firmware Example......73
7.10 extr_intr Firmware Example......74
7.10.1 Testing the Example......75
7.11 Vend_ax Example......75
7.11.1 Testing the vend_ax Example......77

8. Resources **83**

8.1 Hardware Resources......83
8.2 Reference Designs......83
8.2.1 CY4611B - USB 2.0 to ATA Reference Design......83
8.2.2 CY3686 NX2LP-FLEX USB 2.0-to-NAND Reference Design Kit......84
8.3 Application Notes......84
8.4 Technical Reference Manual......86

A. Appendix **87**

A.1 U2 (GAL) code (file is 'FX2LP.ABL')......87
A.2 Board Layout......89
A.3 Schematic......90
A.4 Frequently Asked Questions......91

Revision History **93**

1. Introduction

1.1 Development Kit

The Cypress EZ-USB® FX2LP™ series (abbreviated as FX2LP) is a highly integrated controller family that serves as the basis for any USB high-speed peripheral device. To take full advantage of the USB 2.0 480-Mbps signaling rate, FX2LP contains specialized hardware to buffer USB data and to connect seamlessly to a variety of high-bandwidth external devices such as MCUs, ASICs, and FPGAs.

This guide describes the CY3684 EZ-USB FX2LP Development Kit (DVK), giving detailed instructions that enable you to do the following:

- Install the software tools
- Understand the DVK board and its features
- Test a simple firmware project to confirm tool installation
- Understand the structure of any FX2LP application, including the Cypress firmware framework
- Load and debug code
- Understand the Windows tools provided to help develop and debug applications
- Understand the example projects included in the kit

Note: This guide is also applicable to the CY3674 DVK that contains the Cypress FX1 chip. Instructions to use CY3684 and CY3674 are the same except for the fact that the later board works only at full-speed.

1.2 Kit Contents

The FX2LP DVK contains hardware and software. The software is available on the Cypress website. As Cypress constantly updates its tools, downloading the web-based package ensures you have the latest version.

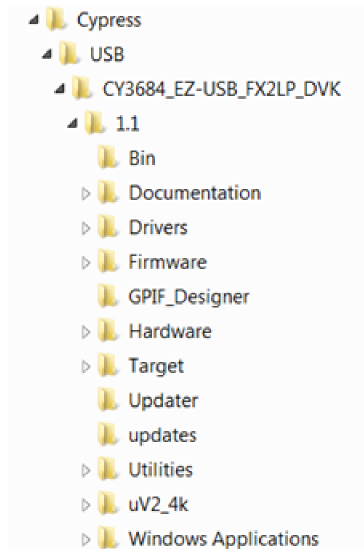
1.2.1 Hardware

- EZ-USB advanced development board
- EZ-USB prototyping board (breadboard)
- USB A-to-B cable
- RS-232 cable
- Quick start guide booklet

1.2.2 Software

Installing the download package creates the folder structure shown in Figure 1-1.

Figure 1-1. DVK Directories

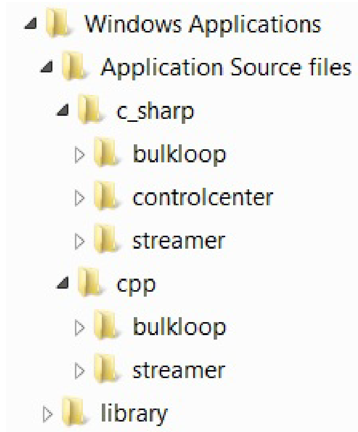


Following is a brief description of the installed folders. Refer to the Development Kit Files chapter on page 31 for a detailed description.

- **uV2_4k** contains an installer for a free version of the Keil integrated development environment (IDE) for writing and debugging FX2LP code using C and assembly language. This free version is fully featured but limited in code size to 4 KB. Cypress examples in the kit are under this limit, so you can modify and compile them as long as you do not exceed the 4-KB code limit.
- **Drivers** contains the Windows driver CyUSB3.sys for communicating with the FX2LP board, both as a firmware loader and as a tester. When you plug in the FX2LP board the first time, you are prompted to install a driver. This folder contains the driver version for various Windows operating system versions. The driver is the bridge between an FX2LP device and the PC host application.
- **Firmware** contains example firmware in the form of Keil projects.
- **GPIF_Designer** contains the installation file for GPIF Designer, a Cypress utility for configuring the FX2LP General Programmable Interface (GPIF). The GPIF provides a high-performance interface to external logic such as FPGAs and ASICs that operate at rates comparable to the USB 2.0 signaling rate of 480 Mbps. After you configure the interface, GPIF Designer generates a .c file for inclusion in your project.
- **Hardware** contains design information for the FX2LP DVK.
- **Target** contains FX2LP firmware files such as include files, the Cypress library, and the firmware framework on which FX2LP applications are built. It also includes Keil monitor code, which can be loaded into FX2LP to debug code using the Keil µVision 2 IDE and a PC serial port (UART).
- **Utilities** contains the program hex2bix.exe. After compiling code with the Keil IDE, you load the resulting .hex file into FX2LP RAM using the USB Control Center. You can also program the onboard EEPROM with your code so that it boots from EEPROM and runs at power-on. The hex2bix utility converts .hex files into the .iic file format required by the boot EEPROM. This conversion is done automatically in the example projects, producing .iic files for every code build operation.

- **Windows Applications** (Figure 1-2) contains files from Cypress SuperSpeed USB Suite, which is available for free download. These applications are used to test the example projects in this document. Both C# and C++ applications are provided. Subfolders contain a compiled executable and the full Microsoft Visual Studio project to allow study and modification. The full download provides libraries for C++ (CyAPI) and the .NET languages (CyUSB.dll), more examples, and documentation for using the libraries.

Figure 1-2. Windows PC Applications



  - **USB Control Center** is a utility for inspecting USB devices, loading FX2LP code, programming the DVK onboard EEPROM, and scheduling USB transfers to exercise the example FX2LP firmware. Every example described in this document uses this utility, so you may want to create a shortcut to it on your desktop. This application is available in C# only.
  - **BulkLoop** is the Windows application used to test the bulkloop firmware example. This application is provided in both C# and C++.
  - **Streamer** is the Windows application used to test the CyStreamer firmware example. This application is provided in both C# and C++.

1.2.3 Other Tools

- A USB-capable PC running Windows 2000, Windows XP, Windows Vista, or Windows 7/8.
- Microsoft Visual Studio. All Windows examples can be edited, compiled, and run using the free Visual Studio Express edition available on the Microsoft website.
- (Optional) The full retail version of the Keil µVision IDE. Purchasing this package removes the 4-KB restriction of the free edition supplied in the FX2LP DVK.
- (Optional) A USB bus analyzer records and displays USB traffic between the PC and the DVK board. This website includes a good list of analyzers.

1.3 Documentation Conventions

Table 1-1. Document Conventions for Guides

Convention	Usage
`Courier New`	Displays file locations, user-entered text, and source code: `C:\ ...cd\icc\`
Italics	Displays file names and reference documentation: Read about the *sourcefile.hex* file in the *PSoC Designer User Guide*.
[**Bracketed, Bold**]	Displays keyboard commands in procedures: [**Enter**] or [**Ctrl**] [**C**]
File > Open	Represents menu paths: File > Open > New Project
Bold	Displays commands, menu paths, and icon names in procedures: Click the **File** icon and then click **Open**.
Times New Roman	Displays an equation: $2 + 2 = 4$
Text in gray boxes	Describes Cautions or unique functionality of the product.

2. Getting Started

This chapter describes the installation of the CY3684 EZ-USB FX2LP DVK software. The process is similar for the CY3674 EZ-USB FX1 DVK.

2.1 Prepare the FX2LP Development Kit

1. Before plugging in the FX2LP kit for the first time, confirm the jumper positions as shown in Figure 2-1 and Table 2-1.

Figure 2-1. FX2LP Development Kit Jumpers

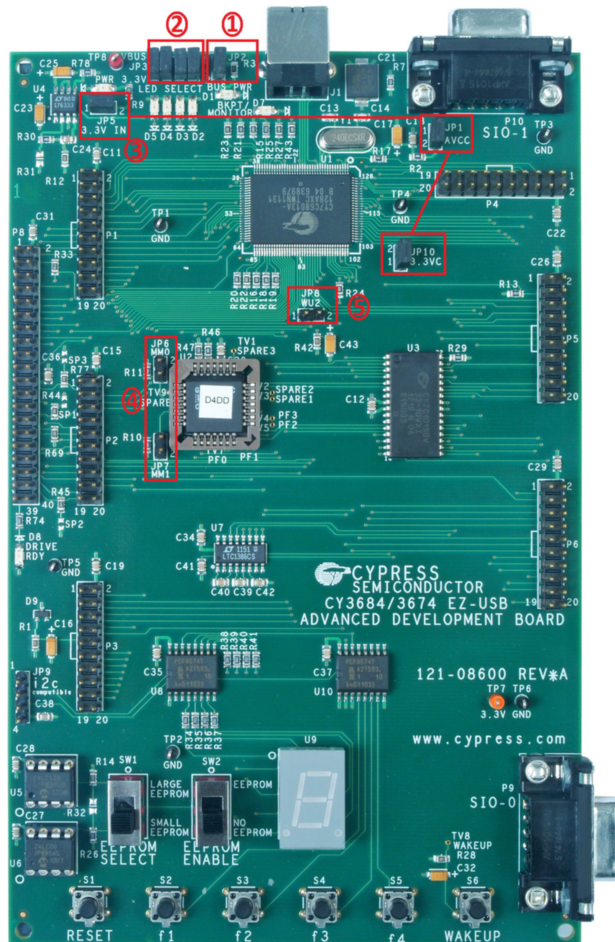


Table 2-1. Jumper Settings

Figure 2-1 Item	JP	State	Purpose
1	2	IN	Power board from USB connector
2	3	IN	All 4 jumpers IN – activate 4 LEDs, D2–D5
3	1, 5, 10	IN	Local 3.3-V power source
4	6, 7	OUT	Memory is configured for development
5	8	Either	Not used (Used for remote wakeup testing)

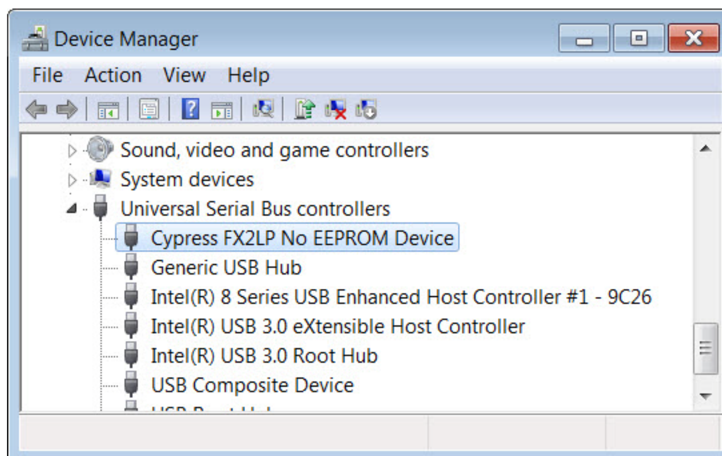
2. In the lower left corner of the board, move the **EEPROM ENABLE** slide switch to the **"NO EEPROM"** (down) position. This enables the FX2LP chip to enumerate as a bootloader device. The other slide switch **(EEPROM SELECT)** can be in either position.

2.2 Kit Software Installation

To install the kit software, follow these steps:

1. Download and install the latest version of the DVK software. Select the **"Typical"** installation type. If you use the recommended default paths, you should see the directory structure shown in Figure 1-1 on page 8. The installation automatically proceeds to install the Keil IDE tools and the GPIF Designer tool.
2. Plug the FX2LP board into a PC USB port. If this is the first time you have plugged it in, you should see pop-up messages to install a USB driver. Navigate to the driver folder and select the subfolder corresponding to your Windows OS version.
3. You can confirm successful driver installation by viewing the Windows **Device Manager** (Figure 2-2).

Figure 2-2. Confirming the Cypress Driver



Once the driver is installed, any Windows applications that communicate with the FX2LP DVK will recognize it.

2.3 Install Hardware

No hardware installation is required for this kit.

3. My First USB 2.0 Transfer using FX2LP

3.1 Introduction

This chapter introduces the FX2LP board and its tools by walking through a simple USB example called “bulkloop.” USB peripherals transfer data over logical entities called “endpoints.” A common endpoint type is called “bulk.” The PC schedules bulk transfers on a time-available basis. Bulk transfers use feedback (acknowledge packets) for flow control and employ error checking with a retry mechanism to guarantee the accurate delivery of USB data packets.

3.2 Application Summary

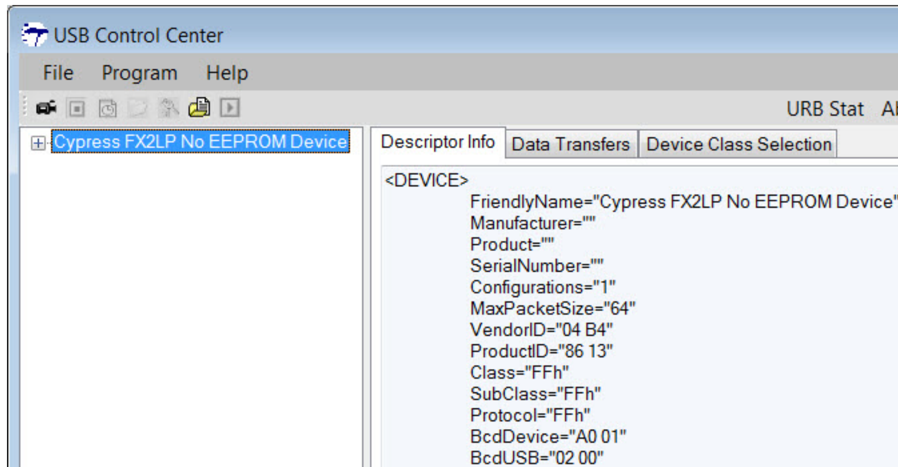
The FX2LP **bulkloop** firmware continuously checks for USB OUT packets sent by a PC. When they are received, the firmware copies the received data into an IN buffer for transmission back to the PC. On the PC side, you use the **USB Control Center** application to send data to FX2LP using OUT transfers and then to launch IN transfers to read back the data. You can confirm that the received data is the same data as that sent by the PC.

The **bulkloop** example utilizes an FX2LP feature called “double buffering”, which allows up to two packets to be stored for later transmission back to the PC.

3.3 Procedure

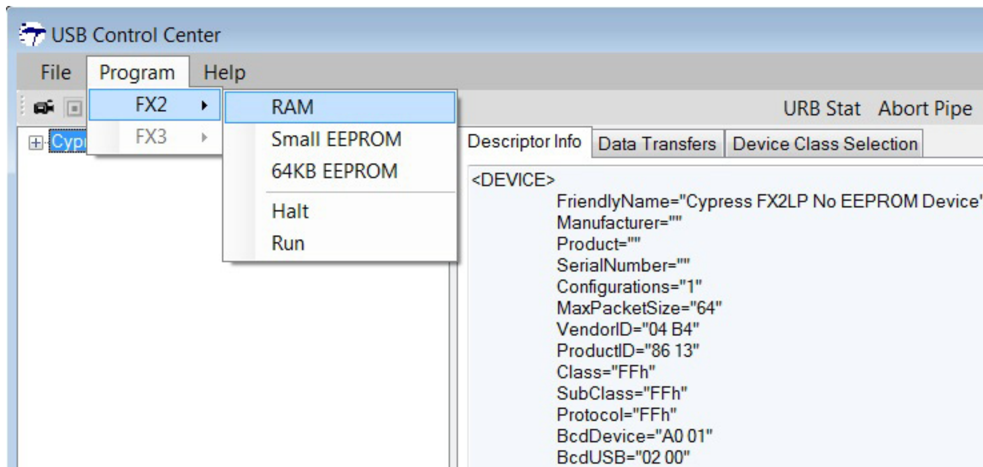
1. Locate the **USB Control Center** folder and navigate to its **bin** directory **(<Installed_directory>\<version>\Windows Applications\Application Source files\c_sharp\controlcenter\bin)**. Double click on the **CyControl.exe** file in either the **Debug** or **Release** folder.
2. If the FX2LP DVK is not plugged in, confirm its settings in Prepare the FX2LP Development Kit on page 11 and attach it to the PC using the USB receptacle and supplied cable. If the driver is correctly installed, you should see the FX2LP board appear as shown in Figure 3-1. If you do not, make sure the **EEPROM SELECT** switch is in the “down” position and reconnect. Deactivating the onboard EEPROM enables FX2LP to be used as a USB code-loading device.

Figure 3-1. FX2LP DVK Recognized by USB Control Center



3. Now you are ready to load the FX2LP firmware **bulkloop.hex** into the board. Click on the Cypress device entry to highlight it, and then choose **Program > FX2 > RAM** (Figure 3-2). In the Firmware folder, open the Bulkloop folder and double-click the **bulkloop.hex** file. After the code loads, the FX2LP DVK disconnects from USB and reconnects as the device created by the loaded firmware **bulkloop.hex**. This is Cypress ReNumeration™ in action – FX2LP has enumerated twice, first as a code loader and then as the bulkloop device created by the loaded firmware.

Figure 3-2. Downloading hex file into FX2LP's RAM



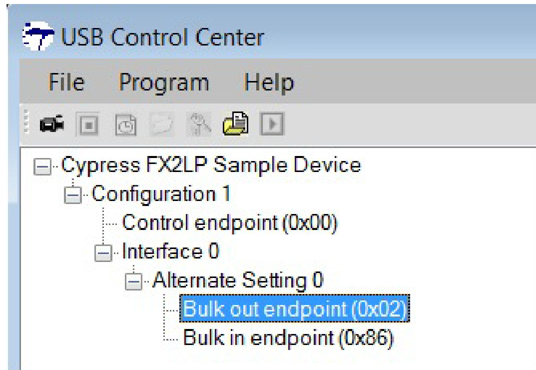
The LED D5 should blink eight times per second for a high-speed USB attachment and once per second for a full-speed attachment. For a high-speed USB attachment, you will see the LED blink slowly for a second or so and then start blinking quickly. This is because a USB device initially attaches at full speed and then negotiates the high-speed connection with the host. This is one of the many firmware details handled by the Cypress USB framework described in FX2LP Code Development Using the EZ-USB Framework chapter on page 39.

The 7-segment readout should light up with the number 0, indicating the number of FX2LP packets ready to be transferred back to the host. None have been received for loopback yet.

Note: Before you load a .hex file into the FX2LP DVK, you must press the Reset button (lower left corner) to reset the FX2LP and thereby re-enable the FX2LP bootloader.

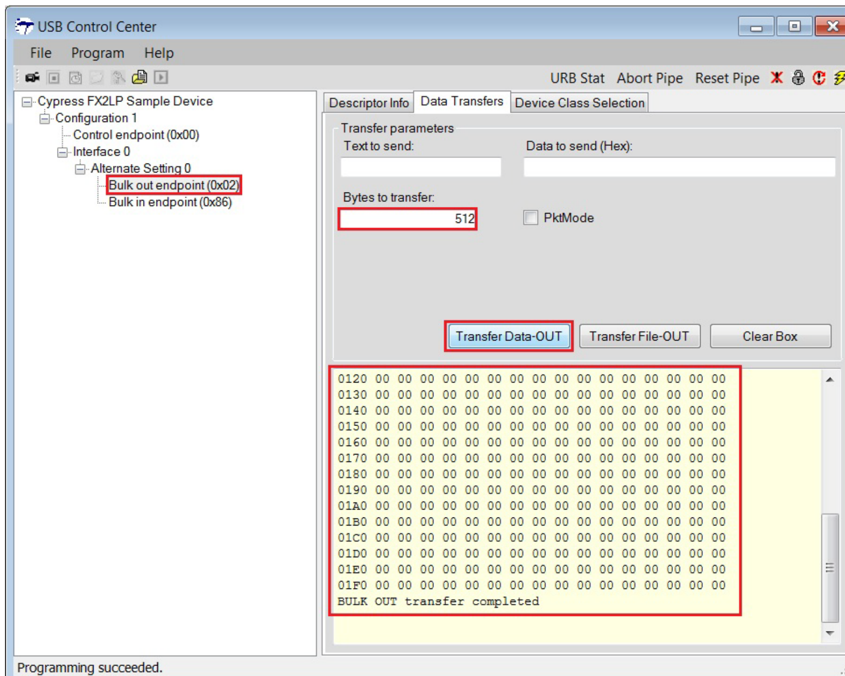
4. Expand the Cypress device entry to reveal the device's bulk endpoints EP2-OUT and EP6-IN (Figure 3-3).

Figure 3-3. Bulkloop Example Device Internals



5. Select the **Data Transfers** tab. Click on the **"Bulk out endpoint (0x02)"** entry in the left-hand panel, and notice that the **Transfer Data** button appears as **Transfer Data-OUT**. Click this button, and observe the following (Figure 3-4):
   a. A total of 512 bytes with zero default values transfer from the PC to the FX2LP board.
   b. LED D3 flickers to indicate the OUT transfer.
   c. The 7-segment readout increments to 1, indicating that one packet has been received over EP2-OUT and loaded into the FX2LP EP6 IN endpoint FIFO, ready for transfer to the PC.

Figure 3-4. Successful Bulk OUT Transfer

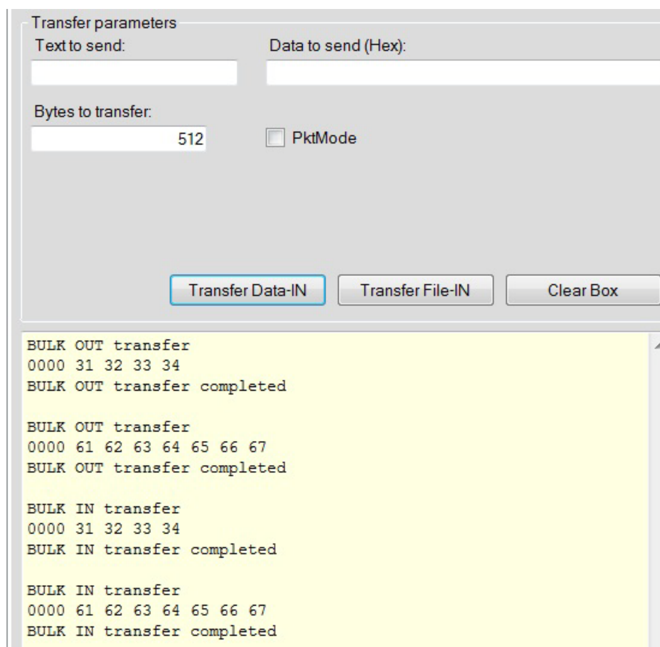


Click the **Transfer Data-OUT** button again. The PC dispatches a second packet to FX2LP, and the 7-segment readout increments to 2, indicating that two FX2LP packets are waiting to be sent to the PC.

6. Highlight the **"Bulk in endpoint (0x86)"** entry. The **Transfer Data** button now appears as **Transfer Data-IN**. Click this button and observe the following:
   a. A total of 512 bytes transfer from FX2LP to the host, which it displays as hexadecimal values.
   b. LED D2 flickers to indicate the IN transfer.
   c. The 7-segment readout decrements from 2 to 1, indicating that one FX2LP packet is ready for transfer to the PC.
7. Click the **Transfer Data-IN** button again. The second queued FX2LP packet transfers to the host, and the 7-segment readout indicates 0 packets waiting. This sequence confirms the double-buffered operation of the two endpoints.
8. Select the **"Bulk out endpoint (0x02)"** again. Then position the mouse cursor inside the **Text to send** box and type "1234". The hex values display in the **Data to send** box, and the **Bytes to Transfer** box increments for every digit typed. Click the **Transfer Data-OUT** button.
9. In the Text to send box, type "abcdefg", and then click the **Transfer Data-OUT** button.
10. Select the **"Bulk in endpoint (0x86)"**, and leave the default **Bytes to Transfer** value of 512 bytes. Click the **Transfer Data-IN** button twice.

As Figure 3-5 demonstrates, a USB peripheral always sends less than or equal to the number of bytes requested (512) and bytes available (4 or 7 in this example).

Figure 3-5. Two Packets OUT, two packets IN



3.4 Summary

This section validated correct installation of the Cypress tools and introduced the USB Control Center as a PC-based tester of FX2LP firmware. You used the USB Control Center to load a precompiled *.hex* file into the FX2LP board and then to test it by scheduling various bulk OUT and IN transfers.

Before exploring other firmware examples in the kit and learning how to write your own FX2LP applications, read the next section for detailed information about the FX2LP DVK.

4. EZ-USB Advanced Development Board (FX2LP DVK)

4.1 Introduction

The EZ-USB advanced development board (Figure 2-1 on page 11) provides a flexible evaluation and design vehicle for the EZ-USB family. The board provides expansion and interface signals on six 20-pin headers. A mating prototype board allows the construction and testing of custom USB designs. All ICs on the board operate at 3.3 V. The board is powered from the USB connector. It implements external expansion RAM to allow an 8051 debug monitor to be loaded without using internal FX2LP RAM, leaving the internal memory free for code development.

4.2 Schematic Summary

Read this description while referring to the FX2LP development board schematic and assembly drawing. Both drawings in PDF form are located in the DVK **Hardware** directory (Figure 1-1 on page 8).

U1 is an EZ-USB FX2LP, part number CY7C68013A-128AC. This full- function version brings out the 8051 address and data buses for external memory expansion. U2, a reprogrammable Gate Array Logic (GAL), provides RAM enable signals for four jumper-selectable memory maps. U3 is a 128-KB RAM, used for external 8051 memory expansion. Only 64 KB of this memory is addressed by the 8051; the A16 pin is connected to a pull-up resistor that optionally can be attached to a GAL output to provide bank switching options.

Note: In case of CY3674, U1 is an EZ-USB FX1 and the part number is CY7C64713-128AXC.

U4 is a 3.3-V, 500-milliamp voltage regulator. U5 and U6 are socketed EEPROMs, used for EZ-USB initialization and 8051 general-purpose access. U7 converts the 3.3-V 8051 serial port signals to bipolar RS-232 levels. U8 and U10 are Philips PCF8574 I/O expanders, which attach to the EZ-USB I2C bus and provide eight GPIO pins. U10 reads the four push-button switches, S2–S5, and U8 drives the 7-segment readout U9.

Six 20-pin headers, P1–P6, provide interface signals to the plug-in prototyping board supplied in this kit. They also serve as connection points for HP (Agilent) logic analyzer pods. P8 contains a subset of signals from P1–P6 on a connector that is pinned out for connection to a straight-through ATA cable.

Two slide switches, SW1 and SW2, control the connection and selection of the two socketed EEPROMs at U5 and U6.

4.3 Jumpers

The function of the jumpers on EZ-USB Development Board are explained in Table 4-1.

Table 4-1. EZ-USB Development Board Jumpers

Jumper	Function	Default	Notes
JP1, JP10	Connects 3.3 V power to the EZ-USB chip.	IN (1–2)	
JP2	Powers the on-board 3.3 V regulator from USB VBUS pin	IN (1–2)	To operate the board in self-powered mode, remove JP2 and supply 4 V to 5 V to JP2–1, and GND to a ground pin (TP1 is a convenient GND point).
JP3	Connects four GAL pins to LEDS D2, D3, D4, D5	IN (1–2) (3–4) (5–6) (7–8)	U2, the onboard GAL, contains code to use the four LEDs as software-programmable indicators. The LEDs can be removed from the GAL pins by removing the four jumpers.
JP5	3.3 V power	IN (1–2)	This jumper supplies 3.3-V power to the board. It can be removed and replaced with a series ammeter to measure board current.
JP6, JP7	Memory map selection	OUT (1–2)	These jumpers select one of the four memory maps for U3, the external 128 KB RAM. See Memory Maps on page 26 for details.
JP8	Wakeup2 pin	OUT (1–2)	Inserting a shorting plug into JP8 connects an onboard RC network (R42–C43) to the secondary remote wakeup pin WU2. This RC network can be used to test the periodic remote wakeup firmware when this dual-purpose pin (it defaults to PA3) is programmed as WU2.
JP9	I^2C bus test points	N/A	The I^2C bus SCL and SDA lines can be monitored or externally connected using JP9.

4.4 EEPROM Select and Enable Switches SW1 and SW2

SW1 selects between two socketed I^2C EEPROMs: U6, which is strapped to address 000, and U5, which is strapped to address 001. SW2 enables or disables whichever EEPROM is selected by SW1.

FX2LP has various startup modes that depend on what FX2LP detects on its I^2C bus at power-on. If nothing is connected, FX2LP enumerates as a USB bootloader as discussed in Getting Started chapter on page 11. If an EEPROM is detected, FX2LP uses EEPROM data as described in this section. After booting, the I2C controller is available for general-purpose use, as in the bulkloop example where the 7-segment readout was driven by an I^2C expander chip.

The FX2LP bootloader accommodates two EEPROM types in “small” and “large” versions, as listed in Table 4-2.

Table 4-2. Typical External EEPROMs

EEPROM Type	Size (bytes)	A2A1A0	Typical P/N
Small	16×8	000	24LC00
	128×8	000	24LC01
	256×8	000	24LC02
Large	8K×8	001	24LC64/5

Small EEPROMs are typically used to supply custom VID and PID information, allowing the internal FX2LP USB device to enumerate with a driver associated with your FX2LP design.

Large EEPROMs are typically used to bootload code into the internal EZ-USB RAM and then start up the 8051 to execute this code as the device defined by your code.

The FX2LP bootloader determines the EEPROM size by first initiating an I2C transfer to address 1010000 (1010 is the I2C address for EEPROMs, and 000 is the subaddress that is set by strapping the EEPROM A2A1A0 pins). If the device supplies an I2C acknowledge pulse, the EZ-USB loader recognizes the EEPROM as a “small” (256 bytes or smaller) device, which uses a single address byte. If this transfer does not return an ACK pulse, the FX2LP bootloader initiates a second I2C transfer, this time to address 10100001 (1010 = EEPROM, subaddress 001). If the I2C device returns an ACK, the FX2LP bootloader recognizes the EEPROM as a “large” device (larger than 256 bytes), which uses two address bytes.

If neither transfer returns an ACK pulse, the FX2LP bootloader acts as if no EEPROM is connected, that is, as if the FX2LP default internal device is capable of loading firmware into internal RAM over USB.

Three FX2LP startup sequences and their associated settings for SW1 and SW2 are as follows.

- Generic: SW2 = NO EEPROM, SW1 = either position

  When no EEPROM is connected to SCL and SDA, the FX2LP chip enumerates using its internal VID and PID values of VID = 0x04B4 and PID = 0x8613. This mode can be selected without removing any socketed EEPROMs by moving SW2 to the “NO EEPROM” (down) position. This electrically disconnects any EEPROMs in sockets U5 and U6. The NO EEPROM mode is used to start up FX2LP with VID/PID values compatible with the Cypress development tools.

  Once the code is running, SW2 can be switched to the ON position to allow 8051 access to the I^2C bus, for example, to reprogram an EEPROM.

- C0 Load: SW2 = EEPROM, SW1 = SMALL EEPROM

  A “C0” load provides FX2LP with external vendor ID (VID), product ID (PID), and device ID (DID) values, allowing the internal USB device to enumerate with custom ID values.

  At power-on, if the FX2LP chip detects an EEPROM with the hex value ‘C0’ as its first byte, it continues to load seven additional EEPROM bytes corresponding to the custom values plus a configuration byte. When FX2LP enumerates, it uses these EEPROM values instead of the hard-wired internal values. Because only eight bytes of data are required, a small EEPROM is generally used for this mode, for example, the 16-byte 24LC00.

- C2 Load: SW2 = EEPROM, SW1 = LARGE EEPROM

  A “C2” load provides a method to load the FX2LP internal RAM with 8051 firmware before enumeration. This boot load mechanism allows FX2LP to enumerate as a fully custom device as defined by the EEPROM code.

  At power-on, if the FX2LP chip detects an EEPROM with the hex value ‘C2’ as its first byte, it continues to load an FX2LP configuration byte, followed by blocks of 8051 code. The last byte loaded takes the 8051 out of reset. This mode usually requires a large EEPROM, such as the 8-KB 24LC64 or 16-KB 24LC128.

Note: The EZ-USB Technical Reference Manual provides complete details about firmware loading, EEPROM formats, and the default FX2LP USB device.

If an EEPROM is connected to the SCL and SDA lines but does not contain 0xC0 or 0xC2 as its first byte, the bootloader reverts to the “nothing connected” case, defaulting to the internal FX2LP USB device being capable of loading code over USB. Once the 8051 completes the boot phase and starts running code, the 8051 firmware has access to any connected EEPROM. This is because the 8051 I^2C controller is independent of the boot load logic.

4.5 Interface Connectors

Six 20-pin headers, P1–P6, on the FX2LP DVK have pins assigned to be compatible with HP (Agilent) logic analyzers, as shown in Table 4-3.

Table 4-3. Logic Analyzer Pinout

Agilent 01650-63203 Pod Pins			
No Connect	1	2	3.3 V from FX2LP board
CLK1	3	4	D15
D14	5	6	D13
D12	7	8	D11
D10	9	10	D9
D8	11	12	D7
D6	13	14	D5
D4	15	16	D3
D2	17	18	D1
D0	19	20	GND

These six headers serve three purposes:

1. They mate with the prototyping board supplied in the FX2LP DVK.
2. They allow direct connection of the HP (Agilent) logic analyzer pods (Agilent P/N 01650-63203).
3. They allow general-purpose probing by other logic analyzers or oscilloscopes.

Table 4-3 shows the logic analyzer pod pin designations. The FX2LP signals on P1–P6 are arranged to fulfill the following requirements:

- High-speed FX2LP strobe signals (PSEN, WR#, CLKOUT, IFCLK, and RD#) are connected to pin 3 of each of the five connectors for P1–P6. Therefore, they are used as the logic analyzer clock, CLK1.
- CLK2 is not used. Instead, each connector brings 3.3-V power from the FX2LP DVK up to the prototype board using pin 2.
- The signals are logically grouped. For example, the 8051 address bus is on P5, and the FX2LP FIFO data, which shares PORTB and PORTD pins, is on P1.

The 20-pin headers on the prototyping board can be stacked. Therefore, it is possible to build custom circuitry on the prototyping board, plug the board into the FX2LP DVK, and still plug logic analyzer pods to the six connectors alternate FX2LP pin names P1–P6. Table 4-4 to Table 4-9 show the FX2LP DVK pin designations for P1–P6. The alternate pin names are also shown.

Table 4-4. Pin Designation (P1)

Alternate	Default	P1		Default	Alternate
	NC	1	2	3.3 V	
	PSEN#	3	4	PD7	FD[15]
FD[14]	PD6	5	6	PD5	FD[13]
FD[12]	PD4	7	8	PD3	FD[11]
FD[10]	PD2	9	10	PD1	FD[9]
FD[8]	PD0	11	12	PB7	FD[7]
FD[6]	PB6	13	14	PB5	FD[5]
FD[4]	PB4	15	16	PB3	FD[3]
FD[2]	PB2	17	18	PB1	FD[1]
FD[0]	PB0	19	20	GND	

Table 4-5. Pin Designation (P2)

Alternate	Default	P2		Default	Alternate
	NC	1	2	3.3 V	
	NC	3	4	RDY1	SLWR
SLRD	RDY0	5	6	CTL5	
	CTL4	7	8	CTL3	
FLAGC	CTL2	9	10	CTL1	FLAGB
FLAGA	CTL0	11	12	PA7	FLAGD
PKTEND	PA6	13	14	PA5	FIFOADR1
FIFOADR0	PA4	15	16	PA3	WU2
SLOE	PA2	17	18	PA1	INT1#
INT0#	PA0	19	20	GND	

Table 4-6. Pin Designation (P3)

Alternate	Default	P3		Default	Alternate
	NC	1	2	3.3 V	
	WR#	3	4	RDY5	
	RDY4	5	6	RDY3	
	RDY2	7	8	BKPT	
	RESET#	9	10	N.C.	
	N.C.	11	12	PC7	GPIFADR7
GPIFADR6	PC6	13	14	PC5	GPIFADR5
GPIFADR4	PC4	15	16	PC3	GPIFADR3
GPIFADR2	PC2	17	18	PC1	GPIFADR1
GPIFADR0	PC0	19	20	GND	

Table 4-7. Pin Designation (P4)

Alternate	Default	P4		Default	Alternate
	NC	1	2	3.3 V	
	CLKOUT	3	4	GND	
	OE#	5	6	CS#	
	5 V	7	8	5 V	
	PLD2	9	10	PLD1	
	N.C.	11	12	D7	
	D6	13	14	D5	
	D4	15	16	D3	
	D2	17	18	D1	
	D0	19	20	GND	

Table 4-8. Pin Designation (P5)

Alternate	Default	P5		Default	Alternate
	NC	1	2	3.3 V	
	IFCLK	3	4	A15	
	A14	5	6	A13	
	A12	7	8	A11	
	A10	9	10	A9	
	A8	11	12	A7	
	A6	13	14	A5	
	A4	15	16	A3	
	A2	17	18	A1	
	A0	19	20	GND	

Table 4-9. Pin Designation (P6)

Alternate	Default	P6		Default	Alternate
	NC	1	2	3.3 V	
	RD#	3	4	INT5#	
	INT4	5	6	T2	
	T1	7	8	T0	
	WAKEUP#	9	10	SDA	
	SCL	11	12	PE7	GPIFADR8
T2EX	PE6	13	14	PE5	INT6
RxD1OUT	PE4	15	16	PE3	RxD0OUT
T2OUT	PE2	17	18	PE1	T1OUT
T0OUT	PE0	19	20	GND	

4.6 ATA Connector P8

Table 4-10 shows the pinout for P8, a 40-pin connector to interface to a standard ATA cable. This is for ATA use only. SP1, SP2, and SP3 should be bridged with solder to connect the appropriate pull-up and pull-down resistors required for ATA. An 80-pin cable is required for UDMA transfer modes and recommended for all transfer modes.

Table 4-10. P8 Pinout (ATA)

Alternate	Default	P8		Default	Alternate
RESET#	PA7	1	2	GND	GND
DD7	PB7	3	4	PD0	DD8
DD6	PB6	5	6	PD1	DD9
DD5	PB5	7	8	PD2	DD10
DD4	PB4	9	10	PD3	DD11
DD3	PB3	11	12	PD4	DD12
DD2	PB2	13	14	PD5	DD13
DD1	PB1	15	16	PD6	DD14
DD0	PB0	17	18	PD7	DD15
GND	GND	19	20	NC	KEYPIN
DMARQ	RDY1	21	22	GND	GND
DIOW#	CTL0	23	24	GND	GND
DIOR#	CTL1	25	26	GND	GND
IORDY	RDY0	27	28	GND	CSEL
DMACK#	CTL2	29	30	GND	GND
INTRQ	PA0	31	32	NC	RESERVED
DA1	PA2	33	34	NC	PDIAG#
DA0	PA1	35	36	PA3	DA2
CS0#	PA4	37	38	PA5	CS1#
DASP#	10K Pull-up	39	40	GND	GND

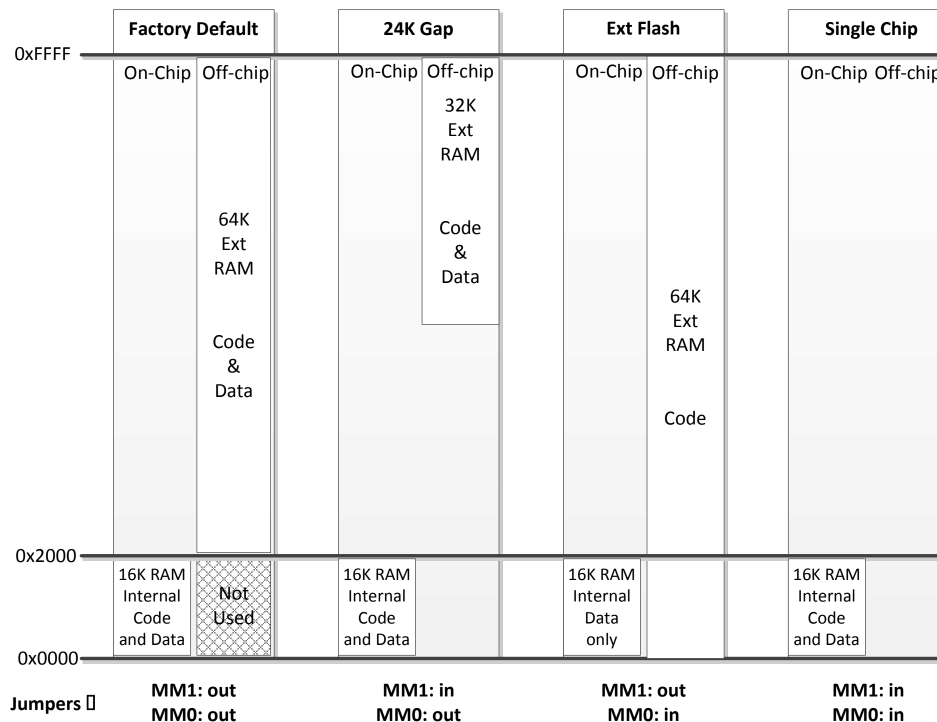
4.7 U2 – 22v10 Gate Array Logic

A 22v10 GAL provides a general-purpose “glue logic” on the board. It provides the logic to combine the PSEN and READ signals and adds memory map support, debug LEDs, and three spare outputs for customer-defined functions. The source code for the GAL logic is included in the **Hardware** directory (Figure 1-1 on page 8).

4.8 Memory Maps

EZ-USB is capable of accessing external RAM for keeping code and data. Figure 4-1 shows the different possible memory maps with the EZ-USB development board.

Figure 4-1. Four EZ-USB Development Board Memory Maps



Note The GAL sets EA=1 for the Ext Flash configuration only, enabling external code memory.

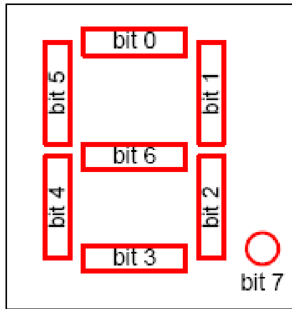
The factory default is to have both MM1 and MM0 jumpers removed. This setting should be used for all development work using the Keil software tools.

- The default configuration provides 16 KB of on-chip code and data memory, plus 48 KB of external RAM. The 8051 begins execution from internal RAM (the GAL sets EA=0). Although there is an apparent overlap between the internal 16 KB and the bottom 16 KB of the external RAM,
  EZ-USB disables RAM strobes for the bottom 16 KB, so there is no conflict. This EZ-USB decoding allows using a standard 64-KB RAM without requiring external decoding to inhibit access to the bottom 16 Kb.
- The second column, “24K gap”, enables the external RAM only for access to its upper 32 KB. This configuration is useful for systems that add external devices that require memory-mapped access. As with the default configuration, the 8051 begins execution from internal RAM (the GAL sets EA=0).
- The third column, “Ext Flash”, allows a flash memory (or other ROM) to be connected to the 8051 bus. This is the only configuration that starts 8051 execution from external memory (the GAL sets EA to ‘1’). Because external memory occupies the bottom 16K, the internal EZ-USB RAM is addressed only as data memory, instead of the combined program/data memory in the other three configurations.
- The fourth column, “Single Chip”, disables all external memory. This configuration is useful for testing the final code to ensure that it does not use external resources present in the development environment.

4.9 I²C Expanders

U8 and U10 are Philips PCF8574 I/O expanders. They connect to the I2C bus SCL and SDA pins and provide eight GPIO pins. U8 provides eight output bits that connect to the 7-segment readout U9. U10 provides eight input bits: four connect to push buttons S2–S5, and four are unconnected. U8 connects to the 7-segment readout U9, using the bit assignments in Figure 4-2. The bits are active-low: a zero turns on the segment.

Figure 4-2. 7-Segment Readout U9 Bit Assignments



The Cypress library EZUSB.LIB (in the Target/lib directory) provides C functions to access the I/O expanders. For example, the code to display the digit “1” (bits 1 and 2 LOW) is:

```
EZUSB_WriteI2C(0x21, 0x01, 0xF9);/* EZUSB_WriteI2C(I2C Device Address,
Number of Data bytes, Data)*/
EZUSB_WaitForEEPROMWrite(0x21);
```

The 7-segment I/O expander’s I²C address is 0x21. The second argument specifies one byte to be written, and the third byte is the data. The second function call is a general-purpose call that waits for the completion of an I2C write transfer.

To read the FX2LP DVK board buttons, declare an xdata variable and pass its address into the read function:

```
BYTE xdata buttons;
EZUSB_ReadI2C(0x20,0x01,&buttons);// Read button states
```

The I/O expander connected to the buttons has an I²C address of 0x20. The buttons occupy the bits shown in Table 4-11 in the returned byte.

Table 4-11. Button Bit Assignments

Bit	Switch	PCB Marking
0	S2	f1
1	S3	f2
2	S4	f3
3	S5	f4

4.10 Indicators – Power and Breakpoint

LED D1 is connected to the PCB 5-V power supply, which is normally supplied from the USB cable (VBUS pin). Alternatively, JP2 can be removed and an external 5-V power can be applied to the JP2 pin 1. In either case, D1 indicates the presence of the 5-V power.

LED D6 is connected to the 3.3-V voltage regulator output.

LED D7 is connected to the EZ-USB breakpoint (BKPT) pin. When using the Keil software development tools, this green LED indicates that the EZ-USB development board has enumerated and the Keil monitor has loaded and started running.

4.11 General-Purpose Indicators

A portion of the GAL (U2) decodes 8051 reads to certain external memory addresses to turn ON and OFF the four general-purpose indicators D2–D5. Figure 4-3 shows the positions of the four indicator LEDs, and Table 4-12 lists the external 8051 addresses that are read to turn them ON and OFF. The four jumpers above the LEDs must be installed to use this feature. These jumpers connect the LEDs to four GAL outputs.

Notes

- The CLKOUT signal is used as a clock to latch the LED output signals from the GAL. If CLKOUT is disabled, the LEDs will not update. (CLKOUT is enabled at power-on, and a program would need to clear bit 1 of the CPUCS register to disable it.)
- To use the LEDs for other purposes, such as wiring to other PC board signals for observation, first remove the shorting plug to disconnect the LED from the GAL. The LED terminal is the bottom pin of the connector, and the GAL I/O pin is the top pin.

Figure 4-3. Four Software-Controlled LEDs

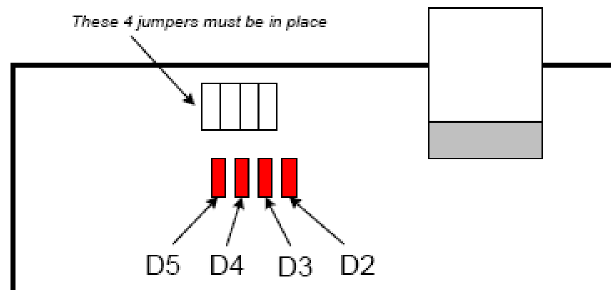


Table 4-12. Read These Addresses to Control LEDs

Indicator	Turn ON by Reading	Turn OFF by Reading
D2	0x88--	0x80--
D3	0x98--	0x90--
D4	0xA8--	0xA0--
D5	0xB8--	0xB0--

The low address byte is “don’t care.” You can add software-driven indicators using the following assembly code:

```
D5ON: mov MPAGE,#B8h ; turn D5 on
```

```
movx a,@r0 ; dummy read
;
D5OFF: mov MPAGE,#B0h ; turn D5 off
movx a,@r0 ; dummy read
```

This code example uses the 8051 8-bit indirect addressing mode. The MPAGE register (SFR 0x92) supplies the high address byte and r0 supplies the low address byte. Register r0 does not require initialization because the low address byte is “don’t care” for the LED decoding.

To turn the LEDs ON and OFF using the C code, declare the external memory locations, and then read their values into dummy variables:

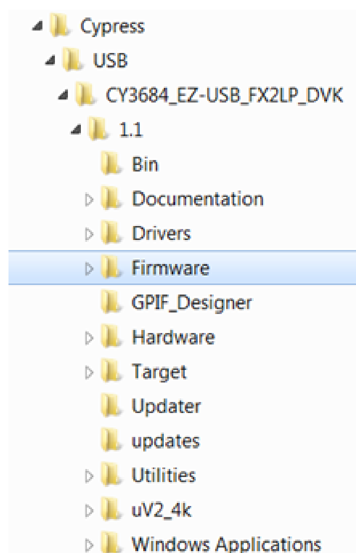
```
xdata volatile unsigned char D5ON _at_ 0xB800;
xdata volatile unsigned char D5OFF _at_ 0xB000;
unsigned char dum;
dum = D5ON; // turn D5 on
dum = D5OFF; // turn D5 off
```

This method for controlling LEDs uses no GPIO pins, making all of them available for user applications. Only data read operations to these addresses activate the LEDs; program execution at these addresses does not.

5. Development Kit Files

This section provides a description of the structure (Figure 5-1) and content of the FX2LP DVK after installation on a user's PC using the **"Typical"** default installation choice.

Figure 5-1. FX2LP DVK Directory Structure



5.1 Bin

This folder contains the following utilities

- **Hex2bix.exe:** This utility is used to convert a firmware image from the Keil compiler (*.hex*) to an EEPROM image (*.iic*).
- **Setenv.bat:** This is a batch file to set path variables for the Keil compiler and firmware examples. Run this batch file with the help of the Windows command prompt to set the environment variables necessary before compiling the firmware examples of the kit.

5.2 Documentation

This directory contains kit documentation, including:

- FX2LP datasheet and errata
- FX2LP Technical Reference Manual
- Application notes for FX1 to FX2 and FX2 to FX2LP migration
- Kit release notes
- Kit quick start guide

5.3 Drivers

This directory contains the Windows driver for operating the DVK. Versions are supplied for Windows XP, Windows Vista, Windows 7, and Windows 8, using 32-bit and 64-bit versions.

5.4 Firmware

This folder contains example projects to demonstrate various FX2LP features. Each folder contains a Keil project file (*xxx.uv2*). Double-click the project file to launch the Keil IDE and load the example project. To try out the precompiled code, load the .hex file into the FX2LP DVK using the USB Control Center as detailed in My First USB 2.0 Transfer using FX2LP chapter on page 13.

5.4.1 Bulksrc

FX2LP provides an endless source and sink of USB bulk data. This program demonstrates how to service bulk endpoints and manage endpoint buffers.

5.4.2 CyStreamer

This program demonstrates the maximum throughput that can be achieved over a USB 2.0 interface with the help of the Windows Streamer application. Both bulk and ISO transfers are supported.

5.4.3 Dev_io

This is an 8051 assembly language program to read the DVK board push buttons and activate the LEDs and 7-segment readout.

5.4.4 Bulkloop Firmware

This is the bulkloop project detailed in the My First USB 2.0 Transfer using FX2LP chapter on page 13 walkthrough. Bulk packets received on an OUT endpoint are looped back to an IN endpoint. FX2LP double buffering is demonstrated.

5.4.5 EP_Interrupts

This is bulkloop firmware that uses interrupts instead of polling status bits. It demonstrates the use of endpoint interrupts.

5.4.6 extr_intr

This project demonstrates the use of 8051 external interrupts INT0, INT1, INT4, INT5, and INT6.

5.4.7 hid_kb

This project implements a DVK board–based keyboard using the USB Human Interface Device (HID) class. The DVK buttons and 7-segment bars implement the keyboard functions shown in Table 5-1.

Table 5-1. DVK HID "Keyboard"

DVK	Keyboard Function
f1 button	Shift key
f2 button	'a' key
f3 button	'b' key
f4 button	'c' key
7-seg top	Screen Lock light
7-seg middle	Caps Lock light
7-seg bottom	Num Lock light

After you use the USB Control Center to load the ***hid_kb.hex*** file into the FX2LP board, the FX2LP board re-enumerates as a standard Windows keyboard. Any active window that accepts text will show the key presses implemented by this application.

5.4.8 Ibn

This project performs bulk loopback of EP2OUT to EP6IN and EP4OUT to EP8IN using the IN-BULK-NAK (IBN) interrupt to initiate the transfer.

5.4.9 iMemtest

This is a memory test firmware example that tests on-chip RAM and blinks **"g-o-o-d"** in the 7-segment readout to indicate success.

5.4.10 LEDCycle

This is a simple 8051 assembly language program to cycle the four DVK board LEDs.

5.4.11 Pingnak

This project performs bulk loopback of EP2OUT to EP6IN and EP4OUT to EP8IN using the PINGNAK interrupt to initiate the transfer.

5.4.12 Vend_ax

The USB specification provides a mechanism, called "vendor requests," to create your own custom USB commands. This project shows how to implement these requests. Application note AN45471 contains detailed information. Using this project with the USB Control Center, you can do things like sending packets from the PC to the FX2LP board to update the onboard LEDs and 7-segment readout.

5.5 GPIF_Designer

This folder contains the installer for GPIF Designer, a graphical tool for describing GPIF waveforms and states. Double-click the executable to install this utility. Application note AN66806 contains detailed information and walks through several design examples. Once you design the interface, GPIF Designer produces a *.c* file to include in your GPIF-based Keil project.

5.6 Hardware

Table 5-2 lists the files in this directory and their descriptions.

Table 5-2. Hardware Directory Files

Files	Description
CY3684_Board_Layout.brd/	FX2LP development board layout source file. This file can be opened using Allegro software.
CY3684_Board_Layout.pdf/	FX2LP board layout files in PDF form.
CY3684_Gerber.zip/	PCB images of different layers of the FX2LP development board PCB.
CY3684_PCBA_BOM.xls/	FX2LP development board bill of materials.
CY3684_Schematic.pdf/	FX2LP board schematic.
CY3684_Schematic.dsn/	Orcad schematic of FX2LP board.
PDC-9022-A-Dimension.PDF, PDC-9022-REVA.pdf, CY3681-2_ASSEMBLY.pdf and PDC-9022- A.zip	Mating prototype board specifications.
fx2lp.abl, fx2lp.jed	Logic equations in Data I/O ABEL format for the GAL22LV10C device on the FX2LP development board, and JEDEC programming file.

5.7 Target

This folder contains support files for the Keil IDE, as listed in Table 5-3.

This directory contains the EZ-USB register definition header files, Keil debug monitor, and so on. Following are the list of files.

Table 5-3. Target Folder Files

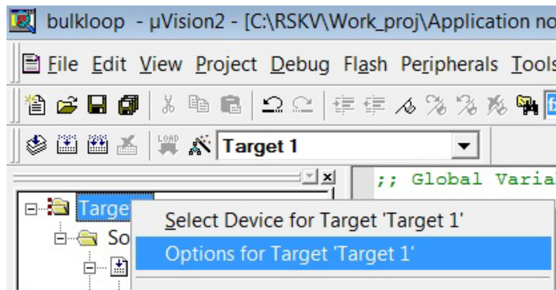
Sub-directory	File	Description
FW\Lp	Fw.c, periph.c, dscr.a51, fw.uv2	This directory contains the basic framework project source files used to develop the firmware examples in the EZ-USB development kits.
Monitor	mon-ext-sio0-c0.hex mon-ext-sio1-c0.hex mon-int-sio0-c0.hex mon-int-sio1-c0.hex	This directory contains Keil debug monitor *.hex* files. Versions are supplied that load into internal FX2LP RAM ("int") and external RAM ("ext") and utilize either the SIO-0 or SIO-1 DB9 connector on the development board. AN42499 has details to debug code using these files.
Inc	fx2regs.h, lpregs.h lpregs.inc, fx2regs.inc Fx2.h, lp.h syncdly.h, fx2sdly.h	These files contain EZ-USB register and basic structure definitions. Also, several delay routines of fixed duration (*syncdly.h/fx2sdly.h*)are defined to be used in framework code.
Lib/Lp	EZUSB.Lib, USBJmpTb.OBJ	This folder contains I^2C read/write routines library (*EZUSB.LIB*) and interrupt vector definitions for the EZ-USB device (*USBJmpTb.OBJ*)
File_Transfer	Various *.hex* files	This folder contains *.hex* data files that can be selected using the USB Control Center **"Transfer File"** feature. Files of various sizes and data patterns are provided.

5.8 Utilities

This folder contains the hex2bix utility that converts .hex files produced by the Keil IDE into .iic files suitable for programming onboard EEPROMs using the USB Control Center. The example firmware projects include a Keil command to automatically run the hex2bix utility at the end of every build operation. This command is accessed as follows:

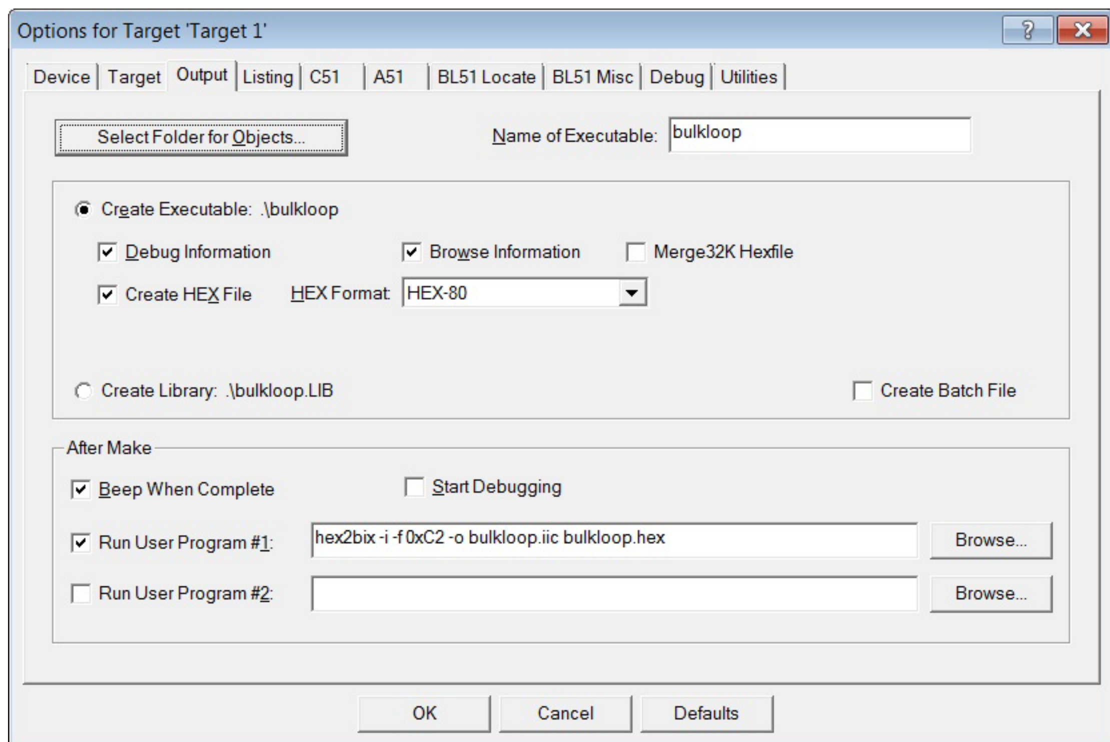
1. Right-click the top-level entry in the Keil **File** tab (usually labeled **“Target 1”**), and select **“Options for Target…”** as shown in Figure 5-2.

Figure 5-2. File Tab



2. Select the **Output** tab, as shown in Figure 5-3.

Figure 5-3. Output Tab



The **Run User Program #1** box is selected, and the text box shows the command line invocation of the hex2bix utility. If you create a new project and want hex2bix to run automatically after every build, copy this line and paste it into the same **Run User Program #1** box in your project. Also, make sure that hex2bix utility is present in the same folder of the new project. Then rename the *bulkloop.iic* and *bulkloop.hex* entries in the command line to reflect your file names, keeping the *.iic* and *.hex* extensions.

5.9 uV2_4k

The Keil IDE installer in this folder runs automatically when you install the FX2LP DVK software.

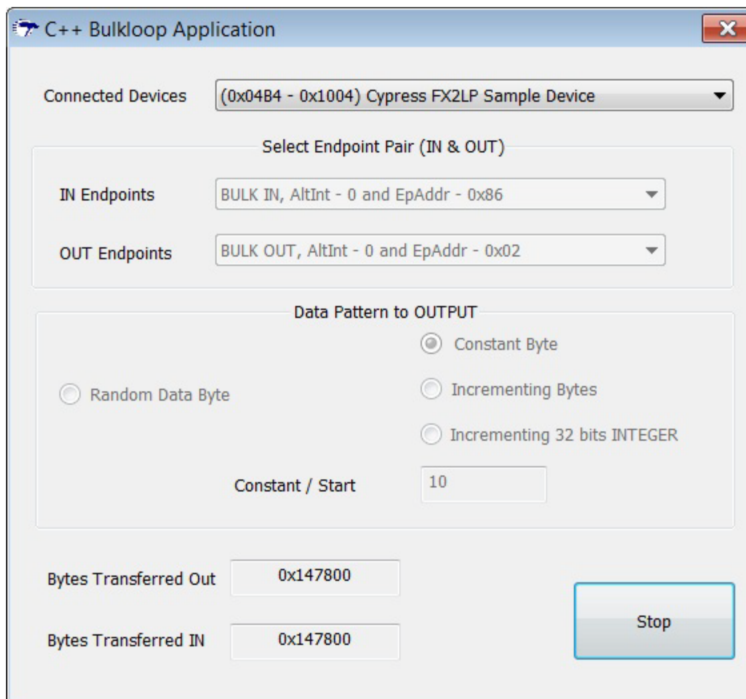
5.10 Windows Applications

Both C# and C++ versions of Windows PC applications are provided with the FX2LP DVK.

5.10.1 BulkLoop

My First USB 2.0 Transfer using FX2LP chapter on page 13 exercises the FX2LP bulkloop firmware using the USB Control Center. This Windows application is a second way to test the bulkloop firmware. It sends a constant stream of OUT packets to the FX2LP board running the bulkloop firmware. Figure 5-4 shows the screen shot of the C++ version of the BulkLoop application. A C# version of the BulkLoop application is also provided with the FX2LP kit.

Figure 5-4. C++ Version of Windows BulkLoop Application

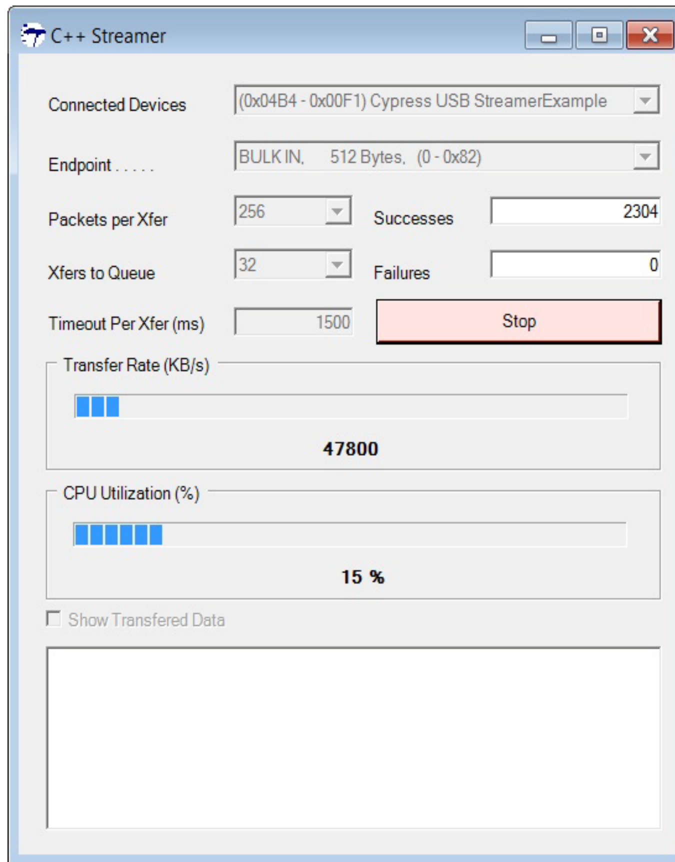


5.10.2 Streamer

This application is used to test the **CyStream** firmware. It continuously streams PC bulk or isochronous data in the OUT or IN directions. It recognizes the attached FX2LP device and provides a drop-down list of detected endpoints from which it either supplies a constant stream of OUT packets or issues a constant stream of IN requests to retrieve FX2LP packets.

On the Windows side, it allows testing with various packets per transfer and transfer queue size. Once the continuous transfers start, a throughput meter shows the transfer rate in kilobytes per second. This utility is very useful for testing your PC/OS/USB controller combination for achievable transfer rates. It is recommended to use the C++ version of the Streamer application to get higher data rates, as shown in Figure 5-5.

Figure 5-5. Windows Streamer Application (C++)



5.10.3 USB Control Center

The USB Control Center is used to download and test FX2LP code. In My First USB 2.0 Transfer using FX2LP chapter on page 13, it is used to test the bulkloop firmware. Every example firmware project can benefit from this Windows application.

The USB Control Center is part of a larger suite of tools, SuperSpeed USB Suite, which is available for free download. The suite includes extensive tools and documentation for writing Windows programs in the C++ or .NET languages that communicate with the FX2LP board via the Cypress driver. Programmer reference guides are included to explain how to make the Cypress library calls to the driver. The examples described in this document, BulkLoop and Streamer, are taken from this suite.

6. FX2LP Code Development Using the EZ-USB Framework

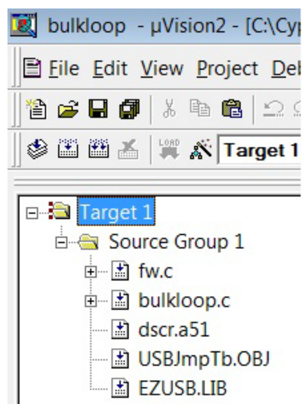
The USB example projects introduced in Firmware on page 32 use the Cypress EZ-USB framework code to implement USB functionality. This section explains the framework structure and shows how to develop your own code using it. Firmware Examples in Detail chapter on page 51 describes each example project in detail.

The firmware framework simplifies and accelerates USB peripheral development using the EZ-USB chip. The framework provides FX2LP register definitions and implements 8051 code for FX2LP chip initialization, USB standard device request handling, and USB power management (device suspend). The user provides a USB descriptor table and code to customize the peripheral function to complete a fully compliant USB device. The framework provides function hooks and example code to help with this process. The framework also provides *EZUSB.LIB*, a library to carry out common functions such as I^2C read and write. Most of the firmware examples in the FX2LP DVK are based on the framework.

6.1 Structure of an FX2LP Application

To explain the structure of a framework-based application, this section returns to the bulkloop firmware example to examine its source files. When you open the ***bulkloop.uv2*** project, you see the project files shown in Figure 6-1.

Figure 6-1. bulkloop.uv2 Project Files

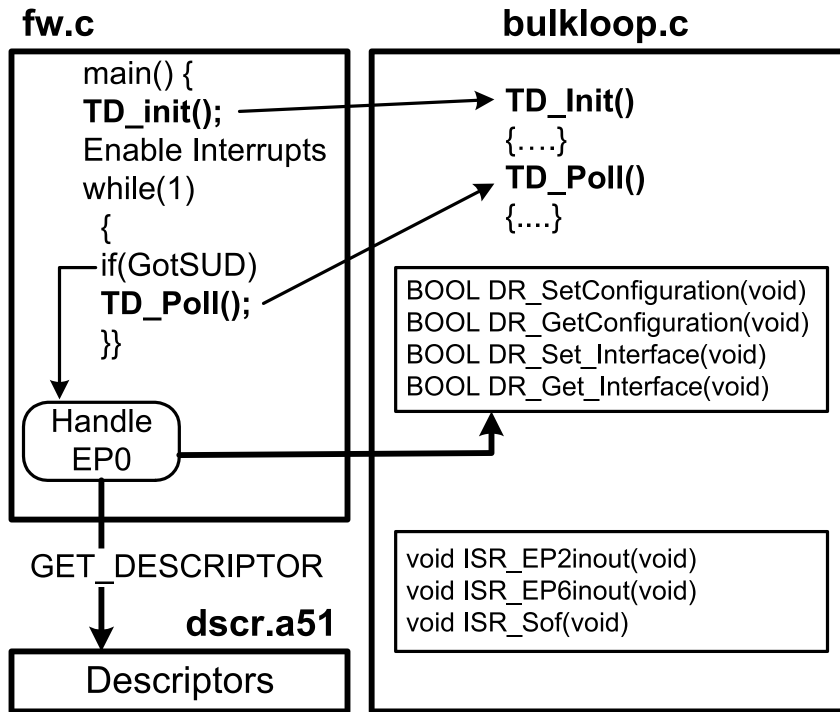


- **fw.c** is the file that contains Cypress USB firmware framework.
- **bulkloop.c** is the file that contains your application-level code. Other Cypress examples may call this code module **peripheral.c**; it is a good idea to rename it for your specific project name.
- **dscr.a51** is an assembly language file containing the USB descriptor data necessary to enumerate the USB device. The file comprises table data in the form of .DB (define byte) statements. You edit this file to customize items such as VID/PID (vendor ID, product ID) and descriptive strings for your design.

- **USBJmpTb.OBJ** is a jump table required by the FX2LP architecture that never requires modification.
- **EZUSB.LIB** contains a library of useful FX2LP functions, mostly dealing with I^2C communication. The source code is included with the FX2LP DVK in the **“lib”** subfolder.

Figure 6-2 shows how the code modules fit together.

Figure 6-2. Firmware Framework Flowchart



fw.c contains the main() function. It performs much of the USB maintenance such as enumeration, and it calls specifically named external functions in the application code module ***bulkloop.c*** whenever customization is required. ***fw.c*** does not require user modification. After performing various housekeeping steps, it calls an external function called “TD_init,” which you provide in ***bulkloop.c***. (The prefix “TD” stands for “task dispatcher.”) Then it enters an endless loop that checks for the arrival of SETUP packets over CONTROL endpoint 0. The loop also checks for the USB suspend event, but this is not used by the bulkloop app. Every time through the loop, it calls the external TD_Poll function, which you provide in ***bulkloop.c***. In this application, the TD_Poll function checks for arriving USB packets and loops them back to the PC.

Every USB peripheral receives two types of requests over its CONTROL endpoint: enumeration and operational.

- **Enumeration:** When a USB device is attached, the host PC sends multiple GET_DESCRIPTOR requests to discover the device type and its requirements as part of a process called “enumeration.” The **fw.c** code intercepts these requests and handles most of them automatically, using the values stored in the **dscr.a51** file. An advantage of using the USB framework is that the code has been tested and verified to pass USB requirements.

- **Operational:** Wherever user code is needed to complete a USB request, **fw.c** calls a specifically named external function with the “DR” (device request) prefix that you provide in the **bulkloop.c** file. For a simple application like bulkloop, there is only one configuration and one interface, so the two DR_Set, DR_Get function pairs in Figure 6-2 simply store the Set values sent by the host and echo them back when the host issues the corresponding Get requests. For more complex configurations, you can use these DR calls (“hooks”) to do things such as changing camera resolutions or routing requests to two different interfaces.

Because bulkloop.c contains a complete template for any USB device, you can use it as the basis for your custom implementation. The remainder of this section describes the three portions of this file that require user code to implement the bulkloop application.

6.1.1 TD_Init

This function does the following:

- Sets the 8051 clock to 48 MHz.
- Turns OFF the development board’s four LEDs. LEDs are turned ON and OFF by reading specific memory locations. This method controls the LEDs without consuming any I/O pins.
- Configures EP2 as a bulk OUT endpoint and EP6 as a bulk IN endpoint. Both are double buffered and use 512-byte FIFOs.
- Enables (“arms”) EP6-OUT to receive two packets. An FX2LP OUT endpoint is enabled by writing any value into its byte count register whose MSB is set. Setting the MSB (called the “SKIP bit”) instructs the FX2LP hardware to give the 8051 control of the packets instead of automatically committing them to the FIFO interface.
- Enables the FX2LP dual auto-pointers. These hardware pointers auto-increment for efficient memory-to-memory byte transfers from the EP6-OUT buffer to the EP2-IN buffer.
- Enables three interrupts: Start of Frame (SOF) and the EP2 and EP6 endpoint interrupts.

6.1.2 TD_Poll

TD_Poll is called in an infinite loop residing in **fw.c** (Figure 6-2). For the bulkloop application, only two tasks are required:

1. Update the 7-segment readout with the number of packets waiting for transmission to the host. The FX2LP register EP6CS (Endpoint 6 Control and Status) provides this number in bits 6-4.
2. Check endpoint FIFO flags to determine when it is time to transfer an OUT packet to an IN buffer. When it is time, move the packet data from the EP2-OUT buffer to the EP6-IN buffer.

To understand how the item 2 transfer decision is made, it is important to understand two points regarding FX2LP endpoint FIFO flags:

- When multiple buffering is used, the FULL and EMPTY flags reflect all the buffers, not just one. Therefore, in the double-buffered case for this example, if one OUT packet is received, the FULL flag remains unasserted because the second buffer is still available for an OUT transfer. Only when a second packet arrives does the FULL flag assert. Similarly, an IN endpoint EMPTY flag asserts only when both buffers are empty, ready for the 8051 to fill them with new data.
- FX2LP updates FIFO flags only after successful receipt or transmission of a USB packet. Therefore a looping copy operation occurs when both of the following two conditions are satisfied:
    - EP2-OUT is not empty.
    - EP6-IN is not full.

In other words, EP2-OUT has a packet, and EP6-IN has room for a packet. Doing the test in this way handles any packet size and takes the double buffering into account.

6.1.3 Interrupt Service Routines

The **bulkloop.c** file contains interrupt service routine (ISR) functions for every USB interrupt source. A few ISRs set flags for the ***fw.c*** code, and others perform USB enumeration functions. You need to fill in code only for the ISRs used by your application.

The following ISR shows how to clear an FX2LP USB interrupt request.

```
// Setup Token Interrupt Handler
void ISR_Sutok(void) interrupt 0
{
   EZUSB_IRQ_CLEAR();
   USBIRQ = bmSUTOK;  // Clear SUTOK IRQ
}
```

The “0” after the interrupt keyword is the ID for all USB interrupt requests. Two interrupt request flags are cleared in a particular order: first the general USB interrupt flag and then the individual USB source flag, in this example the “Setup Token Arrived” flag. This ISR is an example of a code “hook”; you can take any action when a SETUP packet arrives by inserting code into this ISR.

The bulkloop application requires four customized ISRs.

6.1.3.1 Set_Configuration ISR

The host sets the FX2LP configuration as the last step of its enumeration process. This is a good time to initialize the application hardware. The I^2C unit that drives the 7-segment readout is initialized here.

6.1.3.2 EP2INOUT/EP6INOUT ISR

These IRQs fire when a packet is dispatched from EP6-IN or arrives at EP2-OUT. The ISR code turns ON an FX2LP development board LED and then sets an “inblink” (EP6-IN) or “outblink” (EP2-OUT) variable to control how long the LED stays ON.

6.1.3.3 SOF ISR

The SOF ISR serves as a convenient timer, firing every millisecond at full speed and every 125 microseconds at high speed. The ISR code toggles an FX2LP development board LED every 500 times through the ISR, which equates to once per second at full speed and 8 times per second at high speed. The ISR code also decrements the “inblink” and “outblink” variables that were set when IN and OUT packets arrived, turning OFF indicator LEDs when the counters reach zero.

6.1.3.4 *Handling USB Dual Speeds*

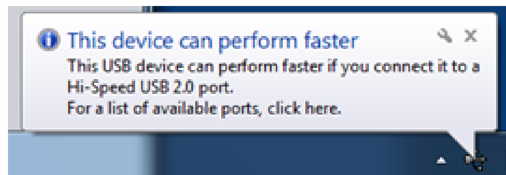
When a USB 2.0 device comes out of reset, it operates at full speed. The host then negotiates with the device using bus signaling to determine its operating speed. FX2LP provides an interrupt to indicate that it has just switched to high-speed operation as the result of the host-device speed negotiation.

A USB 2.0 device must operate at full and high speeds. A high-speed USB device provides two sets of descriptors, one for each speed. Two ISRs take care of designating the proper descriptors depending on speed:

- The ISR_Ures (USB reset) code designates the full-speed descriptor as the “current speed” descriptor, and the high-speed descriptor as the “other speed” descriptor. If plugged into a full-speed port, no further action is required.
- The ISR_Highspeed interrupt service code swaps the “current/other” descriptor designations – high speed is now the “current speed,” and full speed is the “other speed.”

The dual-speed descriptors allow Windows to display the message shown in Figure 6-3 if it detects a high-speed device plugged into a full-speed port.

Figure 6-3. Windows knows when a High-Speed Device can perform better



This nontrivial bit of USB housekeeping is an example of what the Cypress EZ-USB framework does for you. The dual descriptors and interrupt/descriptor swapping code is written for you. All you need to do is fill in the descriptor fields unique to your application.

6.2 Build the Bulkloop Project

In the FX2LP Bulkloop Firmware folder, double-click the ***bulkloop.uv2*** file. This opens the Keil µVision 2 IDE and loads the bulkloop project. To compile and link the project, click the **Rebuild All Target Files** button. This creates the ***bulkloop.hex*** file you downloaded in My First USB 2.0 Transfer using FX2LP chapter on page 13.

6.3 Framework

The Cypress firmware framework is written using the Keil µVision 2 compiler. It is tested only with these tools. Because the source uses several Keil C extensions, compatibility with other compilers is not guaranteed.

For your custom device firmware, you can either start with one of the firmware examples (recommended) or start with the bare framework code located in the Target directory. Create a new directory for your project and copy the various framework source files into it.

After starting the Windows command prompt, run setenv.bat (located in the bin directory) to set up the build environment. This batch file assumes that you have installed the DVK and Keil tools in the default directories. Table 6-1 lists the framework files.

Table 6-1. Files in Firmware Framework

File Name	Description
FW.C	This is the main framework source file. It contains main(), the task dispatcher, and the SETUP command handler. There is no need to modify this file for most firmware projects.
PERIPH.C	This source file is usually renamed to match your application name, for example, *bulkloop.c*. It contains initialization and task dispatch functions that are called from *fw.c*. This is where you customize the framework for your specific device. This file also contains empty ISR functions for all the USB (INT2) and GPIF (INT4) interrupts.
DSCR.A51	Assembly file that contains your device's custom descriptors.
FX2.H/LP.H	Header file containing common EZ-USB constants, macros, data types, and library function prototypes
FX2REGS.H/ LPREGS.H	EZ-USB register declarations and bit mask constants.
SYNCDLY.H/ FX2SDLY.H	Contains the synchronization delay macro. Certain FX2LP registers require sync delays before accessing them.
EZUSB.LIB	EZ-USB Library object code. See EZ-USB Library on page 48 for more details
USBJMPTB.OBJ	Object code that contains the ISR jump table for USB and GPIF interrupts. This file never requires modification.
BUILD.BAT	Batch file for compiling/linking the firmware using the Keil command line tools.
FW.UV2	Keil uVision2 project file for compiling/linking the bare firmware framework code.

6.4 Framework Functions

This section describes the framework functions. The framework provides function hooks (calls) to simplify the addition of user code. The functions are divided into three categories: task dispatcher (TD) calls, device request (DR) calls, and Interrupt Service Routines (ISR).

6.4.1 Task Dispatcher Functions

The following functions are called by the task dispatcher located in main().

6.4.1.1 TD_Init()

```
void TD_Init()
```

This function is called once during the initialization of the framework. It is called before ReNumeration and starting the task dispatcher. Device initialization code goes here.

6.4.1.2 TD_Poll()

```
void TD_Poll()
```

This function is called repeatedly during device operation. It contains code to implement the user's peripheral function. High-priority tasks can be performed in this function, but failure to return from this function in a timely manner prevents the framework from responding to device requests and USB suspend events. If a large amount of processing time is required, it should be split up to execute in multiple TD_Poll() calls.

6.4.1.3 TD_Suspend()

```
BOOL TD_Suspend()
```

This function is called before the framework enters suspend mode. Write code in this function to perform low-power housekeeping such as turning off peripheral devices. Returning TRUE instructs the framework to put FX2LP into USB suspend. Alternatively, the user code can prevent the framework from entering USB suspend by returning FALSE.

6.4.1.4 TD_Resume()

```
void TD_Resume()
```

This function is called after the framework exits the suspend state because the USB bus has signaled resume, or FX2LP has received a remote wakeup signal on its WAKEUP pin. At this point, FX2LP resumes full-power operation.

6.4.2 Device Request Functions

The SetupCommand() function in *fw.c* calls the device request functions. The calls are used to override or augment the actions of the default device request handler.

6.4.2.1 DR_GetDescriptor()

```
BOOL DR_GetDescriptor()
```

This function is called before the framework decodes and acts on the GetDescriptor device request. The register array SETUPDAT contains the eight request bytes from the current SETUP request. The SETUPDAT data can be parsed by user code to determine which Get_Descriptor command was issued. If TRUE is returned, the framework parses and implements the request. If FALSE is returned, the framework does nothing.

This hook is provided to allow user code to handle USB Get_Descriptor requests that supplement the standard USB requests. For example, the USB Chapter 9 specification does not deal with USB class requests such as HID and Vendor. To implement a Get_Descriptor requests unique to a class, insert code in this function to check the eight SETUPDAT registers for the descriptor you are looking for (for example, Get_Report for HID) and supply the custom descriptor, if necessary. Then return FALSE because your code has handled the Get_Descriptor request; the framework does not need to. If you do not find the Get_Descriptor unique to your class, return TRUE so the framework will handle the request.

Figure 6-4 is a simplified excerpt from the **hid_kb.c** firmware example. The framework has already done most of the work – received the SETUP packet, decoded the request to be Get_Descriptor, and called your DR_GetDescriptor() function. The third SETUPDAT byte contains the descriptor type, which the code checks for HID or REPORT. If it finds either of these, it sets up the transfer and returns FALSE so the framework does not further parse the request. If your code finds neither HID descriptor type, it returns TRUE so the framework can process the standard request as usual.

Figure 6-4. How HID Uses Get_Descriptor()

```
BOOL DR_GetDescriptor(void)
{
    BYTE length,i;

    pHIDDscr = (WORD)&HIDDscr;
    pReportDscr = (WORD)&HIDReportDscr;
    pReportDscrEnd = (WORD)&HIDReportDscrEnd;

    switch (SETUPDAT[3])
    {
        case GD_HID:                    //HID Descriptor
            SUDPTRH = MSB(pHIDDscr);
            SUDPTRL = LSB(pHIDDscr);
            return (FALSE);
        case GD_REPORT:                 //Report Descriptor
            length = pReportDscrEnd - pReportDscr;

         AUTOPTR1H = MSB(pReportDscr);
         AUTOPTR1L = LSB(pReportDscr);

         for(i=0;i<length;i++)
            EP0BUF[i]=XAUTODAT1;

         EP0BCL = length;
         return (FALSE);
        default:
            return(TRUE);
    }
}
```

As USB evolves and expands, this hook allows your code to respond to any new Get_Descriptor requirements. This is the main reason for structuring the framework code to include these customization hooks.

6.4.2.2 DR_SetConfiguration()

```
BOOL DR_SetConfiguration()
```

The framework calls this function as its only response to the Set_Configuration request. Your code should store the requested configuration number it finds in SETUPDAT[2] and activate the requested interface if applicable (most apps use only one configuration).

6.4.2.3 *DR_GetConfiguration()*

```
BOOL DR_GetConfiguration()
```

The framework calls this function as its only response to the Get_Configuration request. Your code should return the value it received from the Set_Configuration request.

6.4.2.4 *DR_SetInterface()*

```
BOOL DR_SetInterface()
```

The framework calls this function as its only response to the Set_Interface request. Your code should store the alternate setting it finds in SETUPDAT[2] and activate the requested setting, if applicable.

6.4.2.5 *DR_GetInterface()*

```
BOOL DR_GetInterface()
```

The framework calls this function as its only response to the Get_Interface request. Your code should return the alternate setting value it received in the Set_Interface request.

6.4.2.6 *DR_GetStatus()*

```
BOOL DR_GetStatus()
```

This function is called before the framework implements the Get_Status standard request. The framework handles standard status requests, so this function normally only needs to return TRUE.

6.4.2.7 *DR_ClearFeature()*

```
BOOL DR_ClearFeature()
```

This function is called before the framework implements the Clear_Feature device request. The framework handles standard Clear_Feature requests, so this function normally only needs to return TRUE.

6.4.2.8 *DR_SetFeature()*

```
BOOL DR_SetFeature()
```

This function is called before the framework implements the Set_Feature device request. The framework handles standard Set_Feature requests, so this function normally only needs to return TRUE.

6.4.2.9 *DR_VendorCmnd()*

```
void DR_VendorCmnd()
```

This function is called when the framework determines that a vendor-specific USB request was received. This function has no return value. The framework does not implement any vendor-specific commands. However, the FX2LP internal logic uses vendor-specific command 0xA0 to implement software uploads and downloads. Therefore, vendor command 0xA0 is never passed to the user's code.

6.4.3 ISR Functions

FX2LP supports over 40 USB and GPIF auto-vectored interrupts. (“Auto-vector” refers to FX2LP hardware that automatically directs interrupt requests to individual memory locations.) ***PERIPH.C*** contains stub ISR functions for all these interrupts. This section documents the ISRs that require special handling by device firmware. For more information, refer to the “Interrupts” section in the EZ-USB Technical Reference Manual.

FX2LP interrupts have enable bits that must be set to activate the following ISRs.

6.4.3.1 *ISR_Sudav()*

```
void ISR_Sudav(void) interrupt 0
```

This function is called on receiving the Setup Data Available interrupt. This function needs to set the global flag GotSUD to TRUE so the device request handler loop can detect the SETUP packet arrival.

6.4.3.2 *ISR_Sof()*

```
void ISR_Sof(void) interrupt 0
```

This function is called on receiving the Start of Frame interrupt. The only mandatory action for this interrupt is to clear the interrupt request. As seen in the bulkloop example, this is an excellent place to insert timer code since the SOF interrupt fires once per millisecond at full speed, and eight times per millisecond at high speed.

6.4.3.3 *ISR_Ures()*

```
void ISR_Ures(void) interrupt 0
```

This function is called on receiving the USB (bus) Reset interrupt. In your custom code, place any housekeeping that must be done in response to a USB bus reset. Since FX2LP enumerates at full speed after a reset, the reset code should set the configuration descriptor pointers to the full-speed version, as explained in Interrupt Service Routines on page 42. This code is included in all USB firmware examples in this kit.

6.4.3.4 *ISR_Susp()*

```
void ISR_Susp(void) interrupt 0
```

This function is called on receiving the USB Suspend interrupt. The default framework code sets the global variable Sleep to TRUE in this routine. This is required for the Task Dispatcher to detect and handle the suspend event.

6.4.3.5 *ISR_Highspeed()*

```
void ISR_Highspeed(void) interrupt 0
```

This function is called on receiving the USB HISPEED interrupt. In your custom code, place any housekeeping that must be done in response to a transition to high-speed mode in this routine.

As explained in Interrupt Service Routines on page 42, code in this ISR must set the configuration descriptor pointers to the high-speed version. This code is included in all USB firmware examples in this kit.

6.5 EZ-USB Library

The EZ-USB library is an 8051 .LIB file that implements functions that are common to many firmware projects. These functions need not be modified and therefore are provided in library form. However, the kit includes the source code for the library if you need to modify a function or only want to know how something is done.

In addition to providing common functions, the library creates register definitions for all EZ-USB registers. The source code and the compiled library files are located in the ***Target\Lib\lp*** folder.

6.5.1 Building the Library

Only the full retail version of the Keil tools can build library files; the evaluation version supplied with the kit does not build them. After starting the Windows command prompt, run ***setenv.bat*** (located in the bin directory) to set up the build environment. This batch file assumes that you have installed the DVK and Keil tools in the default directories. To build the library, run the **build.bat** file from the command prompt.

Build.bat also assembles the ***usbjmptb.a51*** file to create ***usbjmptb.obj***. This file contains the jump table for the USB (INT2) and GPIF (INT4) auto-vectored interrupts. See the EZ-USB Technical Reference Manual in the kit documentation for more information on auto-vector interrupts.

6.5.2 Library Functions

6.5.2.1 EZUSB_Delay()

```
void EZUSB_Delay(WORD ms)
```

This function performs a looped wait for a given number of milliseconds. The parameter ms determines the length of the busy wait. Upon completion of the delay, the function returns. This function adjusts the delay based on the CPU clock.

6.5.2.2 EZUSB_Discon()

```
void EZUSB_Discon(BOOL renum)
```

This function performs a USB disconnect/reconnect. It disconnects FX2LP from USB, delays for 1500 ms, clears any pending USB interrupts (INT2), reconnects, and returns. The parameter renum sets the EZ-USB renumerate bit in the USB control register (USBCS). If renum is TRUE, the renumerate bit is set and FX2LP firmware is responsible for handling all USB device requests on endpoint 0. This function is called in the FX2LP firmware framework to initiate the ReNumeration process after downloading the application firmware.

6.5.2.3 EZUSB_GetStringDscr()

```
STRINGDSCR xdata * EZUSB_GetStringDscr(BYTE StrIdx)
```

This function returns a pointer to the instance of a string descriptor in the descriptor table. The instance is determined by the StrIdx parameter. If the descriptor table does not contain the given number of instances, then the function returns a NULL pointer. This function is used in the FX2LP firmware framework to pass the string descriptor to the USB host as follows.

```
void   *dscr_ptr;
switch(SETUPDAT[3])
            {
                 case GD_STRING:             // String
                  if(dscr_ptr = (void *)EZUSB_GetStringDscr(SETUPDAT[2]))
                  {
                     SUDPTRH = MSB(dscr_ptr);
                     SUDPTRL = LSB(dscr_ptr);
                  }
           }
```

6.5.2.4 EZUSB_Susp()

```
void EZUSB_Susp(void)
```

This function suspends the processor in response to a USB suspend event. This function will not return until the suspend is cleared by a USB bus resume or a wakeup event on the FX2LP WAKEUP pin. If FX2LP has not detected a suspend event when this function is called, it immediately returns.

6.5.2.5 EZUSB_Resume()

```
void EZUSB_Resume(void)
```

Resume is a signal initiated by the device or host driving a “K” state on the USB bus, requesting that the host or device be taken out of its low-power “suspended” mode. A USB device can only signal a Resume if it has reported (via its configuration descriptor) that it is “remote wakeup capable” and only if the host has enabled remote wakeup from that device. This function generates the K state on the USB bus required for a USB device to signal a remote wakeup. This function should be called following a USB suspend, and it gets executed when FX2LP comes out of suspend due to the assertion of wakeup pins (WAKEUP and WU2).

6.5.2.6 I^2C Routines

```
void EZUSB_InitI2C(void);
BOOL EZUSB_WriteI2C_(BYTE addr, BYTE length, BYTE xdata *dat);
BOOL EZUSB_ReadI2C_(BYTE addr, BYTE length, BYTE xdata *dat);
BOOL EZUSB_WriteI2C(BYTE addr, BYTE length, BYTE xdata *dat);
BOOL EZUSB_ReadI2C(BYTE addr, BYTE length, BYTE xdata *dat);
void EZUSB_WaitForEEPROMWrite(BYTE addr);
```

These functions automate access to I^2C devices (such as the EEPROM), the 7-segment display, and buttons on the DVK board. See the **vend_ax** and **dev_io** firmware examples for details on using these functions.

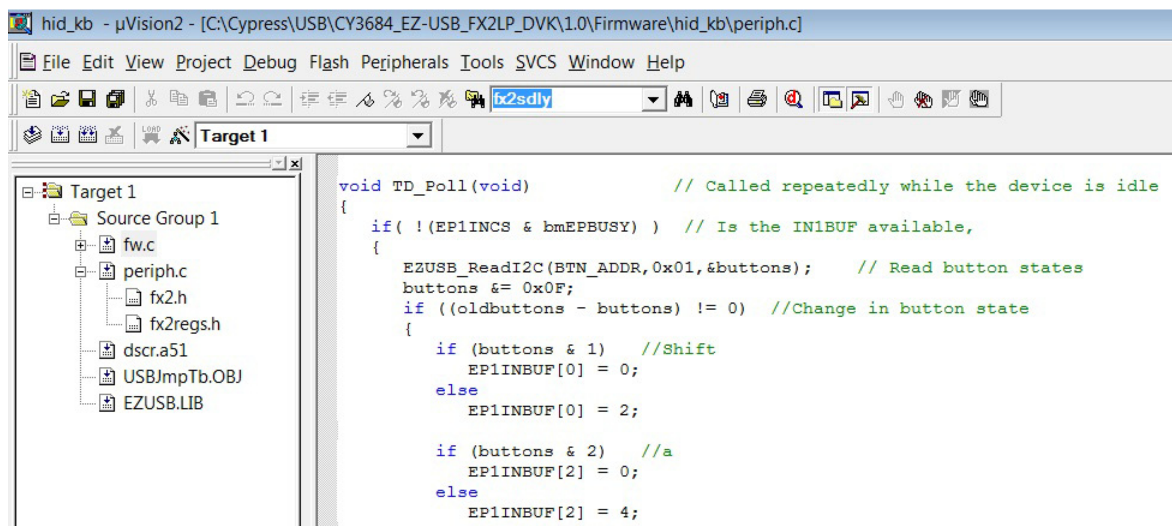
7. Firmware Examples in Detail

This chapter describe the steps to run the firmware examples provided with the DVK installation. These examples run both on CY3684 and CY3674, with the USB speed differences. If you are using CY3684 (FX2LP DVK) then the USB transfers run at High Speed and if you are using CY3674 (FX1 DVK) then the USB transfers run at full speed. Steps to run the firmware examples are valid for both CY3684 and CY3674 even though we showed for CY3684.

7.1 hid_kb Firmware Example

This example describes the implementation of a four-button virtual HID-Class keyboard using the EZ-USB DVK board. Open the ***hid_kb.Uv2*** project file in Keil µVision2 IDE. Figure 7-1 provides the project screen shot in the IDE.

Figure 7-1. hid_kb Project View in keil µVision2 IDE



The firmware example detects if any of the four push buttons are pressed (on the development board) and sends the relevant fixed data to the host PC. For the HID-class devices, such as the keyboard and mouse, the USB bandwidth requirements is typically 64 KB/s. Most of the HID devices are either low-speed or full-speed devices. Due to this low data rate requirement of the device, only the endpoint EP1 (64-byte buffer) is selected for both IN and OUT interrupt transfers. The high-speed data endpoints EP2, EP4, EP6, and EP8 are disabled, as shown in the following code snippet:

```
EP1OUTCFG = 0xB0; // valid, interrupt OUT, 64 bytes, Single buffered
EP1INCFG = 0XB0; // valid, interrupt IN, 64 bytes, Single buffered
EP2CFG = EP4CFG = EP6CFG = EP8CFG = 0; // disable unused endpoints
```

For a typical HID device, the data related to events, such as button press, keystrokes, and mouse clicks, are transferred to the host in the form of input reports using an interrupt IN endpoint. The host PC requests HID reports using the control endpoint or an interrupt OUT endpoint. The firmware sets EP1IN and EP1OUT as interrupt endpoints for data transfers. Table 7-1 summarizes the mapping of push buttons to keyboard buttons on the FX2LP development board.

Table 7-1. Function Mapping of Development Board Buttons

EZ-USB Development Board Push Button	Function
f1	Shift
f2	Send ‘a’
f3	Send ‘b’
f4	Send ‘c’

The function TD_poll () in the firmware (***periph.c***) is where the periodic checking of a new push button event is done. Following is the code snippet of this function.

```
if( !(EP1INCS & bmEPBUSY) ) // Is the EP1INBUF
                            //available,
{
EZUSB_ReadI2C(BTN_ADDR,0x01,&buttons); // Read button states
buttons &= 0x0F;
if ((oldbuttons - buttons) != 0) //Change in button state
{
if (buttons & 1) //Shift
EP1INBUF[0] = 0;
else
EP1INBUF[0] = 2;
if (buttons & 2) //a
EP1INBUF[2] = 0;
else
EP1INBUF[2] = 4;
if (buttons & 4) //b
EP1INBUF[3] = 0;
else
EP1INBUF[3] = 5;
if (buttons & 8) //c
EP1INBUF[4] = 0;
else
EP1INBUF[4] = 6;
EP1INBUF[1] = 0;
EP1INBC = 5;
}
oldbuttons = buttons;
}
```

7.1.1 Building Firmware Example Code for EZ-USB Internal RAM and External EEPROM.

- Click the **Build Target button** at the top right corner of the IDE. The **Build** window of the Keil IDE (Figure 7-2) shows the successful compilation of the entire project.

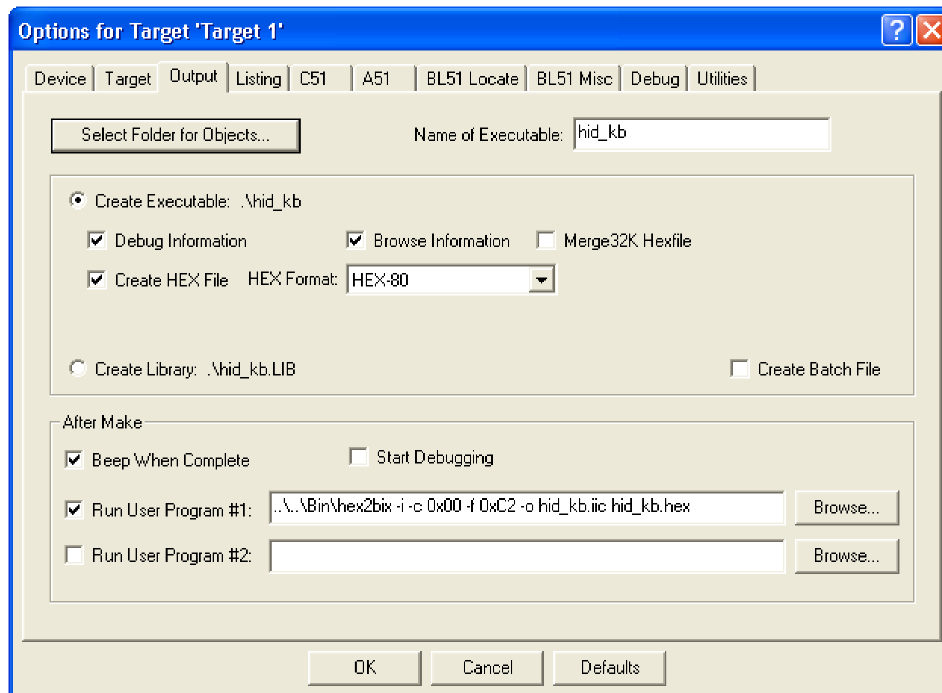
Figure 7-2. Build Window Snapshot of hid_kb Project

```
Build target 'Target 1'
compiling fw.c...
compiling periph.c...
assembling dscr.a51...
linking...
Program Size: data=55.5 xdata=4476 code=2837
creating hex file from "hid_kb"...
User command #1: ..\..\Bin\hex2bix -i -c 0x00 -f 0xC2 -o hid_kb.iic hid_kb.hex
Intel Hex file to EZ-USB Binary file conversion utility
Copyright (c) 2012-2013, Cypress Semiconductor Inc.
2882 Bytes written.
Total Code Bytes = 2837
Conversion completed successfully.
"hid_kb" - 0 Error(s), 0 Warning(s).
```
Build | Command | Find in Files

Note: Observe that the “Total Code Bytes” of the **hid_kb** project is less than the 4-KB code limit of the Keil µVision 2 IDE provided with the kit.

- **Firmware for EZ-USB RAM:** The output of the **Build Target** is *hid_kb.hex*, which is the relevant file for downloading to EZ-USB RAM.
- **Firmware for external I^{2}C EEPROM:** To generate an EEPROM compatible firmware image, the Keil IDE invokes the *hex2bix.exe* utility to convert the output file *hid_kb.hex* to *hid_kb.iic*. Right-click on **Target1** in the project window and select **Options for Target 'Target1**.' This will result in a pop-up of the Keil settings for this project. Select the **Output** tab and observe at the bottom of the IDE, the hex2bix utility is invoked as shown in Figure 7-3.

Figure 7-3. hid_kb Project Output Image Settings



In the **Run User Program #1** section, observe that the hex2bix utility is invoked in the following manner:

```
..\..\Bin\hex2bix -i -c 0x00 -f 0xC2 -o hid_kb.iic hid_kb.hex
```

Refer to *readme.txt* in the Cypress\USB\Util\Hex2Bix folder to learn more about the hex2bix utility.

7.1.2 Method to Download Firmware Image to EZ-USB Internal RAM

1. On the EZ-USB(FX2LP/FX1) board, move the switch **SW2** to the **NO EEPROM** side.
2. Connect the USB A-to-B cable from the J1 connector onboard to a Windows PC USB host controller port.
3. The FX2LP development board should by default bind to the drivers in the folder **<Installed_directory>\<version>\Drivers** for the corresponding OS.
4. Open the USB Control Center application (*CyControl.exe*) from the location **<Installed_directory>\<version>\Windows Applications\Application Source files\c_sharp\controlcenter\bin\Release**. Observe the EZ-USB FX2LP listed as **"Cypress FX2LP Sample Device."**
5. Click on the Cypress device entry to highlight it, and then choose **Program > FX2 > RAM**. In the firmware folder, open the hid_kb folder and double-click the *hid_kb.hex* file. After the code loads, the FX2LP DVK disconnects from USB and reconnects as the device created by the loaded firmware, *hid_kb.hex*.
6. After download, the image does not require a Cypress USB driver for testing the four-button virtual keyboard functionality. Keyboard functionality is handled by the Windows OS native HID drivers.

7.1.3 Method to Download Firmware Image to External I^2C EEPROM

1. On the FX2LP development board, select **SW2-NO EEPROM** and connect the USB A-to-B cable from the J1 connector on the board to a Windows PC USB host controller port. The EZ-USB FX2LP device enumerates with the default VID/PID.

2. Before programming the EEPROM image file (*.iic*), select **SW2-EEPROM** and **SW1-LARGE EEPROM** as switch settings to select large EEPROM U5 on board.

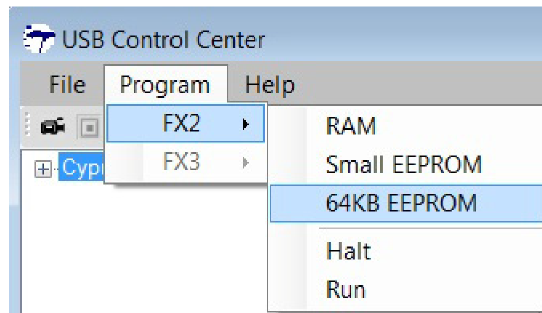
3. Open the USB Control Center application (*CyControl.exe*) from the location **<Installed_directory>\<version>\Windows Applications\Application Source files\c_sharp\controlcenter\bin\Release**. Observe the EZ-USB FX2LP listed as **"Cypress FX2LP Sample Device"**.

4. Click on the Cypress device entry to highlight it, and then choose **Program > FX2 > 64KB EEPROM** as shown in Figure 7-4. In the firmware folder, open the hid_kb folder and double-click the *hid_kb.iic* file. The USB Control Center window shows the successful completion of image download to large EEPROM U5-24LC128

5. Press the RESET button, S1, again and this prompts the EZ-USB device to boot from the large EEPROM Image *hid_kd.iic*.

6. After download, the image does not require a Cypress USB driver for testing the four-button virtual keyboard functionality. The complete functionality is handled by the Windows OS native HID drivers.

Figure 7-4. USB Control Center Display to Download Image to Large EEPROM



7.1.4 Binding Cypress USB Driver for the Downloaded Firmware Image

The hib_kb project contains firmware for a HID-class keyboard device (interface class: HID = 03 and subclass = 00) and uses the Microsoft native HID driver, instead of Cypress generic USB driver.

7.1.5 Testing the hid_kb Firmware Example Functionality

The EZ-USB development board enumerates as a human interface device (HID). Open the Device Manager window by typing devmgmt.msc in **Start > Run**. In Windows Vista and Windows 7 OS platforms, type *devmgmt.msc* directly in the vacant box near the **Start** button. The device will be shown as part of the HID devices list. Open Notepad in Windows and click in the text area. Press buttons F2, F3, and F4 sequentially and observe the letters “a, b, c” as they appear in the window. Press them simultaneously with F1 and observe the appearance of the letters “A, B, C” in the Notepad window. Press the Screen Lock, Caps Lock and Num Lock keyboard keys and observe that three of the 7-segment bars light along with the keyboard indicators.

7.2 IBN Firmware Example

7.2.1 Description

This example illustrates the configuration of EZ-USB to accept bulk data from the host and loop it back to the host using an IN-BULK-NAK (IBN) interrupt. Click on the **ibn.Uv2** project file at **<Installed_directory>\<Version>\Firmware\ibn**. In the TD_init() function of the ***ibn.c*** file four endpoints are configured to handle bulk transfer: two OUT endpoints and two IN endpoints. The four endpoints defined in the descriptor file must be configured in this function with the following statements:

```
EP2CFG = 0xA2;
SYNCDELAY;
EP4CFG = 0xA0;
SYNCDELAY;
EP6CFG = 0xE2;
SYNCDELAY;
EP8CFG = 0xE0
```

The key characteristics of each endpoint are as follows:

- Endpoint 2: OUT, bulk, double-buffered
- Endpoint 4: OUT, bulk, double-buffered
- Endpoint 6: IN, bulk, double-buffered
- Endpoint 8: IN, bulk, double-buffered

Writing to these registers typically takes more than the two clock cycles needed for a MOVX instruction. Therefore, the SYNCDELAY macro is added. The TRM provides the list of registers that need this delay function when writing to them. After they are configured, the OUT endpoints need to be armed to accept packets from the host. Because the endpoints are double buffered, you must arm the endpoint twice. Arming essentially frees up the buffers, making them available to receive packets from the host.

By writing a ‘1’ to bit7 of the byte count register, the endpoint is armed.

```
EP2BCL = 0x80; // arm EP2OUT by writing byte count w/skip.
SYNCDELAY;
EP2BCL = 0x80;
SYNCDELAY;
EP4BCL = 0x80; // arm EP4OUT by writing byte count w/skip.
SYNCDELAY;
EP4BCL = 0x80;
```

The previous lines arm the two OUT endpoints by skipping two packets of data, making the buffers available to receive OUT data.

```
NAKIRQ = bmBIT0; // clear the global IBN IRQ
NAKIE |= bmBIT0; // enable the global IBN IRQ
IbnFlag = 0x00; // clear our IBN flag
IBNIRQ = 0xFF; // clear any pending IBN IRQ
IBNIE |= bmEP6IBN | bmEP8IBN; // enable the IBN interrupt
                                 for EP6 and EP8
```

The firmware clears the IBN flags of all endpoints and any pending IBN interrupts and enables the IBN interrupt for EP6 and EP8.

```
AUTOPTRSETUP |= 0x01;
```

This enables the auto-pointer used for data transfer in the TD_Poll() function. The loopback is implemented in the TD_Poll function, which is called repeatedly when the device is idle. Endpoints 2

and 4 are armed to accept data from the host. This data is transferred to endpoint 6 and endpoint 8 respectively. To implement this, endpoint 2 is first checked to see if it has data by reading the endpoint 2 empty bit in the endpoint status register (EP2468STAT). If endpoint 2 has data (sent from the host), then check if the host has requested data on EP6 by reading the EP6 In-Bulk-Flag bit in the IbnFlag variable. If the host has requested data on EP6, then the data is transferred.

This decision is executed by the following statement:

```
if (!(EP2468STAT & bmEP2EMPTY) && (IbnFlag & bmEP6IBN) )
// if there is new data in EP2FIFOBUF and the IBN flag for EP6 has been
set, //then copy the data from EP2 to EP6
```

The data transfer is carried out by the execution of the following loop:

```
for( i = 0x0000; i < count; i++ )
{
// setup to transfer EP2OUT buffer to EP6IN buffer using AUTOPOINTER(s)
EXTAUTODAT2 = EXTAUTODAT1;
}
```

As auto pointers are enabled, the pointers increment automatically.

```
EXTAUTODAT2 = EXTAUTODAT1;
```

After this statement transfers the data, endpoint 2 has to be rearmed to accept a new packet from the host. Endpoint 6 has to be committed, making the FIFO buffers available to the host for reading data from endpoint 6. This is accomplished by the following statements:

This is accomplished by the following statements:

```
EP6BCH = EP2BCH;
SYNCDELAY;
EP6BCL = EP2BCL; // commit EP6IN by specifying the number of bytes the
host can read //from EP6
SYNCDELAY;
EP2BCL = 0x80; // re (arm) EP2OUT
```

The EP6 IBN flag bit in the IbnFlag variable is cleared. The EP6 IBN interrupt request is cleared by setting the corresponding bit in the IBNIRQ register. Finally, the EP6 IBN interrupt is enabled by setting the corresponding bit in the IBNIE register.

```
IbnFlag &= ~bmEP6IBN; // clear the IBN flag
IBNIRQ = bmEP6IBN; // clear the IBN IRQ
IBNIE |= bmEP6IBN; // enable the IBN IRQ
```

The same operation is carried out to implement a data loop with endpoints 4 and 8.

When the host requests an IN packet from an EZ-USB bulk endpoint, the endpoint NAKs (returns the NAK PID) until the endpoint buffer is filled with data and armed for transfer, at which point the EZ-USB at which point FX2LP answers the IN request with data. Until the endpoint is armed, a flood of IN-NAKs can tie up bus bandwidth. Therefore, if the IN endpoints are not always kept full and armed, it may be useful to know when the host is "knocking at the door, requesting IN data." The IBN interrupt provides this notification. It fires whenever a bulk endpoint NAKs an IN request. The IBNIE/ IBNIRQ registers contain individual enable and request bits for each endpoint, and the NAKIE/ NAKIRQ registers each contain a single-bit, IBN, that is the ORed combination of the individual bits in IBNIE/IBNIRQ, respectively. The EZ-USB FX2LP firmware framework provides hooks for all the interrupts that it implements. The example project uses the ISR_Ibn ISR to handle the IBN interrupt for EP6 and EP8.

```
void ISR_Ibn(void) interrupt 0
{
```

```
int i;
// disable IBN for all endpoints
IBNIE = 0x00;
EZUSB_IRQ_CLEAR(); // clear the global USB IRQ
// Find the EP with its IBN bit set
for (i=0;i<8;i++)
{
if (IBNIRQ & (1 << i))
{
IbnFlag |= (1 << i); // set the appropriate IBN flag bit
IBNIRQ |= (1 << i); // clear the IBN IRQ for this endpoint
}
}
NAKIRQ |= bmBIT0; // clear the global IBN IRQ
// re-enable IBN interrupt for any endpoints that don't already have
// an IBN pending in IbnFlag
IBNIE = (bmEP6IBN | bmEP8IBN) & ~IbnFlag;
}
```

7.2.2 Building Firmware Example Code for EZ-USB RAM and EEPROM

Click on **Build Target** at the top right corner of the IDE. The firmware example builds successfully since the “Total Code Bytes” of the IBN firmware example is less than the 4-KB code limit of the Keil µVision2 IDE provided with the kit. The output of the **Build Target** is the *ibn.hex* and *ibn.iic* files.

7.2.3 Method to Download Firmware Image to EZ-USB Internal RAM and External EEPROM

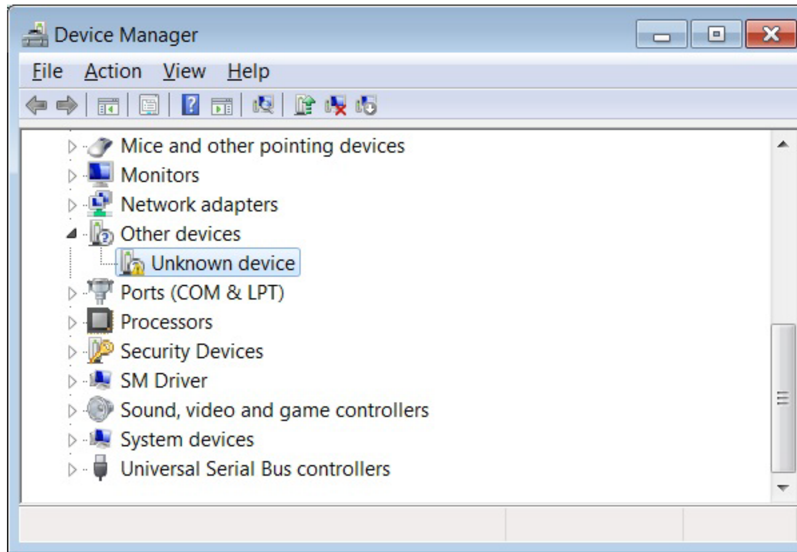
Refer to the sections Method to Download Firmware Image to EZ-USB Internal RAM on page 54 and Method to Download Firmware Image to External I2C EEPROM on page 54 and follow a similar procedure to download either *ibn.hex* to the RAM or *ibn.iic* to the large EEPROM using the USB Control Center. After download, the firmware re-enumerates with the PC using its internal VID/PID x04B4/0x1004.

7.2.4 Binding Cypress USB Driver for the Downloaded Firmware Image

The **IBN** project uses vendor-class (0xFF) with VID/PID 0x04B4/1004. This example should bind with the Cypress generic USB driver, *CyUSB3.sys*, and the driver information file, *CyUSB3.inf*, which contains the relevant VID/PID of this example. To manually install the driver, follow these steps.

1. In Windows, invoke **Start > Computer** and right-click **Properties > Select Device Manager**. Locate the FX2LP device entry with a yellow symbol in the "Other devices" list, as shown in Figure 7-5.

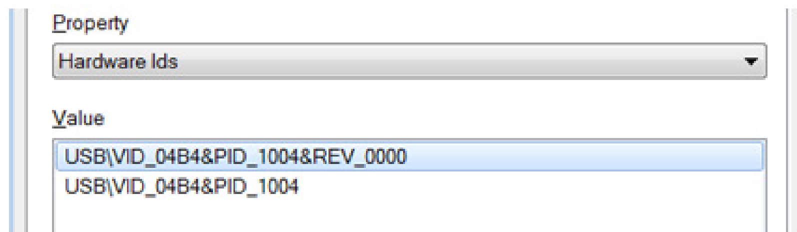
Figure 7-5. Device Manager showing the presence of new device connection



Note: In Windows XP invoke **Start > My Computer**, right-click "Properties" click the **Hardware** tab, and select "Device Manager."

2. Right-click on the yellow device entry to verify the VID/PID of the device. Choose **Properties > Details**. Select "Hardware Ids" and observe if the default VID/PID is 0x04B4/0x1004, as shown in Figure 7-6.

Figure 7-6. VID/PID Values



3. Right-click on the device entry (with a PID of 0x1004) and select **Browse my computer for driver software**. On a 64-bit Windows 7 machine, the driver is located at **<Installed_directory>\<version>\Drivers\win7\x64**.

7.2.5 Testing the IBN Firmware Functionality

Following are the detailed steps to test the functionality.

1. After the board enumerates, use the USB Control Center to send 512 bytes from EP2 to EP6. The data received should be the same as the data sent. The host can send 512 bytes of user-defined data to endpoint 2 using the USB Control Center. For example, select **"Bulk out endpoint (0x02)"** in the left pane of the USB Control Center and select the **Data Transfers** tab on the right. In the **Data Transfers** tab, enter the **Bytes to Transfer** as '512' and then click the **Transfer Data-OUT** button.
2. This data can be read back from endpoint 6 using the USB Control Center. For example, select **"Bulk in endpoint (0x86)"** in the left pane of the USB Control Center and select the **Data Transfers** tab on the right. In the **Data Transfers** tab, enter the number of **Bytes to Transfer** as '512' and then click the **Transfer Data-IN** button to read back the data. Similarly, loopback using endpoints 4 and 8 can also be tested. Since EP2 and EP4 are double buffered, they can contain only two packets of data.
3. On sending a packet to these endpoints when both the buffers are full, the endpoints NAK the transfer because there is no space available. If an IN transfer is requested on either EP6 or EP8, the corresponding IBN interrupt is asserted and data is transferred from EP2 to EP6 or from EP4 to EP8. This data appears in the USB Control Center window.
4. You can test the previous steps by trying to send data to EP2 and EP4 without reading the data out of EP6 or EP8. After the first two transfers, all the successive OUT transfers fail. This persists until an IN transfer is made on EP6 or EP8.

   Note: For EZ-USB FX1, the previous steps can be repeated with a data transfer length of 64 bytes instead of 512 bytes.

7.3 Pingnak Firmware Example

7.3.1 Description

This project illustrates the configuration of the EZ-USB device to accept bulk data from the host and loop it back to the host and the use of the PING-NAK interrupt. Click on ***pingnak.Uv2*** located at **<Installed_directory>\<Version>\Firmware\pingnak** and observe the code. Four endoints are configured in the TD_init() function of ***pingnak.c*** to handle bulk transfer: two OUT endoints and two IN endpoints. The four endpoints defined in the descriptor file must be configured in this function with the following statements:

```
EP2CFG = 0xA2;
SYNCDELAY;
EP4CFG = 0xA0;
SYNCDELAY;
EP6CFG = 0xE2;
SYNCDELAY;
EP8CFG = 0xE0
```

The key characteristics of each endpoint are as follows:

- Endpoint 2: OUT, bulk, double buffered
- Endpoint 4: OUT, bulk, double buffered
- Endpoint 6: IN, bulk, double buffered
- Endpoint 8: IN, bulk, double buffered

Writing to these registers typically takes more than the two clock cycles needed for a MOVX instruction. Therefore, the SYNCDELAY macro is added. The EZ-USB Technical Reference Manual at **<Installed_directory>\<Version>\Documentation** provides the list of registers that need this

delay function when writing to them. After they are configured, the OUT endpoints need to be armed to accept packets from the host. Because the endpoints are double buffered, you must arm the endpoint twice. Arming essentially frees up the buffers, making them available to receive packets from the host. By writing a ‘1’ to bit 7 of the byte count register, the endpoint is armed.

```
EP2BCL = 0x80; // arm EP2OUT by writing byte count w/skip.
SYNCDELAY;
EP2BCL = 0x80;
SYNCDELAY;
EP4BCL = 0x80; // arm EP4OUT by writing byte count
               //w/skip.
SYNCDELAY;
EP4BCL = 0x80;
```

After configuration, the OUT endpoints are 'armed' to accept data from the host. An OUT endpoint is said to be armed if it is ready to accept data from the host. Each endpoint is configured as double buffered. The OUT endpoints are armed by setting the skip bit in the byte count registers. This leaves them empty to receive a new packet from the host. It also clears any pending PING-NAK interrupts and enables the PING-NAK interrupt for EP2 and EP4. The loopback is implemented in the TD_Poll() function that is called repeatedly when the device is idle. Endpoints 2 and 4 are armed to accept data from the host. This data is transferred to endpoint 6 and endpoint 8 respectively. First, endpoint 2 is checked to see if it has data by reading the endpoint 2 empty bit in the endpoint status register (EP2468STAT). If endpoint 2 has data (sent from the host), the capability of endpoint 6 to receive the data is checked by reading the endpoint 6 full bit in the endpoint status register. If endpoint 6 is not full, then the data is transferred. This decision is executed by the following statements:

```
if (!(EP2468STAT & bmEP2EMPTY))
{// check EP2 EMPTY (busy) bit in EP2468STAT (SFR), core set's this bit
when
// FIFO is empty
if (!(EP2468STAT & bmEP6FULL))
{// check EP6 FULL (busy) bit in EP2468STAT (SFR), core set's this bit
// when FIFO is full
```

The data pointers are initialized to the corresponding buffers. The first auto-pointer is initialized to the first byte of the endpoint 2 FIFO buffer. The second auto-pointer is initialized to the first byte of the endpoint 6 FIFO buffer. The number of bytes to be transferred is read from the byte count registers of endpoint 2. The registers EP2BCL and EP2BCH contain the number of bytes written into the FIFO buffer by the host. These two registers give the byte count of the data transferred to the FIFO in an OUT transaction as long as the data is not committed to the peripheral side. This data pointer initialization and loading of the count is done in the following statements:

```
APTR1H = MSB( &EP2FIFOBUF ); // Initializing the first data pointer
APTR1L = LSB( &EP2FIFOBUF );
AUTOPTRH2 = MSB( &EP6FIFOBUF ); // Initializing the second data pointer
AUTOPTRL2 = LSB( &EP6FIFOBUF );
count = (EP2BCH << 8) + EP2BCL; // The count value is loaded from the byte
// count registers
```

The data transfer is carried out by the execution of the following loop:

```
for( i = 0x0000; i < count; i++ )
{
// setup to transfer EP2OUT buffer to EP6IN buffer using AUTOPOINTER(s)
EXTAUTODAT2 = EXTAUTODAT1;
}
```

Because auto-pointers have been enabled, the pointers increment automatically, and the statement

```
EXTAUTODAT2 = EXTAUTODAT1;
```

transfers data from endpoint 2 to endpoint 6. Each time the statement is executed, the auto-pointer is incremented. It is executed repeatedly to transfer each byte from endpoint 2 to endpoint 6. After the data is transferred, endpoint 2 has to be rearmed to accept a new packet from the host. Endpoint 6 has to be committed, making the FIFO buffers available to the host for reading data from endpoint 6. This is accomplished by the following statements:

```
EP6BCH = EP2BCH;
SYNCDELAY;
EP6BCL = EP2BCL; // commit EP6IN by specifying the number of bytes the
host can read //from EP6
SYNCDELAY;
EP2BCL = 0x80; // re (arm) EP2OUT
```

The same operation is carried out to implement a data loop with endpoints 4 and 8.

High-speed USB implements a PING-NAK mechanism for (bulk and control) OUT transfers. When the host wishes to send OUT data to an endpoint, and the previous data transfer was answered with a NYET, it first sends a PING token to see if the endpoint is ready (for example, if it has an empty buffer). If a buffer is not available, the FX2LP returns a NAK handshake. PING-NAK transactions continue to occur until an OUT buffer is available, at which time the FX2LP answers a PING with an ACK handshake and the host sends the OUT data to the endpoint. EZ-USB implements PING-NAK interrupt as EP0PING, EP1PING, and so on, one for each endpoint. The EPxPING interrupt is asserted when the host PINGs an endpoint and the FX2LP responds with a NAK because the particular endpoint buffer memory is not available. The FX2LP firmware framework provides hooks for all the interrupts that it implements. The example project uses the ISR_Ep2pingnak and ISR_Ep4pingnak ISRs to handle EP2PING and EP4PING interrupts respectively.

```
void ISR_Ep2pingnak(void) interrupt 0
{
SYNCDELAY; // Re-arm endpoint 2
EP2BCL = 0x80;
EZUSB_IRQ_CLEAR(); // clear the EP2PING interrupt
NAKIRQ = bmEP2PING;
}
```

The ISR_Ep2pingnak discards the previous data that is stored in one of the buffers of endpoint 2 by rearming the endpoint (that is, EP2BCL = 0x80). Therefore, EP2 can now receive the data that is currently being sent by the host because there is space available in one of its buffers. It then clears the interrupt by setting a particular bit in NAKIRQ because it has been serviced. The same operation is carried out to service the EP4PING interrupt in ISR_Ep4pingnak.

7.3.2 Building Firmware Example Code for EZ-USB RAM and EEPROM

Click the **Build Target** button at the top right corner of the IDE. The “Total Code Bytes” of the pingnak firmware example is less than the 4-KB code limit Keil µVision2 IDE provided with the kit. The output of the **Build Target** is *pingnak.hex* and *pingnak.iic* files.

7.3.3 Method to Download Firmware Image to EZ-USB Internal RAM and External EEPROM

Refer to Method to Download Firmware Image to EZ-USB Internal RAM on page 54 and Method to Download Firmware Image to External I2C EEPROM on page 54 and follow a similar procedure to download either *pingnak.hex* to RAM or *pingnak.iic* to the large EEPROM using the USB Control Center. Both images are located at **<Installed_directory>\<Version>\Firmware\pingnak**. After downloading, the firmware re-enumerates with the PC using its internal VID/PID 0x04B4/0x1004.

7.3.4 Binding Cypress USB Driver for the Downloaded Firmware Image

The pingnak project uses vendor class (0xFF) with VID/PID 0x04B4/1004. This example should bind with the Cypress generic USB driver, *cyUSB3.sys*, and the driver information file, *CyUSB3.inf,* which contains the relevant VID/PID of this example. Follow the procedure outlined to manually bind the driver using the Windows Hardware Wizard. If you have performed the binding process for any one of the firmware examples, you can skip it for this example.

7.3.5 Testing the pingnak Firmware Functionality

Follow these steps to test the pingnak firmware with a USB 2.0 connection.

1. After the board re-enumerates, use the USB Control Center to send 512 bytes from EP2 to EP6. The data received should be the same as the data sent. You can send 512 bytes of user-defined data from the host to endpoint 2 using the USB Control Center. For example, select **"Bulk out endpoint (0x02)"** in the left pane of the USB Control Center and select the **Data Transfers** tab on the right. In the **Data Transfers** tab, enter the **Bytes to Transfer** as '512' and then click the **Transfer Data-OUT** button.
2. This data can be read back from endpoint 6 using the USB Control Center. For example, select **"Bulk in endpoint (0x86)"** in the left pane of the USB Control Center and select the **Data Transfers** tab on the right. In the **Data Transfers** tab, enter the number of **Bytes to Transfer** as '512' and then click the **Transfer Data-IN** button to read back the data. Similarly, loopback using endpoint 4 and 8 can also be tested. Because EP2 and EP4 are double buffered, they can contain only two packets of data. After sending a packet to these endpoints when both the buffers are full, the endpoints NAK the transfer because there is no space available. This asserts the PING-NAK interrupt of the NAKing endpoint.
3. The ISRs that handle the PING-NAK interrupt (ISR_Ep2pingnak and ISR_Ep4pingnak) discard the previous data that is stored in one of the endpoint buffers by rearming the endpoint. Therefore, the endpoints can receive the data that is currently sent by the host because there is space in one of its buffers.
4. You can test the previous steps by continuously sending data to EP2 and EP4 without reading the data out of EP6 or EP8. Because the PING-NAK ISR rearms the endpoints, you can continuously transmit data to EP2 and EP4, and the transfer always succeeds. The data present in the buffers of EP2 and EP4 at any point of time will be the latest two packets of data sent from the host.

7.4 Bulkloop Example

7.4.1 Description

This project illustrates the configuration of FX2LP to accept bulk data from the host and loop it back to the host. Click on ***bulkloop.Uv2*** at **<Installed_directory>\<Version>\Firmware\Bulkloop** and observe the source code. Four endpoints are configured in the TD_init() function of ***bulkloop.c*** to handle bulk transfer: one OUT endpoint and one IN endpoint. The two endpoints defined in the descriptor file have to be configured in this function in the following statements:

```
EP2CFG = 0xA2;
SYNCDELAY;
EP6CFG = 0xE2;
SYNCDELAY;
```

The key characteristics of each endpoint are as follows:

- Endpoint 2 - OUT, bulk, double buffered
- Endpoint 6 - IN, bulk, double buffered

After configuration, the OUT endpoints are armed to accept data from the host. An OUT endpoint is said to be armed if it is ready to accept data from the host. Each endpoint is configured as double buffered.

```
SYNCDELAY;
EP2BCL = 0x80;// arm EP2OUT by writing byte count
                                w/skip.
SYNCDELAY;
EP2BCL = 0x80;
SYNCDELAY;
```

The previous lines arm the two OUT endpoints by skipping two packets of data, making the buffers available to receive OUT data.

```
AUTOPTRSETUP |= 0x01;
```

This enables the auto-pointer used for data transfer in the TD_Poll function. The data loopback is implemented in the TD_Poll function that is called repeatedly when the device is idle. Endpoint 2 is armed to accept data from the host. This data is transferred to endpoint 6. First, endpoint 2 is checked to see if it has data by reading the endpoint 2 empty bit in the endpoint status register (EP2468STAT). If endpoint 2 has data (sent from the host), the capability of endpoint 6 to receive the data is checked by reading the endpoint 6 full bit in the endpoint status register. If endpoint 6 is not full, then the data is transferred. This decision is executed by the following statements:

```
if (!(EP2468STAT & bmEP2EMPTY))
{// check EP2 EMPTY (busy) bit in EP2468STAT (SFR), core set's this bit
when
// FIFO is empty
if (!(EP2468STAT & bmEP6FULL))
{// check EP6 FULL (busy) bit in EP2468STAT (SFR), core set's this bit
// when FIFO is full
```

The data pointers are initialized to the corresponding buffers. The first auto-pointer is initialized to the first byte of the endpoint 2 FIFO buffer. The second auto-pointer is initialized to the first byte of the endpoint 6 FIFO buffer. The number of bytes to be transferred is read from the byte count registers of endpoint 2. The registers EP2BCL, EP2BCH contain the number of bytes written into the FIFO buffer by the host. These two registers give the byte count of the data transferred to the FIFO in an OUT transaction as long as the data is not committed to the peripheral side. This data pointer initialization and loading of the count is done in the following statements:

```
APTR1H = MSB( &EP2FIFOBUF ); // Initializing the first data pointer
APTR1L = LSB( &EP2FIFOBUF );
AUTOPTRH2 = MSB( &EP6FIFOBUF ); // Initializing the second data pointer
AUTOPTRL2 = LSB( &EP6FIFOBUF );
count = (EP2BCH << 8) + EP2BCL; // The count value is loaded from the byte
// count registers
```

The data transfer is carried out by the execution of the following loop:

```
for( i = 0x0000; i < count; i++ )
{
// setup to transfer EP2OUT buffer to EP6IN buffer using AUTOPOINTER(s)
EXTAUTODAT2 = EXTAUTODAT1;
}
```

Because auto-pointers have been enabled, the pointers increment automatically, and the statement

```
EXTAUTODAT2 = EXTAUTODAT1;
```

transfers data from endpoint 2 to endpoint 6. Each time the statement is executed, the auto-pointer is incremented. It is executed repeatedly to transfer each byte from endpoint 2 to 6.

After the data is transferred, endpoint 2 has to be rearmed to accept a new packet from the host. Endpoint 6 has to be committed to make the FIFO buffers available to the host for reading data from endpoint 6.

This is accomplished by the following statements:

```
EP6BCH = EP2BCH;
SYNCDELAY;
EP6BCL = EP2BCL; // commit EP6IN by specifying the number of bytes the
host can read //from EP6
SYNCDELAY;
EP2BCL = 0x80; // re (arm) EP2OUT
```

7.4.2 Building Bulkloop Firmware Example Code for EZ-USB RAM and EEPROM

- Click on **Build Target** button at the top right corner of the IDE. Figure 7-7 displays a snapshot of the **Build** window of the Keil IDE, showing the successful compilation of the entire project.

Figure 7-7. Output Window Snapshot of Bulkloop Project Build

Note Observe in Figure 7-7 that the total "code" bytes of the bulkloop project is less than the 4-KB code limit Keil µVision2 IDE provided with the kit.

- **Firmware output for EZ-USB RAM:** The output of the **Build Target** is *bulkloop.hex* which is the relevant file for downloading to EZ-USB RAM.
- **Firmware output for external EEPROM:** To generate an EEPROM-compatible firmware image, the Keil IDE invokes the *hex2bix.exe* utility to convert the output file *bulkloop.hex* into *bulkloop.iic*. Right-click on "**Target1**" in the project window and select **Options for Target 'Target1'**.This will result in a pop-up of the Keil settings for this project. Select the **Output** tab and observe at the bottom of IDE that the hex2bix utility is invoked in the **Run User program#1** section and that the **hex2bix** utility is invoked in the following manner

```
..\..\Bin\hex2bix -i -c 0x00 -f 0xC2 -o bulkloop.iic bulkloop.hex
```

Refer to the *readme.txt* in the Cypress\USB\Util\Hex2Bix folder to learn more about the hex2bix utility

7.4.3 Method to Download Bulkloop Firmware Image to Internal RAM or EEPROM

Refer to Method to Download Firmware Image to EZ-USB Internal RAM on page 54 and Method to Download Firmware Image to External I2C EEPROM on page 54 and follow a similar procedure to download ***bulkloop.hex*** to RAM and *bulkloop.iic* to the 64-KB EEPROM using **USB Control Center**. The ***bulkloop.hex*** and ***bulkloop.iic*** files located at **<Installed_directory>\<Version>\Firmware\Bulkloop** must be chosen accordingly for EZ-USB FX1 and FX2LP. After downloading, the firmware re-enumerates with PC using its internal VID/PID-0x04B4/0x1004.

7.4.4 Binding Cypress USB Driver for the Downloaded Firmware Image

The bulkloop firmware uses vendor class (0xFF) with VID/PID 0x04B4/1004. This example should bind with Cypress generic USB driver ***cyUSB3.sys*** and driver information file *CyUSB3.inf*, which contains the relevant VID/PID of this example. Follow the procedure outlined to manually bind the driver using the Windows Hardware Wizard. If you have performed the binding process for any one of the firmware examples, you can skip it for this example.

7.4.5 Testing the Bulkloop Firmware Functionality

The bulkloop firmware functionality can be tested using the following applications, which are provided with the kit installation. These are taken from the SuiteUSB package.

- USB Control Center
- Bulkloop (C++)
- Bulkloop (C# .NET)

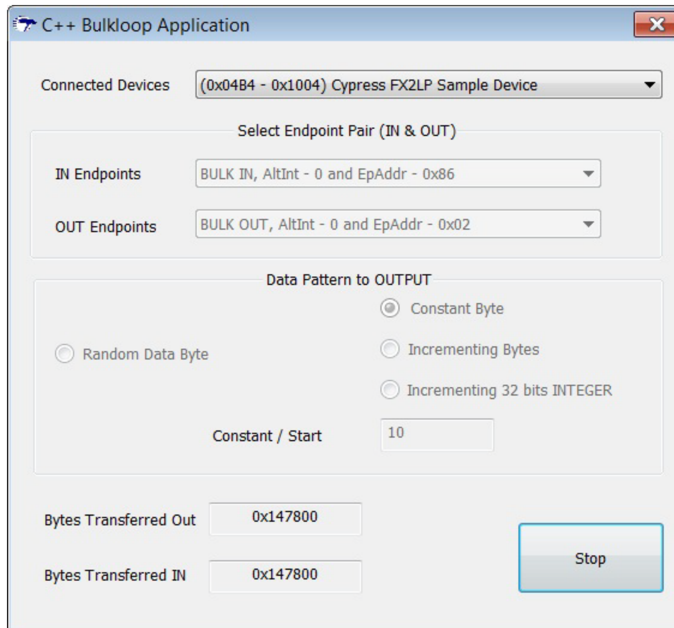
7.4.5.1 Test Using USB Control Center PC Application

Steps to test the bulkloop firmware using the USB Control Center application are described in My First USB 2.0 Transfer using FX2LP chapter on page 13.

7.4.5.2 Test using Bulkloop C++ Application

The bulkloop firmware can be tested using this C++ application called “bulkloop”. For the 32-bit Windows OS, bulkloop can be accessed at **<Installed_directory>\<version>\Windows Applications\Application Source files\cpp\bulkloop\x86\Release**. Select the EZ-USB device in the drop down menu and also select any one the bulk endpoint pairs: EP2/EP6 or EP4/EP8. Please note that the bulkloop firmware provided with this kit supports only the EP2/EP6 pair. Figure 7-8 summarizes the entire operation. You can select different data patterns of bulk USB packets under **Send Data pattern** and enter a maximum transfer size up to 2048 bytes.

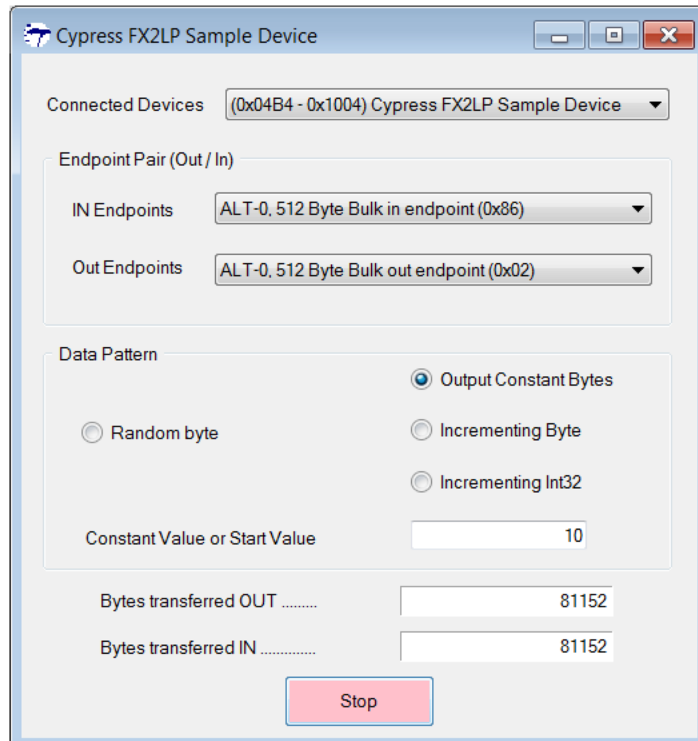
Figure 7-8. Bulkloop C++ Application



7.4.5.3 *Testing Bulkloop Example using Bulkloop C# .NET Application*

The bulkloop firmware can be tested using the Bulkloop C# .NET application (Figure 7-9), which is located at **<Installed_directory>\<Version>\1.1\Windows Applications\Application Source files\c_sharp\bulkloop\bin\Release**. Select the bulkloop OUT and bulkloop IN endpoint pairs EP2 and EP6. Click **Start** and observe the number of successful bulk IN and bulk OUT transfers.

Figure 7-9. Data loop back using Bulkloop C# .NET Application



7.5 Bulksrc Firwmare Example

7.5.1 Description

This project illustrates the configuration of the EZ-USB device to accept bulk data from the host and loop it back to the host. Click on the ***bulksrc.Uv2*** project located at **<Installed_directory>\<Version>\Firmware\Bulksrc** and observe the code. Five endpoints are configured in the TD_init() function of ***bulksrc.c*** to handle bulk transfer. Two OUT (EP2/EP4) endpoints and two IN (EP6/EP8) endpoints are double-buffered pairs. The fifth endpoint is EP1, which acts as both the bulk IN and bulk OUT endpoint with a 64-byte buffer. These are defined in the descriptor file **(*dscr.a51*)**. The endpoints are configured in this TD_init function in the following statements:

```
EP1OUTCFG = 0xA0; EP1INCFG = 0xA0;
SYNCDELAY;
EP2CFG = 0xA2;
SYNCDELAY;
EP4CFG = 0xA0;
SYNCDELAY;
EP6CFG = 0xE2;
SYNCDELAY;
EP8CFG = 0xE0;
```

After configuration, the OUT endpoints are armed to accept data from the host. An OUT endpoint is said to be armed if it is ready to accept data from the host. Each endpoint is configured as double buffered.

```
/* since the defaults are double buffered we must write dummy byte counts
twice */
SYNCDELAY;
EP2BCL = 0x80;                  // arm EP2OUT by writing byte count w/skip.
SYNCDELAY;
EP4BCL = 0x80;
SYNCDELAY;
EP2BCL = 0x80;             /* arm EP4OUT by writing byte count w/skip. */
SYNCDELAY;
EP4BCL = 0x80;
```

The previous lines arm the two OUT endpoints by loading byte counts for two data packets with the skip bit set, making the buffers available to receive OUT data.

The IN endpoint, EP6, is armed with an incrementing pattern of data starting with 0x2, regardless of the data sent on the EP2 bulk OUT endpoint, as shown in the following code.

```
for (i=0;i<512;i++) EP6FIFOBUF[i] = i+2;
SYNCDELAY;
EP6BCH = 0x02;
SYNCDELAY;
EP6BCL = 0x00;
}
```

In the TD_poll() function, if there is packet content in EP2, then it is rearmed, discarding the current data.

```
// if there is some data in EP2 OUT, re-arm it
if(!(EP2468STAT & bmEP2EMPTY))
{
```

```
SYNCDELAY;
EP2BCL = 0x80;
}
```

Endpoint EP6 is rearmed with an incrementing pattern of data starting with 0x2.

```
// if EP6 IN is available, re-arm it
If(!(EP2468STAT & bmEP6FULL))
{
SYNCDELAY; EP6BCH = 0x02; SYNCDELAY; EP6BCL = 0x00;
}
```

The contents received from the EP4 OUT endpoint are copied to a temporary buffer, myBuffer[], and rearmed.

```
// if there is new data in EP4FIFOBUF, then copy it to a temporary buffer
if(!(EP2468STAT & bmEP4EMPTY))
{
APTR1H = MSB( &EP4FIFOBUF ); APTR1L = LSB( &EP4FIFOBUF );

AUTOPTRH2 = MSB( &myBuffer ); AUTOPTRL2 = LSB( &myBuffer );

myBufferCount = (EP4BCH << 8) + EP4BCL;

for( i = 0x0000; i < myBufferCount; i++ )
{
EXTAUTODAT2 = EXTAUTODAT1;
}

SYNCDELAY;
EP4BCL = 0x80;// re(arm) EP4OUT
}
```

If the EP8 bulk IN endpoint is empty, then the contents of the temporary buffer are transferred to an auto-pointer and copied to the EP8 IN buffer, as shown in the following code.

```
/* if there is room in EP8IN, then copy the contents of the temporary
buffer to it */
if(!(EP2468STAT & bmEP8FULL) && myBufferCount)
{
APTR1H = MSB( &myBuffer ); APTR1L = LSB( &myBuffer );

AUTOPTRH2 = MSB( &EP8FIFOBUF ); AUTOPTRL2 = LSB( &EP8FIFOBUF );

for( i = 0x0000; i < myBufferCount; i++ )
{
/* setup to transfer EP4OUT buffer to EP8IN buffer using AUTO- POINTER(s)
in SFR space */
EXTAUTODAT2 = EXTAUTODAT1;
}

SYNCDELAY;
EP8BCH = MSB(myBufferCount); SYNCDELAY;
EP8BCL = LSB(myBufferCount);// arm EP8IN
}
```

7.5.2 Building Bulksrc Firmware Example Code for EZ-USB RAM and EEPROM

Click the **Build Target** button at the top right corner of the IDE. The “Total Code Bytes” of the **Bulksrc** firmware example is less than the 4-KB code limit of the Keil µVision2 IDE, provided with the kit. The output of the **Build Target** is the *bulkext.hex* and *bulkext.iic* files.

7.5.3 Method to Download Firmware Image to EZ-USB Internal RAM and EEPROM

Refer to Method to Download Firmware Image to EZ-USB Internal RAM on page 54 and Method to Download Firmware Image to External I2C EEPROM and follow a similar procedure to download *bulksrc.hex* to the RAM and *bulkext.iic* to Large EEPROM using `USB Control Center`. The *bulkext.hex* and *bulkext.iic* files are located at `<Installed_directory>\<Version>\Firmware\Bulksrc`. After downloading, the firmware re-enumerates with the PC using its internal VID/PID 0x04B4/0x1004.

7.5.4 Binding Cypress USB Driver for the Downloaded Firmware Image

The **Bulksrc** firmware uses vendor-class (0xFF) with VID/PID 0x04B4/1004. This example should bind with the Cypress-generic USB driver, *CyUSB3.sys*, and driver information file, *CyUSB3.inf* which contains the relevant VID/PID of this example. Follow the procedure outlined in Binding Cypress USB Driver for the Downloaded Firmware Image to manually bind the driver using the Windows Hardware Wizard. If you have performed the binding process for any one of the previous firmware examples, you can skip it for this example.

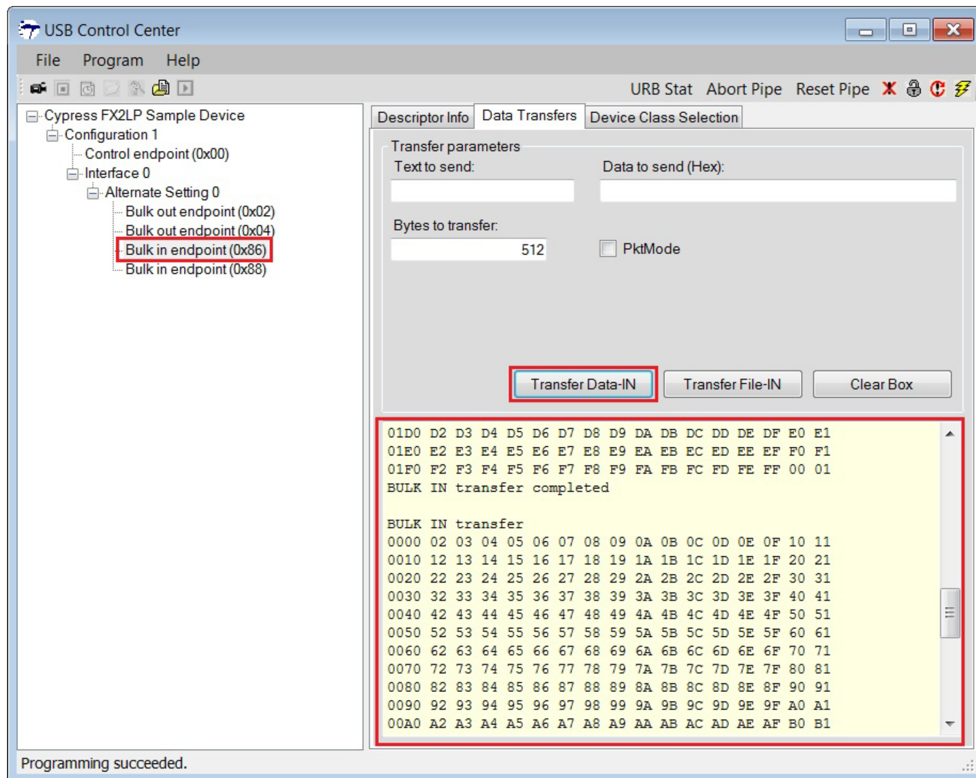
7.5.5 Testing Bulksrc Firmware Functionality

The Bulksrc firmware functionality can be tested using the USB Control Center utility as follows.

Open the USB Control Center application (*CyControl.exe*) from the location <Installed_directory>\<version>\Windows Applications\Application Source files\c_sharp\controlcenter\bin\Release. Observe that the EZ-USB FX2LP is listed as “Cypress FX2LP Sample Device.”

You can perform any number of OUT transfers on bulk out endpoint (0x02). Select “Bulk out endpoint (0x02)” in the left pane of the USB Control Center and keep clicking the **Transfer Data-Out** button in the **Data Transfers** tab on the right. You can see the USB Control Center sending multiple packets to this bulk OUT endpoint. Similarly, select “Bulk In endpoint (0x86)” in the left pane of the USB Control Center and keep clicking the **Transfer Data-In** button in the **Data Transfers** tab on the right. You can see the USB Control Center receiving multiple packets (incremental data from 02) from this bulk IN endpoint, as shown in Figure 7-10.

Figure 7-10. Bulk IN Data Transfer on EP6 Endpoint



The data transferred on EP4 is exactly looped back to EP8. Internally, the loopback is performed through a temporary buffer (myBuffer [512]).

Note: For EZ-USB FX1, the previous steps can be repeated with a data transfer length of 64 bytes instead of 512 bytes.

7.6 EP_Interrupts Example

7.6.1 Description

The **EP_interrupts** example works in a similar manner as bulkloop on EZ-USB FX2LP.The major differences include the addition of 64-byte EP1 as a bulk OUT/IN endpoint to the existing four endpoints: EP2, EP4, EP6, and EP8.The endpoints are re-armed using their respective ISRs. Following are the interrupts for these endpoints which are used to schedule the data transfers.

- EP1: 64-byte bulk OUT/IN – ISR_Ep1in() and ISR_Ep1out()
- EP2: 512-byte bulk OUT – ISR_Ep2inout()
- EP4: 512-byte bulk IN – ISR_Ep4inout()
- EP6: 512-byte bulk OUT – ISR_Ep6inout()
- EP8: 512-byte bulk OUT – ISR_Ep8inout()

7.6.2 Building EP_Interrupts Firmware Example Code for EZ-USB RAM and EEPROM

Click the **Build Target** button at the top right corner of the IDE. The “Total Code Bytes” of the **EP_Interrupts** firmware example is less than the 4-KB code limit of the Keil µVision2 IDE provided with the kit. The output of the **Build Target** is the *EP_Interrupts.hex* and *EP_Interrupts.iic* files

7.6.3 Method to Program EP_Interrupts Firmware Image to EZ-USB Internal RAM and EEPROM

Refer to section Method to Download Firmware Image to EZ-USB Internal RAM on page 54 and Method to Download Firmware Image to External I2C EEPROM on page 54 and follow similar procedure to download *EP_Interrupts.hex* to RAM and *EP_Interrupts.iic* to 64-KB EEPROM using the USB Control Center. The *EP_Interrupts.hex* and *EP_Interrupts.iic* files are located at <Installed_directory>\<Version>\Firmware\EP_Interrupts. After downloading, the firmware re-enumerates with the PC using its internal VID/PID 0x04B4/0x1004.

7.6.4 Binding Cypress USB Driver for the Downloaded Firmware Image

The **EP_Interrupts** firmware uses vendor class (0xFF) with VID/PID 0x04B4/1004.This example should bind with Cypress generic USB driver *cyUSB3.sys* and driver information file cyUSB3.inf, which contains the relevant VID/PID of this example. Follow the procedure outlined to manually bind the driver using the Windows Hardware Wizard. If you have performed the binding process for any one of the previous firmware examples, you can skip it for this example.

7.6.5 Testing EP_Interrupts Firmware Functionality

The example firmware should be tested in a manner similar to the **bulkloop** example.The bulk data transfers on EP1 are tested with a length of 64 bytes and 512 bytes for EP2, EP4, EP6, and EP8. The process is similar to the one outlined in Testing the Bulkloop Firmware Functionality on page 66.

7.7 iMemtest Firmware Example

This example does a data integrity check by writing and reading back the data on different memories inside the EZ-USB device such as GPIF waveform memory (0xE400) and endpoint buffer memories (0xE740, 0xF000, and so on), and avoids the range where the firmware is located. If the written data and read data match then “GOOD” is displayed on the 7-segment display-U9 and if there are errors at any specific memory location, the corresponding location is displayed. The example is compiled using the Keil IDE similar to previous examples and corresponding images for RAM (imemtest.hex) and EEPROM (imemtest.iic) can be generated. Both the images are located at **<Installed_directory>\<Version>\Firmware\iMemtest**. After downloading the images for RAM (*iMemtest.hex*) or EEPROM (*iMemtest.iic*), using the process outlined in Method to Download Firmware Image to EZ-USB Internal RAM on page 54, observe the 7-segment display for either a “g-o-o-d” string or the exact location of memory write/read failure.

7.8 LEDcycle Firmware Example

This example is used to test the connectivity between the EZ-USB IC and general-purpose LEDs D2–D5. Ensure that all four jumpers on JP3 are shorted to connect the LEDs before downloading the example. The example is compiled using the Keil IDE, similar to previous examples and corresponding images for RAM (*LEDcycle.hex*) and EEPROM (*LEDcycle.iic*) can be generated. Both the images are located at **<Installed_directory>\<Version>\Firmware\LEDCycle**. After downloading the images using the process outlined in the sections Method to Download Firmware Image to EZ-USB Internal RAM on page 54 and Method to Download Firmware Image to External I2C EEPROM on page 54 observe that LEDs D2–D5 are turned ON and OFF in a periodic manner.

7.9 Dev_IO Firmware Example

This example is used to test the connectivity of the 7-segment display and the push-button switches (F2, F3) on the FX2LP DVK. The 7-segment display (U9) and push buttons are connected to Philips PCF8574 I/O expanders (U8 and U10). The example is compiled using the Keil IDE similar to previous examples and corresponding images for RAM (*Dev_IO.hex*) and EEPROM (*Dev_IO.iic*) can be generated. Both the images are located at `<Installed_directory>\<Version>\Firmware\dev_io`. After downloading the images using the process outlined in the sections Method to Download Firmware Image to EZ-USB Internal RAM on page 54 and Method to Download Firmware Image to External I2C EEPROM on page 54, press the F2 push button and observe the decrement values in the range 0xF–0x0. Similarly, pressing F3 increments the values in the range 0x0–0xF, starting from the current value. Observe the 7-segment display of the values for each button press.

7.10 extr_intr Firmware Example

This example is used to demonstrate the use of external interrupts INT0, INT1, INT4, INT5, and INT6. The relevant ISRs for these external interrupts are provided in **isr.c**. Table 7-2 lists the registers and associated pins for the interrupts.

Table 7-2. External Interrupts and Register Definitions in EZ-USB Device

Interrupt	Interrupt Enable	Interrupt Pin	Priority Control	Natural Priority	Interrupt Request Flag	Interrupt Type	Interrupt type Controlling Bit
INT0	IE.0	PA.0	IP.0	1	TCON.1	Level or edge sensitive, active low	[TCON.0]
INT1	IE.2	PA.1	IP.2	3	TCON.3	Level or edge sensitive, active low	[TCON.2]
INT4	EIE.2	See note 1	EIP.2	10	EXIF.4	Edge sensitive, active high	--
INT5	EIE.4	See note 1	EIP.3	11	EXIF.5	Edge sensitive, active low	--
INT6	EIE.6	PE.5	EIP.4	12	EICON.3	Edge sensitive, active high	--

Refer to the section 4, “Interrupts,” in the EZ-USB Technical Reference Manual.

Notes:

1. The INT4 and INT5 have dedicated pins only in the 100 and 128 package. The pin for INT4 is shared between the GPIF, FIFO, and INT4 interrupts; setting INTSETUP.1 to “0” enables the INT4 operation. The default ***USBJmpTb.a51*** has an auto-vectoring option for INT4. To disable it, the following lines are commented out in ***USBJmpTb.a51***:

- CSEG AT 53H
- USB_Int4AutoVector equ $ + 2
- ljmp USB_Jump_Table

2. IE, EIE, IP, EIP, TCON, EXIF, and EICON are all SFRs. For a description of them, refer to the EZ-USB Technical Reference Manual.
3. Active low interrupts are falling edge triggered, and active high interrupts are rising edge triggered. In the example, the following register configurations are done in *extr_int.c* to set up the interrupts:

```
//INT0 and INT1
PORTACFG = 0x03; // PA0 and PA1 are pins for INT0 and INT1 respectively.
TCON |= 0x05; // INT0 and INT1 are configured as Edge triggered
interrupts.
//INT4
INTSETUP &= ~0x02; // If INTSETUP.1=0, then INT4 is supplied by the pin.
Else, the
// interrupt is supplied internally by FIFO/GPIF sources.
//INT5 is a dedicated pin, available in the 100 and 128 pin packages.
//INT6
PORTECFG = 0x20; // PE5 is INT6
OEE &= ~0x20;
//Enable External Interrupts
```

```
EIE |= 0x1C; // Enable External Interrupts 4, 5 and 6
IE |= 0x05; // Enable External Interrupts 0 and 1
//Clear Flags
EXIF &= 0xBF; // Clear INT4 EXIF.6 Flag
EXIF &= 0x7F; // Clear INT5 EXIF.7 Flag
EICON &= 0xF7; // Clear INT6 EICON.3 Flag
EA = 1; // Enable Global Interrupt
```

The ISRs for each of these external interrupts are defined in *isr.c*. These routines clear the interrupt and toggle the relevant port pin and any one of the LEDs from D2 to D5.

```
void ISR_EXTR4(void) interrupt 10
{
EXIF &= 0xBF; // Clear INT4 EXIF.6 Flag
IOC ^= 0x10; // Toggle pin 4 of PortC
}
```

The example is compiled using the Keil IDE, similar to previous examples, and corresponding images for RAM (extr_intr.hex) and EEPROM (*extr_intr.iic*) can be generated. Both the images are located at `<Installed_directory>\<Version>\Firmware\extr_intr`. Download the images using the process outlined in Method to Download Firmware Image to EZ-USB Internal RAM on page 54 and Method to Download Firmware Image to External I2C EEPROM on page 54.

7.10.1 Testing the Example

The code implements a function generator that can generate a square wave of a known frequency (use a low frequency, for example, a 100-Hz signal to view LED toggling). When the respective interrupts are triggered, the LED toggles. When an INT0 interrupt occurs, PC.0 and D2 are toggled. Similarly, on INT1/ INT4/ INT5/ INT6, PC.1 and D3/ PC.4 and D4/ PC.5 and D5/ PC.6 are toggled. The port C pin toggling can be checked by connecting those pins to a DSO. Table 7-3 shows the port pins corresponding to the external interrupts.

Table 7-3. Port Pins and External Interrupt Mapping

Pin Name	Port/Jumper Name on CY3684/CY3674
Port C	P3
LEDs	JP3
INT0	P2.19
INT1	P2.18
INT4	P6.5
INT5	P6.4
INT6	PE.5

7.11 Vend_ax Example

This example demonstrates the use of different vendor commands. Vendor commands are used to accomplish unique tasks, such as EZ-USB reset, RAM download, setting a different frequency for the FX2LP I^2C interface, communicate with an external SRAM, and so on. The vendor commands are defined in the *vend_ax.c* source file of the example. Open the project by clicking on ***vend_ax.uv2***, located at **<Installed_directory>\<Version>\Firmware\vend_ax** and observe the

vendor commands implemented in the C routine - DR_VendorCmnd (void). Following are the vendor commands defined in the ***vend_ax.c*** file:

Table 7-4. Vendor Command Definitions in vend_ax Example

S.No	Vendor Command/Macro Definition	Function
1	0xA2/VR_EEPROM	Downloads data to a small EEPROM
2	0xA3/ VR_RAM	Downloads data to internal or external RAM
3	0xA6/ VR_GET_CHIP_REV	Retrieves the current revision of EZ-USB(FX1/FX2LP)/MOBL-USB FX2LP18 IC
4	0xA8/VR_RENUM	The EZ-USB device disconnects and reconnects.
5	0xA9/VR_DB_FX	Selects double-byte addressed large EEPROM U5 and the contents can be uploaded or downloaded to EEPROM
6	0xAA/VR_I2C_100	Sets the I2C interface to 100 kHz
7	0xAB/VR_I2C_400	Sets the I2C interface to 400 kHz

The example is compiled using the Keil IDE, similar to previous examples, and corresponding images for RAM (*vend_ax.hex*) and EEPROM (*vend_ax.iic*) can be generated. Both the images are located at `<Installed_directory>\<Version>\Firmware\vend_ax`. Using the USB Control Center the images can be downloaded as outlined in Method to Download Firmware Image to EZ-USB Internal RAM on page 54 and Method to Download Firmware Image to External I2C EEPROM on page 54.

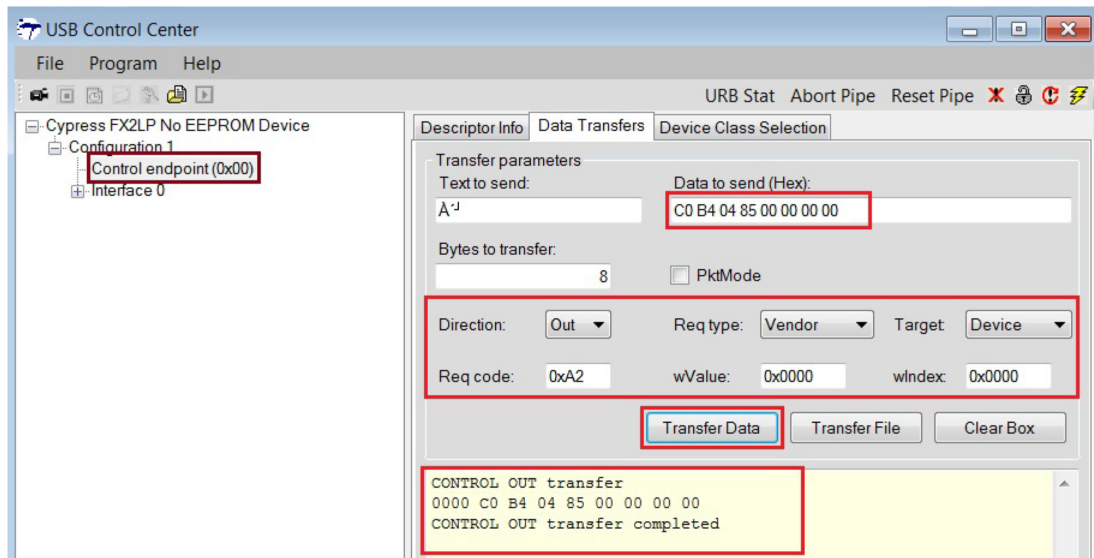
7.11.1 Testing the vend_ax Example

1. **0xA2 command – Read/write to EEPROM**
   As noted in Table 7-4, this command is used to read and write contents to a small EEPROM. To test this command on FX2LP DVK, change the switch SW2 (EEPROM ENABLE) to EEPROM position and SW1 (EEPROM SELECT) in SMALL EEPROM position. Note that these switch settings need to be changed only after downloading the firmware hex file (*vend_ax.hex*).

   a. Test using USB Control Center

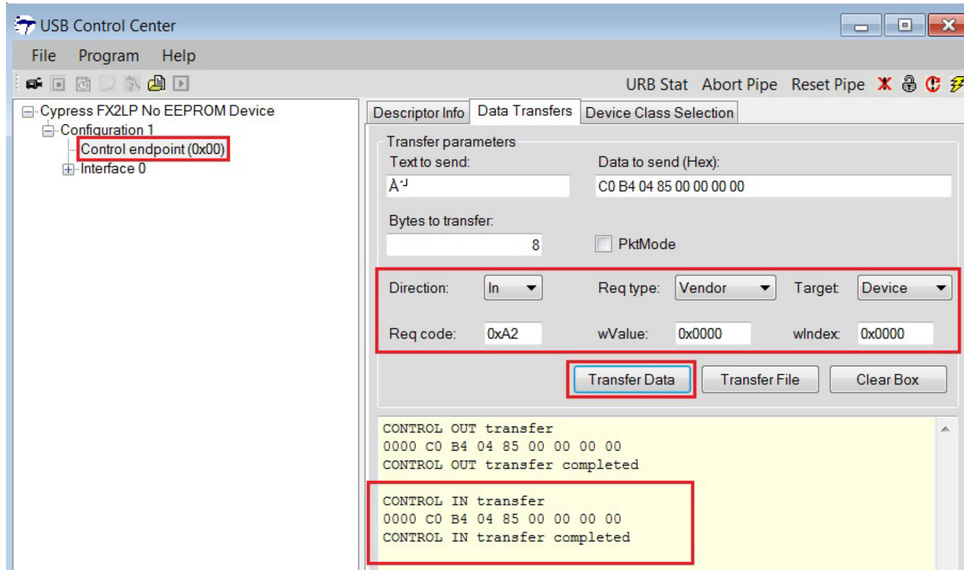
   To write the contents to a small EEPROM, select **Direction** = Out, **Req Type** = Vendor, **Target** = Device, **Bytes to Transfer** = 8 (data to read), and **Req Code** = 0xA2, and enter data to send as **C0 B4 04 85 00 00 00 00** in the **Data to send** box. Click on the **Transfer Data** button and observe the EEPROM getting programmed. Figure 7-11 summarizes the entire operation.

Figure 7-11. A2 Vendor Command Write Operation using USB Control Center



To read the contents of the small EEPROM, select **Direction** = In, **Req Type** = Vendor, **Target** = Device, **Bytes to Transfer** = 8 (data to read), and **Req Code** = 0xA2 for reading the data on control endpoint. Click the **Transfer Data** button. Figure 7-12 summarizes the entire operation.

Figure 7-12. A2 Vendor Command Read Operation using USB Control Center



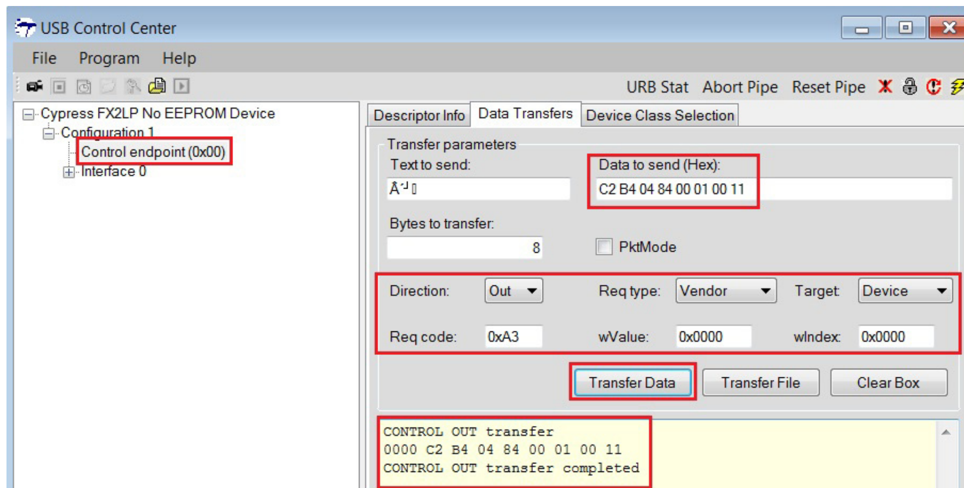
2. **0xA3 command-Download data to RAM**

   This command is used to download data to either the EZ-USB internal (0x0000–0x3FFFF) RAM or the external RAM memory.

   a. Test using USB Control Center

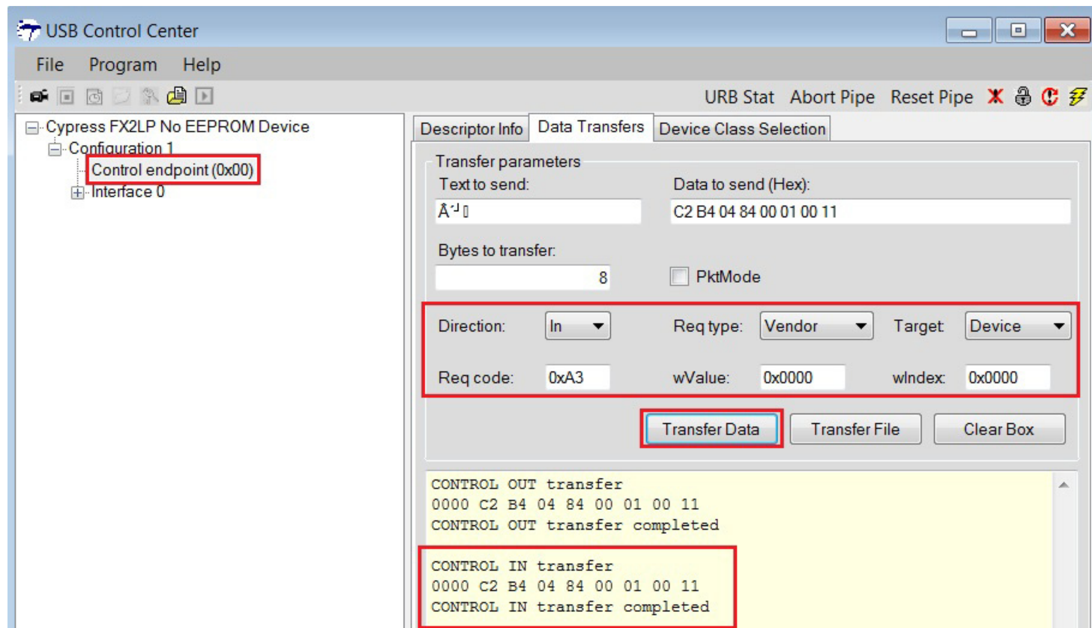
   To write the contents to RAM, select Direction = Out, Req Type = Vendor, Target = Device, Bytes to Transfer = 8 (data to read), and Req Code = 0xA3, and enter data to send as **C2 B4 04 84 00 01 00 11** in the **Data to send (Hex)** box. Click the **Transfer Data** button and observe the RAM getting programmed. Figure 7-13 summarizes the entire operation.

Figure 7-13. A3 Vendor Command Read Operation using USB Control Center



To read the contents from RAM, select Direction = In, Req Type = Vendor, Target = Device, Bytes to Transfer = 8 (data to read), and Req Code = 0xA3. Click the **Transfer Data** button and observe that the RAM written previously matches the read data. Figure 7-14 summarizes the entire operation.

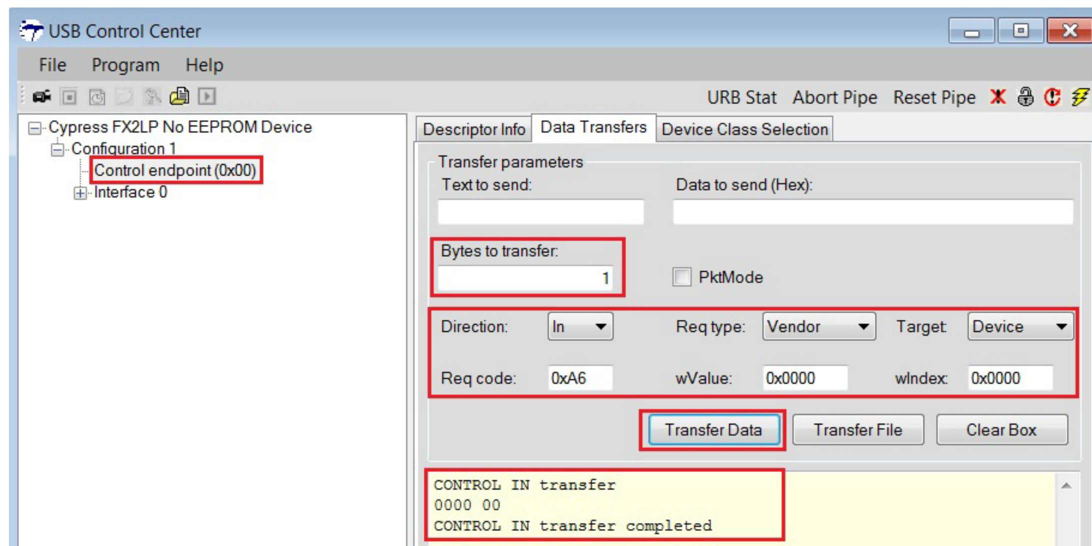
Figure 7-14. A3 Vendor Command Read Operation using USB Control Center



3. **0xA6 command – Get chip revision**

   To retrieve the current revision of the EZ-USB(FX1/FX2LP) or MOBL-USB(FX2LP18) device, this command is used. Figure 7-15 summarizes the entire operation using the USB Control Center.

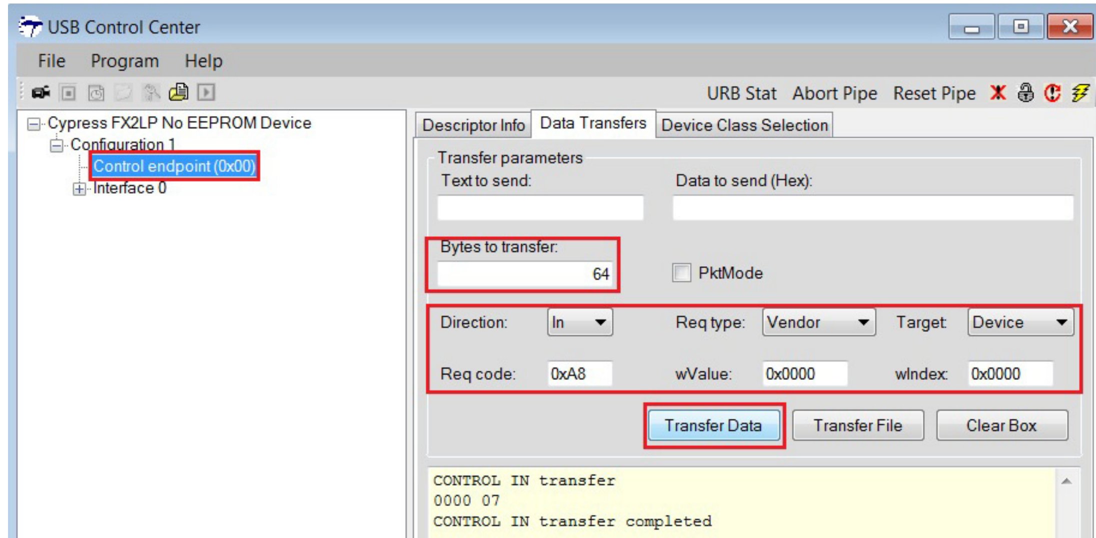
Figure 7-15. A6 Vendor Command using USB Control Center



4. **0xA8 command – EZ-USB ReNumeration**

   This command is used to disconnect and re-connect the EZ-USB IC using the CPUCS register. The EZ-USB re-enumerates. Observe the Cypress device disappearing from the USB Control Center window and re-appearing in the same window. Figure 7-16 summarizes the command trigger using the USB Center.

Figure 7-16. A8 Vendor Command Operation using USB Control Center



5. **0xA9 command – Read/Write Large EEPROM**

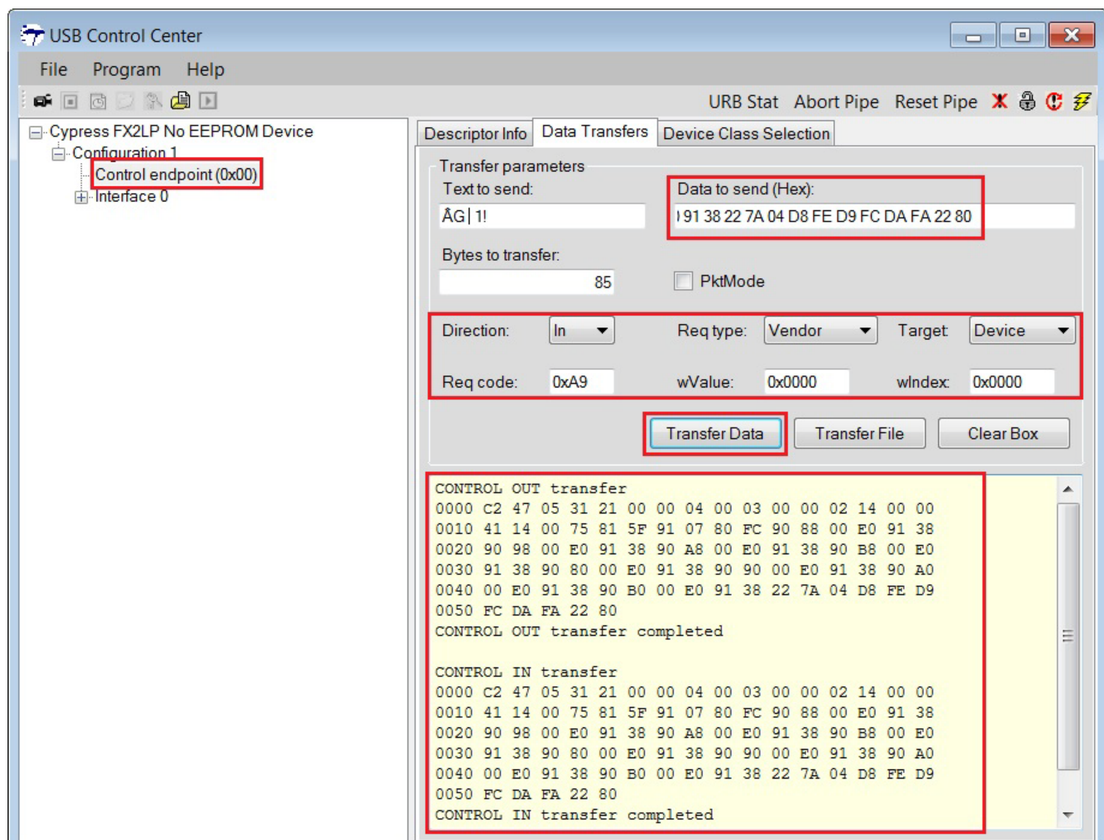
   To test this command on FX2LP DVK, change the switch SW2 (EEPROM ENABLE) to EEPROM position and SW1 (EEPROM SELECT) in LARGE EEPROM position. Note that these switch settings need to be changed only after downloading the firmware hex file (*vend_ax.hex*).

   To read/write the contents of Large EEPROM-U5, select Direction = In/OUT, Req Type = Vendor, Target = Device. **Bytes to Transfer** automatically gets updated if there is pre-defined data. In the Figure 7-17, the data of the *LEDCycle.iic* file is programmed in the OUT direction and read back in the IN direction bytes (data to read). The LEDCycle.iic file contents are as follows:

   C2 47 05 31 21 00 00 04 00 03 00 00 02 14 00 00 41 14 00 75 81 5F 91 07 80 FC 90 88 00
   E0 91 38 90 98 00 E0 91 38 90 A8 00 E0 91 38 90 B8 00 E0 91 38 90 80 00 E0 91 38 90
   90 00 E0 91 38 90 A0 00 E0 91 38 90 B0 00 E0 91 38 22 7A 04 D8 FE D9 FC DA FA 22 80

   Figure 7-17 summarizes the entire operation. Press the RESET button after programming and observe LED D2-D5 glowing in a periodic manner.

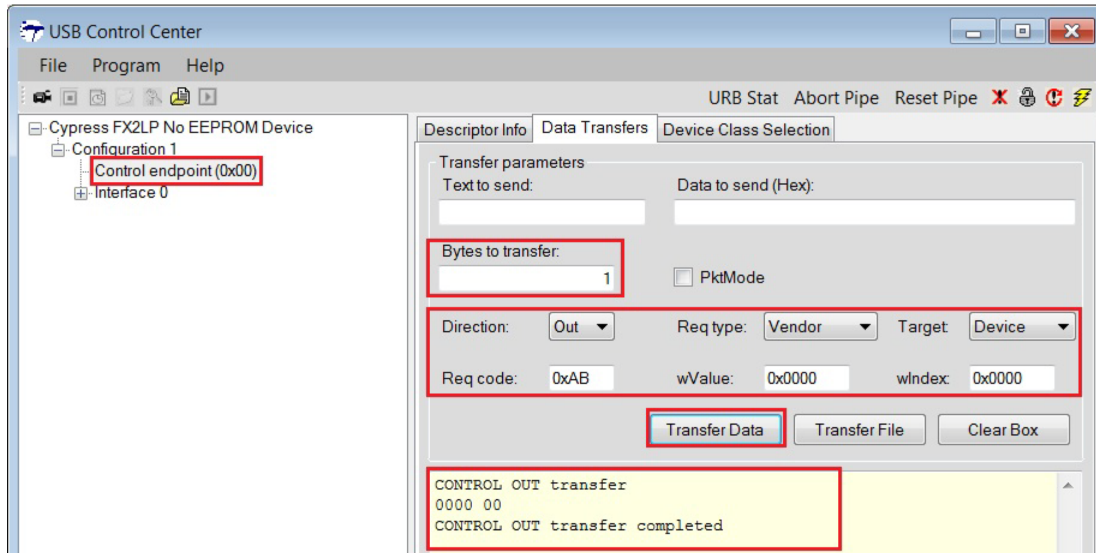
Figure 7-17. A9 Vendor Command Operation using USB Control Center



6. 0xAA/0xAB – Setting I^{2}C interface frequency

Using this command, the I^{2}C interface frequency can be set to 100 kHz or 400 kHz. 0xAA command sets the I^{2}C interface frequency to 100 KHz and 0xAB command sets the I^{2}C interface frequency to 400 KHz. Figure 7-18 summarizes the command trigger using the USB Control Center.

Figure 7-18. AA/AB Vendor Command Operation using USB Control Center



8. Resources

8.1 Hardware Resources

The CY3674/CY3684 development kit has several hardware resources that guide you in designing your own custom board. The documents in the hardware directory of the DVK kit software are:

- **CY3674_PCBA_BOM.xls/CY3684_PCBA_BOM.xls:** This document lists all the vendor hardware components used in designing both the development boards. The components used in both the development boards are identical. The only change between them is the component U1 replacement. EZ-USB FX1 128-pin package (CY7C64713-128AXC) is pin-to-pin compatible with EZ-USB FX2LP IC(CY7C68013A-128AXC) and replacing the IC is the only change required.
- **CY3684_Schematic.dsn/CY3674_Schematic.dsn:** These documents show the schematic design of the EZ-USB development board. The CY3684 and CY3674 schematics are identical. The only change is the replacement of EZ-USB FX2LP IC with EZ-USB FX1 part.
- **CY3674_Gerber.zip/CY3674_Gerber.zip:** This file can be used to understand the via, trace lengths, electrical connections, and so on, of the EZ_USB development board. Both .zip files contain identical source files since the development board is identical for both the kits.
- **CY3674_Board_Layout.pdf/CY3684_Board_Layout.pdf:** This is a non-editable layout file for EZ-USB development boards.
- **CY3674_Board_Layout.brd/CY3684_Board_Layout.brd:** This is an editable layout file for EZ-USB development boards. The file can be viewed using the Allegro PCB software.
- **Proto Board:** This directory contains the daughter card design files. The board is stacked on top of the EZ-USB development board and contains prototype area to validate the interface between EZ-USB GPIF and external devices like SRAM, Sensors, etc.
- **PAL Code:** This directory contains logic source inside the GAL22LV10C device on the EZ-USB development board. This PLD enables access to external SRAM memory. The relevant source code is provided in the Appendix.

8.2 Reference Designs

8.2.1 CY4611B - USB 2.0 to ATA Reference Design

You can test a variety of storage devices using the CY4615 DVK board by changing only the EEPROM configuration (.iic) files, but storage device related features cannot be updated. The CY4611B reference design kit can be used to add or update features. The board that comes along with CY4611B is based on the EZ-USB FX2LP™ chip, a general-purpose USB 2.0 high-speed device. After programming the ATA/ATAPI command processing firmware and the configuration files (.iic) combined, the board emulates AT2LP (similar to CY4615B DVK board). Here, you can modify the firmware by adding new features or modifying the existing firmware logic. The reference design kit contains documents related to hardware, firmware, and application software useful while working with the board available in this kit.

8.2.2 CY3686 NX2LP-FLEX USB 2.0-to-NAND Reference Design Kit

The CY3686 EZ-USB NX2LP-Flex™ USB 2.0 Development Kit is designed to accelerate development of a NAND flash-based USB 2.0 application featuring the USB 2.0 NAND controller (CY7C68033 and CY7C68034). Design a feature-rich thumbdrive with fingerprint sensor or GPS or add NAND storage to your DVB card. NX2LP-Flex eliminates the need for EEPROMs in your firmware based designs.

8.3 Application Notes

- AN65209 - Getting Started with FX2LP

  This application note presents the features and resources available to speed up the EZ-USB® FX2LP™-based design from concept to production. This document serves as a starting point for the new user to get familiar with FX2LP. It also gives an overview of the design resources available.

- AN15456 - Guide to Successful EZ-USB® FX2LP™ and EZ-USB FX1™ Hardware Design and Debug

  This application note outlines a process that isolates many of the most likely causes of EZ-USB® FX2LP™ and EZ-USB FX1™ hardware issues. It also facilitates the process of catching potential problems before building a board and assists in the debugging when getting a board up and running.

- AN4053 - Streaming Data Through Isochronous/Bulk Endpoints on EZ-USB® FX2™ and EZ-USB FX2LP™

  This application note provides brief background information on what is involved while designing a streaming application using the EZ-USB FX2 or the EZ-USB FX2LP part. It provides information on streaming data through bulk endpoints, isochronous endpoints, and high-bandwidth isochronous endpoints along with pitfalls to consider and avoid while using the FX2/FX2LP for designing high-bandwidth applications.

- AN58069 - Implementing an 8-Bit Parallel MPEG2-TS Interface Using Slave FIFO Mode in FX2LP

  This application note explains how to implement an 8-bit parallel MPEG2-TS interface using the Slave FIFO mode. The example code uses the EZ-USB FX2LP™ at the receiver end and a data generator as the source for the data stream. The hardware connections and example code are included along with this application note.

- AN58170 - Code/Memory Banking Using EZ-USB®

  The EZ-USB® FX1/FX2LP family of chips has an 8051 core. The 8051 core has a 16-bit address line and is only able to access 64 KB of memory. However, the firmware size sometimes exceeds 64 KB This application note describes methods of overcoming this 64 KB limitation and also demonstrates the implementation of one such method.

- AN57322 - Interfacing SRAM with FX2LP over GPIF

  This application note discusses how to connect Cypress SRAM CY7C1399B to FX2LP over the General Programmable Interface (GPIF). It describes how to create read and write waveforms using the GPIF Designer. This application note is also useful as a reference to connect FX2LP to other SRAMs.

- AN14558 - Implementing a SPI Interface with EZ-USB FX2LP™

  This application note details two approaches (bit-banging general purpose I/O (GPIO) pins, using the UART block) to implement SPI Master interface on FX2LP, including the example code. The application note also shows how to use a Microsoft Visual Studio application (USB Control Center) to communicate with SPI slave devices connected to FX2LP using USB Vendor Requests. An SPI EEPROM (25AA160B) is used as an example of a slave SPI device.

- AN1193 - Using Timer Interrupt in Cypress EZ-USB® FX2LP™ Based Applications

  This application note is aimed at helping EZ-USB® FX2LP™ based firmware developers use timer interrupts in their applications, by providing a framework based timer interrupt program written in C. The assumption is made that one has a general understanding of how interrupts work within the 8051 concept. When this program is run, you should be able to light the seven-segment LED on the FX2LP Development Board (CY3684) with a 0-9 count, and control the step rate (1s - 5s) using BULK OUT endpoint transfers from the EZ-USB Control Panel.

- AN63787 - EZ-USB FX2LP™ GPIF and Slave FIFO Configuration Examples using FX2LP Back-to-Back Setup

  AN63787 discusses how to configure the general programmable interface (GPIF) and slave FIFO's of EZ-USB FX2LP™ in both manual mode and auto mode, to implement an 8-bit asynchronous parallel interface. This Application Note is tested with two FX2LP development kits connected in a back-to-back setup. The first one acting in master mode and the second in slave mode.

- AN61244 - Firmware Optimization in EZ-USB® FX1/FX2LP

  The EZ-USB® FX1/FX2LP family of chips has an 8051 core and uses the standard 8051 instruction set. However, it has a few enhancements compared to the standard 8051. This application note describes firmware optimization methods in EZ-USB FX1/FX2LP. Some of these methods are common for any processor and some specific to the 8051 core of EZ-USB FX1/FX2LP.

- AN70983 - EZ-USB FX2LP™ Bulk Transfer Application in C# Using SuiteUSB C# Library (CyUSB.dll)

  AN70983 demonstrates how to build an application in Visual C# to send bulk data out and receive it back over a bulk endpoint of FX2LP, which is developed using Cypress SuiteUSB C# library (CyUSB.dll) for creating Windows applications using Microsoft Visual Studio. This document also explains associated firmware used in FX2LP to implement loopback transfers on bulk endpoints, and the application is tested with FX2LP Development kit.

- AN74505 - EZ-USB® FX2LP™ - Developing USB Application on MAC OS X using LIBUSB

  AN74505 describes a host application built on the MAC OS platform that uses libusb. The host application (Cocoa Application) communicates with the BULK IN and BULK OUT endpoints of FX2LP, using the interfaces provided by the APIs of libusb. This host application implements the transfer only with devices that pass the particular VID/PID(=0x04B4/0x1004) identification.

- AN58764 - Implementing a Virtual COM Port in FX2LP

  This application note explains how to implement a virtual COM port device using the standard Windows driver in FX2LP. This information helps in easy migration from UART to USB. The example code is provided with the application note, along with the required descriptors, class specific request handling, and the INF file required for enumeration.

- AN50963 - Firmware Download Methods to FX1/FX2LP

  This is an advanced document on firmware download techniques and readers are expected to be familiar with VC++ programming, USB 2.0 protocol, FX1/FX2LP architecture and device configuration options. Refer FX1/FX2LP datasheet and Technical Reference Manual available in Cypress website for more details on FX1/FX2LP product architecture and configuration details.

- AN45471 - Vendor Command Design Guide for the FX2LP

  Vendor commands are used to issue commands to a device, by which tasks unique to an application are accomplished. This application note demonstrates how you can quickly design USB vendor commands to perform specific features of products. In addition, using the Cypress CyConsole utility to issue vendor commands is also explained.
- AN58009 - Serial (UART) Port Debugging of FX1/FX2LP Firmware

  This application note describes the code needed in the FX2LP firmware for serial debugging. This code enables the developer to print debug messages and real time values of the required variables using a PC terminal emulation program or capture it in a file using the UART engine in FX2LP.
- AN42499 - Setting Up, Using, and Troubleshooting the Keil™ Debugger Environment

  This application note is a step-by-step beginner's guide to using the Keil Debugger. This guide covers the serial cable connection from PC to SIO-1/0, the monitor code download, and required project settings. Additionally, the guidelines to start and stop a debug session, set breakpoints, step through code, and solve potential problems are considered.

8.4 Technical Reference Manual

EZ-USB® Technical Reference Manual

EZ-USB technical reference manual describes all the registers of FX2LP along with the example codes to configure them.

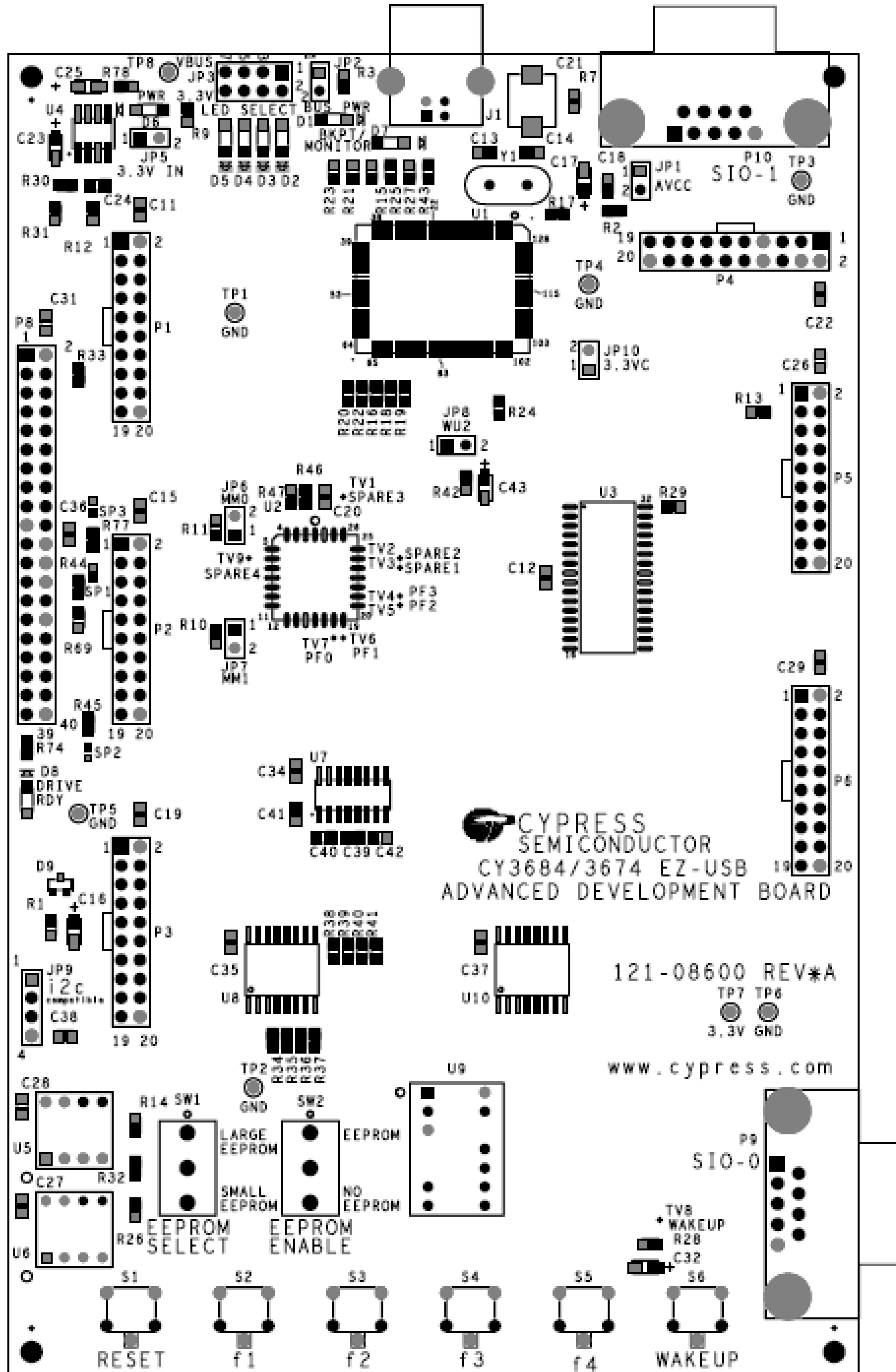
A. Appendix

A.1 U2 (GAL) code (file is 'FX2LP.ABL')

```
MODULE fx2lp
" Swapped dipswitch settings 00 and 10 on 4-3-98 to allow the all-switchon
default
x,c,z = .X.,.C.,.Z.;
"Inputs
A12,A13,A14,A15 pin 11,12,13,16;
A11 pin 4;
nRD,nPSEN,CLKOUT pin 6,5,2;
mm1,mm0 pin 9,7;
"Outputs
EA,nRAMOE,nRAMCE pin 21,25,27;
PF0,PF1,PF2,PF3 pin 17,18,19,20 istype 'reg_sr';
modesw = [mm1,mm0]; " two dipswitches
addr = [A15,A14,A13,A12,A11,nRD];" high nibble of the address bus + RD
equations
" The 3681 board turns PF0 on at 0x80xx reads and off at 0x81xx reads.
" This board turns PF0 on at 0x8xxx reads and off at 0x88xx reads.
PF0.S = (addr == ^b100000);
PF0.R = (addr == ^b100010);
PF0.CLK = CLKOUT;
PF1.S = (addr == ^b100100);
PF1.R = (addr == ^b100110);
PF1.CLK = CLKOUT;
PF2.S = (addr == ^b101000);
PF2.R = (addr == ^b101010);
PF2.CLK = CLKOUT;
PF3.S = (addr == ^b101100);
PF3.R = (addr == ^b101110);
PF3.CLK = CLKOUT;
WHEN (modesw == 00) THEN" No external memory
{
nRAMCE = 1;
nRAMOE= 1;
EA = 0;
}
ELSE WHEN(modesw == 01) THEN" Ext P&D mem at 8000 (can add mem to 0-8K)
{
!nRAMCE= A15;
!nRAMOE= !nRD # !nPSEN;" Combine program & data memory
EA = 0;
}
```

```
ELSE WHEN(modesw == 11) THEN" Ext P&D mem at 0000 and 8000
{
!nRAMCE = 1;
!nRAMOE= !nRD # !nPSEN;
EA = 0;
}
ELSE WHEN(modesw == 10) THEN" All program mem external
{
!nRAMCE = 1;
!nRAMOE =!nRD # !nPSEN;
EA = 1;
}
test_vectors
([mm1,mm0,A15,nRD,nPSEN] -> [nRAMCE, nRAMOE, EA])
[ 0 , 0 , x , x , x ] -> [ 1 , 1 , 0];" 10: all mem selects and
strobes OFF
[ 0 , 1 , 0 , 1 , 1 ] -> [ 1 , 1 , 0];" 01: top of mem for rd or
psen
[ 0 , 1 , 1 , 1 , 0 ] -> [ 0 , 0 , 0];" PSEN only
[ 0 , 1 , 1 , 0 , 1 ] -> [ 0 , 0 , 0];" RD only
[ 0 , 1 , 1 , 1 , 1 ] -> [ 0 , 1 , 0];" Neither RD or PSEN
[ 1 , 1 , 0 , 1 , 0 ] -> [ 0 , 0 , 0];" 11: top and bot mem for rd
or psen
[ 1 , 1 , 0 , 0 , 1 ] -> [ 0 , 0 , 0];
[ 1 , 1 , 0 , 1 , 1 ] -> [ 0 , 1 , 0];
[ 1 , 1 , 1 , 1 , 0 ] -> [ 1 , 0 , 0];" PSEN
[ 1 , 1 , 1 , 0 , 1 ] -> [ 1 , 0 , 0];" RD
[ 1 , 1 , 1 , 1 , 1 ] -> [ 1 , 1 , 0];" neither
[ 1 , 0 , 1 , 1 , 0 ] -> [ 1 , 0 , 1];" PSEN
[ 1 , 0 , 1 , 0 , 1 ] -> [ 1 , 0 , 1];" RD
[ 1 , 0 , 1 , 1 , 1 ] -> [ 1 , 1 , 1];" neither
test_vectors
([nRD,nPSEN] -> [nRAMOE])
[ 0 , 0 ] -> [ 0 ];
[ 0 , 1 ] -> [ 0 ];
[ 1 , 0 ] -> [ 0 ];
[ 1 , 1 ] -> [ 1 ];
test_vectors
(addr -> [PF0, PF1, PF2, PF3])
[1,0,0,0,0,0] -> [0, 0, 0, 0];
[1,0,0,0,1,0] -> [1, 0, 0, 0];
END
```

A.2 Board Layout



A.3 Schematic

The only difference between the CY3674 and CY3684 kits is the Cypress EZ-USB part. While the development board of the CY3674 kit includes FX1 (128-pin package), the CY3684 kit has FX2LP. All other components and layout are similar to the CY3684 kit board.

A.4 Frequently Asked Questions

Q1: What is the first step, after viewing the printed material from the box?

A1: Make sure the hardware works well enough to run the tutorial. After software installation, plug in a development board; go through the DVK tutorial. The tutorial is located in the *EZ-USB DVK User Guide*, which is found in the Start menu under **Cypress** > **USB** > **Help**. The tutorial is short and worthwhile.

Q2: What is the first example to try?

A2: While following the tutorial, you will read the Device ID from the development board, and then load the dev_io example. This turns on the LED so you know that firmware has been loaded and runs correctly.

Q3: Where do I find the soft copy of the *EZ-USB Getting Started*?

A3: See `<Installed_directory>\<Version>\Documentation\EZ-USB (R) DEVELOPMENT KIT USER GUIDE.pdf.`

Q4: Where do I find the soft copy of the *EZ-USB Technical Reference Manual* (TRM)?

A4: See `<Installed_directory>\<Version>\Documentation\EZ-USB(R) Technical Reference Manual.pdf`. This is the key reference to use. You can search for the material you are most interested in quickly.

Q5: Where is the EZ-USB data sheet?

A5: See `<Installed_directory>\<Version>\Documentation\EZ-USB FX1 Datasheet.pdf` for the CY3674 and `<Installed_directory>\<Version>\Documentation\EZ-USB(R) FX2LP Datasheet.pdf` for the CY3684.

Q6: Where is the EZ-USB Development Board schematic (pdf and Orcad files)?

A6: See `<Installed_directory>\<Version>\Hardware` for EZ-USB Kits .

Q7: Where can I find the errata?

A7: See `<Installed_directory>\<Version>\Documentation\SILICON ERRATA FOR EZ-USB(TM) FX1 PRODUCT FAMILY.pdf` for the CY3674 and `<Installed_directory>\<Version>\Documentation\ERRATA FOR THE EZUSB-FX2LP.pdf` for the CY3684.

Q8: How do I to generate "myapp" from (framework)?

A8: Create a (framework based) project folder by copying the ‘fw’ example folder `<Installed_directory>\<Version>\Target\Fw\LP` a new location (under ‘Examples’). Then rename the folder to the new project name. Remove the .hex file, and .Uv2 file. Rename periph.c to <NewPrj>.c, and then create a new uV2 project file. See FX2LP Code Development Using the EZ-USB Framework chapter on page 39 for more information.

Q9: How do I build an EEPROM image to burn my code?

A9: See the tutorial for information about generating EEPROMs.

Q10: Where can I get a summary of the registers?

A10: See the register summary in the TRM.

Q11: Are there any examples?

A11: Yes, see the examples and readme files in the Examples folder.

Q12: Please provide details about the environment setup.

A12: If you install into the default directory, `<Installed_directory>\<Version>` then you can build and debug examples with the Keil uV2 project files provided. The project files have hard-coded paths in them; installing to a different, non-default directory location breaks these project files. Also, there are build.bat files for the projects in the Example folders. To run the build.bat files from the command line, you need to set some paths and environment variables, which can be done by running the batch file `<Installed_directory>\<Version>\Bin\setenv.bat` before typing 'build'. Again, if the kit software or Keil tools are installed to a non-default location, you need to modify the setenv.bat file. The setenv.bat also has directions on how to create a Start menu option to open an MS-DOS window with the correct environment set up.

Q13: Which DB-9 do I plug my mon-51 cable into?

A13: Use SIO-1 by default. There are other versions of the monitor in `<Installed_directory>\<Version>\Target\\Monitor`. They can be loaded using the Control Panel. There are different versions that load to Internal or external RAM memory and use SIO-0 or SIO-1, as indicated by the name.

Revision History

Document Revision History

Document Title: CY3674/CY3684, EZ-USB® Development Kit User Guide Document Number: 001-66390				
Revision	**ECN#**	**Issue Date**	**Origin of Change**	**Description of Change**
**	3164528	02/07/2011	ROSM	Initial version of user guide.
*A	3363028	05/09/2011	NMMA	Updated EZ-USB Advanced Development Board (FX2LP DVK) chapter on page 17: Updated "Schematic Summary" on page 17: Updated description.
*B	3638279	06/06/2012	NMMA	The document has been updated with the OOB review comments.
*C	3660764	06/29/2012	NMMA	Minor text edits. Updated correct path "Start->All programs->Cypress->Cypress Suite USB 3.4.7-->CyConsole" in all instances across the document.
*D	4138153	09/27/2013	DBIR	Updated USB PC Host Utilities and SuiteUSB Applications chapter: Updated "SuiteUSB Applications": Removed the section "Bulkloop Applications".
*E	4287828	02/21/2014	NPRS	Updated in new template. Completing Sunset Review.
*F	4509941	09/22/2014	RSKV	Updated Introduction chapter on page 7: Updated "Development Kit" on page 7: Updated description. Updated "Kit Contents" on page 7: Updated "Software" on page 8: Updated description. Updated Figure 1-2. Removed "Required Tools Not Included". Removed "Other Suggested Tools". Added "Other Tools" on page 9. Updated Getting Started chapter on page 11: Added "Prepare the FX2LP Development Kit" on page 11. Updated "Kit Software Installation" on page 12: Updated description. Updated Figure 2-2. Added My First USB 2.0 Transfer using FX2LP chapter on page 13.

Document Revision History *(continued)*

Document Title: CY3674/CY3684, EZ-USB® Development Kit User Guide Document Number: 001-66390				
Revision	**ECN#**	**Issue Date**	**Origin of Change**	**Description of Change**
*F (cont.)	4509941	09/22/2014	RSKV	Updated EZ-USB Advanced Development Board (FX2LP DVK) chapter on page 17: Updated "Introduction" on page 17: Updated description. Updated "Schematic Summary" on page 17: Updated description. Updated "EEPROM Select and Enable Switches SW1 and SW2" on page 19: Updated description. Updated "Interface Connectors" on page 21: Updated description. Updated Table 4-3. Updated "Memory Maps" on page 26: Updated Figure 4-1. Updated "I2C Expanders" on page 27: Updated description. Updated "General-Purpose Indicators" on page 28: Updated description. Updated Development Kit Files chapter on page 31: Updated "Bin" on page 31: Updated description. Updated "Documentation" on page 31: Updated description. Updated "Drivers" on page 32: Updated description. Updated "Firmware" on page 32: Updated description. Updated "GPIF_Designer" on page 33: Updated description. Updated "Hardware" on page 34: Updated description. Updated "Target" on page 34: Updated description. Updated "Utilities" on page 35: Updated description. Updated Figure 5-2. Updated Figure 5-3. Updated "uV2_4k" on page 36: Updated description. Added "Windows Applications" on page 36.

Document Revision History *(continued)*

Document Title: CY3674/CY3684, EZ-USB® Development Kit User Guide Document Number: 001-66390				
Revision	**ECN#**	**Issue Date**	**Origin of Change**	**Description of Change**
*F (cont.)	4509941	09/22/2014	RSKV	Updated FX2LP Code Development Using the EZ-USB Framework chapter on page 39: Removed "Frameworks Overview". Added "Structure of an FX2LP Application" on page 39. Added "Build the Bulkloop Project" on page 43. Updated "Framework" on page 44: Updated description. Updated "Framework Functions" on page 45: Updated "Task Dispatcher Functions" on page 45: Updated description. Updated "Device Request Functions" on page 45: Updated description. Updated Figure 6-4. Updated "ISR Functions" on page 47: Updated description. Updated "EZ-USB Library" on page 48: Updated "Library Functions" on page 49: Updated description. Removed "Cypress USB Drivers for EZ-USB Kits" chapter. Removed "USB PC Host Utilities and SuiteUSB Applications" chapter. Updated Firmware Examples in Detail chapter on page 51: Updated "hid_kb Firmware Example" on page 51: Updated description. Updated Figure 7-1. Updated "Building Firmware Example Code for EZ-USB Internal RAM and External EEPROM." on page 53: Updated Figure 7-2. Updated "Method to Download Firmware Image to EZ-USB Internal RAM" on page 54: Updated description. Updated "Method to Download Firmware Image to External I2C EEPROM" on page 54: Updated description. Updated Figure 7-4. Updated "Testing the hid_kb Firmware Example Functionality" on page 55: Updated description. Updated "IBN Firmware Example" on page 56: Updated "Description" on page 56: Updated description. Updated "Binding Cypress USB Driver for the Downloaded Firmware Image" on page 59: Updated description. Updated Figure 7-5. Updated "Pingnak Firmware Example" on page 60: Updated "Description" on page 60: Updated description. Updated "Binding Cypress USB Driver for the Downloaded Firmware Image" on page 63: Updated description. Updated "Testing the pingnak Firmware Functionality" on page 63: Updated description.

Document Revision History *(continued)*

Document Title: CY3674/CY3684, EZ-USB® Development Kit User Guide Document Number: 001-66390				
Revision	**ECN#**	**Issue Date**	**Origin of Change**	**Description of Change**
*F (cont.)	4509941	09/22/2014	RSKV	Updated Firmware Examples in Detail chapter on page 51: Updated “Bulkloop Example” on page 63: Updated “Building Bulkloop Firmware Example Code for EZ-USB RAM and EEPROM” on page 65: Updated Figure 7-7. Updated “Method to Download Bulkloop Firmware Image to Internal RAM or EEPROM” on page 66: Updated description. Updated “Binding Cypress USB Driver for the Downloaded Firmware Image” on page 66: Updated description. Updated “Testing the Bulkloop Firmware Functionality” on page 66: Updated description. Updated Figure 7-8. Updated Figure 7-9. Updated “Bulksrc Firwmare Example” on page 68: Updated “Description” on page 68: Updated description. Updated “Method to Download Firmware Image to EZ-USB Internal RAM and EEPROM” on page 70: Updated description. Updated “Testing Bulksrc Firmware Functionality” on page 71: Updated description. Updated Figure 7-10. Updated “EP_Interrupts Example” on page 72: Updated “Binding Cypress USB Driver for the Downloaded Firmware Image” on page 72: Updated description. Updated “iMemtest Firmware Example” on page 73: Updated description. Updated “LEDcycle Firmware Example” on page 73: Updated description. Updated “Dev_IO Firmware Example” on page 73: Updated description. Updated “extr_intr Firmware Example” on page 74: Updated description. Updated “Testing the Example” on page 75: Updated description. Updated “Vend_ax Example” on page 75: Updated description. Updated “Testing the vend_ax Example” on page 77: Updated description. Updated Figure 7-11. Updated Figure 7-12. Updated Figure 7-13. Updated Figure 7-14. Updated Figure 7-15. Updated Figure 7-16. Updated Figure 7-17. Updated Figure 7-18. Removed “Debugging Using Keil Monitor Program”.

Document Revision History *(continued)*

Document Title: CY3674/CY3684, EZ-USB® Development Kit User Guide Document Number: 001-66390				
Revision	**ECN#**	**Issue Date**	**Origin of Change**	**Description of Change**
*F (cont.)	4509941	09/22/2014	RSKV	Updated Resources chapter on page 83: Updated “Reference Designs” on page 83: Removed “CY4651 v1.3 - Cypress and AuthenTec Reference Design for Biometric Security in External USB Hard Disk Drives”. Updated “CY3686 NX2LP-FLEX USB 2.0-to-NAND Reference Design Kit.” on page 84: Updated description. Updated “Application Notes” on page 84: Updated description. Added “Technical Reference Manual” on page 86.
Distribution: External **Posting**: None				