Please note that Cypress is an Infineon Technologies Company.
The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

Continuity of document content
The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

Continuity of ordering part numbers
Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.

AN45471

Create Your Own USB Vendor Commands Using FX2LP™

Author: Rich Peng

Associated Project: Yes

Associated Part Family: CY7C68013A/CY7C68014A/CY7C68015A/CY7C68016A

Related Resources: click here

To get the latest version of this application note, or the associated project file, please visit http://www.cypress.com/go/AN45471.

More code examples? We heard you.

To access a variety of FX2LP code examples, please visit our USB High-Speed Code Examples webpage.

This application note explains how to create a custom USB device using EZ-USB® FX2LP™ and communicate using vendor requests. The associated code example implements six different vendor commands which can be tested using the FX2LP Development Board (CY3684) and the Cypress USB Control Center application.

Contents

1 Introduction....1
2 Vendor Commands....2
2.1 0: Standard Request....3
2.2 01: Class Request....3
2.3 02: Vendor Request....3
3 Vendor Command Example Code....4
4 Test Drive....4
5 How It Is Done....10
5.1 fw.c....10
5.2 Custom_Request_Descriptors.a51....10
5.3 EEPROM.C....10
5.4 EZUSB.LIB....10
5.5 Custom_Requests.c....10
5.6 USBJmpTb.OBJ....11
6 Code Analysis....11
7 Debugging the Code....12
8 Customizing the USB Control Center....13
9 The Last Step....13
10 Resources and Additional Information....13
11 Summary....13
Document History....14
Worldwide Sales and Design Support....15
Products....15
PSoC® Solutions....15

1 Introduction

Standard USB products, such as keyboards, mice, and disk drives, follow a class of devices defined by the USB Implementers Forum (USB-IF). These device classes provide the standardization that has helped make USB successful.

However, the architects of USB realized that not all devices would fit into a standard class. In addition, sometimes there are advantages to adding functionality to a class. To handle these situations, the architects created the “vendor-specific” request type.

In the USB specification, a special code designates this type, and specific values for fields using these requests are *not* defined — each vendor is required to define them. FX2LP firmware source code is attached to this application note to demonstrate the use of USB vendor commands.

Using the new USB commands added in the attached example, you can send a hexadecimal number to the 7-segment readout, illuminate the four LEDs, read the onboard EEPROM/device RAM and write to the EEPROM. You could easily add this new set of commands (and others) to FX2LP code under development to help with debugging.

As another usage example, a data mover app might use BULK IN and OUT endpoints to stream data and custom commands to implement out-of-band control that does not interfere with data in the BULK endpoints. Custom commands might include “start sending”, “stop sending”, and “loop data”. This natural separation of data and control makes for clean and efficient designs.

To see how this works, let’s first inspect the 8 bytes defined by USB as a device request. USB uses this request to launch all USB operations in a device.

2 Vendor Commands

This section shows the format of a USB device request and describes the different USB requests based on the bmRequestType value.

Table 1. USB Device Request Packet Format

Byte	Field	Meaning
0	bmRequestType	Request type
1	bRequest	Actual request
2	wValue	Varies by request
4	wIndex	Varies by request
6	wLength	Number of data bytes

Table 1 shows the USB device request packet format. Each request consists of an 8-byte data packet. Requests always use the control endpoint EP0. This packet is the main USB dispatcher; it determines the request direction, the type, and the recipient of the request. Standard requests travel in SETUP packets, and they may optionally be followed by DATA packets if required by the request.

Table 2. Definition of the bmRequestType Bits

Bits	Field	Values
7	Direction	0: Host to Device (OUT) 1: Device to Host (IN)
6..5	Type	0: Standard 1: Class **2: Vendor** 3: Reserved
4..0	Recipient	0: Device 1: Interface 2: Endpoint 3: Other 4-31: Reserved

Table 2 shows the format for the first byte, bmRequestType. Bit 7 tells us the direction for the request. Bits 6:5 specify the request *type*.

2.1 0: Standard Request

Every USB device must respond to standard requests. The USB host issues standard requests when a USB device is plugged in and, optionally, during operation to select different configurations and interfaces. The “get acquainted” process at plug-in is called enumeration. The standard requests are defined in Chapter 9 of the USB Specification, which is found at (http://www.usb.org/developers/docs/usb_20_070113.zip). To obtain USB certification, a device must pass “Chapter 9” compliance tests provided by the USB-IF.

The host issues standard requests, called Get_Descriptor requests, to initialize the device and to query it about its capabilities and requirements. Finally, the host configures the device (Set_Configuration request), and the device is ready for operation.

2.2 01: Class Request

During enumeration, a device might return descriptor information that declares it as belonging to a standard class. This informs the host that the device complies with a secondary USB specification. To send requests to a device conforming to a USB class, the host sets the request type to “Class” (0x01) in the bmRequestType field. This means that the remaining fields in the device request have specific meanings defined by the class specification to which the device belongs. For example, to retrieve a report (data structure) from a HID-class peripheral, such as a mouse, the standard request fields are pre-defined by the HID spec as follows:

Table 3. Standard Request Packet for a HID-Class Peripheral

Byte	Field	Meaning
0	bmRequestType	1 **01** 00001
1	bRequest	GET_REPORT = 0x01
2	wValue	Report Type and ID
4	wIndex	Interface number
6	wLength	Report Length

After a standard request is directed to a USB class (the highlighted “01” in the bmRequestType field), the remaining fields are defined by that class’ specification. For example, for the HID “GET_REPORT” request, the wValueH byte is defined as a report type (01=Input, 02=Output, 03=Feature), and the wValueL byte is defined as an optional Report ID. Don’t worry about what these mean; the point is that a class request predefines the meanings of the remaining bytes in the standard request packet.

2.3 02: Vendor Request

Vendor requests allow you to define your own meanings for the bytes in the standard request packet. You have the power to make up your own USB class. When the host issues a standard request with the bmRequestType equal to 0/1**10**00000 (8 bits, most-significant bit can be 0 or 1), it knows that the remaining bytes mean what you defined. For this to work, both sides, host and device, must agree on the meaning of these bytes. Because you write both the FX2LP device code and the host program that accesses your custom device, you can ensure compatibility.

Vendor requests might be created in the following situations:

- You cannot find any predefined class to fit your application
- You want to add functionality to an existing class

This application note uses the Cypress CY3684 FX2LP™ Development Kit to define six vendor requests to talk to the FX2LP development board, using a Windows application. This information helps you to create your own vendor commands.

3 Vendor Command Example Code

The following materials are required to understand this application note and to exercise and modify the example code:

- CY3684 Development Kit. This kit contains the FX2LP Evaluation Board, which this application note uses as its custom device.
- The latest version of the free CY3684 EZ-USB® FX2LP folder, which is available at http://www.cypress.com/?rID=14321. Unzip this file and run the CY3684Setup.exe file. Select the "Typical" install and the setup program will create the installation directory at C:\Cypress\USB. This directory includes Windows drivers, the free Keil toolset (full-function but limited to 4-Kbyte code size), and other FX2LP goodies (such as hardware design files of the FX2LP DVK, documents to use FX2LP DVK and firmware examples). For this app note, we need only the Windows driver and the Keil tools. The app note code is under 4 Kbytes, so you can use the free Keil version to study and modify this code (within reason).
- The app. Download, unzip, and install the file associated with this app note (same link as the app note) as C:\Cypress\Custom_Requests. This installs the sub-folder "Source," which contains all of the source files.

This note uses a VC# program to test the custom device. You can download Cypress's free set of Visual Studio development tools at http://www.cypress.com/?rID=34870. This installation creates the directory C:\Cypress\Cypress Suite USB x.y.z, where xyz is the version number. This folder contains a Visual Studio project called "USB Control Center," which you can use to test our custom FX2LP device. Included are both the compiled .exe file and its source code so that you can run it or modify the source with a free copy of Microsoft Visual Studio Express 2008, 2010, or 2012. The executable is in the CyUSB.NET\bin subdirectory.

4 Test Drive

Before delving into the code, this section shows you how to test the custom device we will define. Follow these steps to see the custom device in action.

1. Prepare the FX2LP board jumpers according to Table 4.

Table 4. EZ-USB FX2LP Board Jumper Settings

JP	State	Purpose
6,7	OUT	Memory config. for development
2	IN	Power the board from the USB connector
1,5,10	IN	Local 3.3-V source
3	IN	All 4 jumpers IN — activate 4 LEDs, D2-D5
8	Either	Not used (for remote wakeup testing)

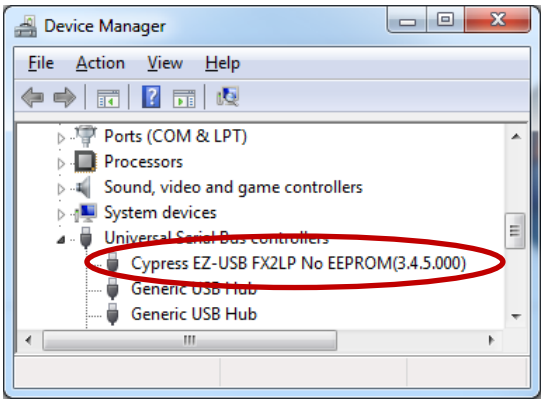
2. In the lower-left corner of the board, move the EEPROM ENABLE slide switch to the "NO EEPROM" (down) position. This allows the FX2LP chip to enumerate as a bare device, ready to download code into either onboard RAM or EEPROM. The other slide switch (EEPROM SELECT) can be in either position.
3. Plug the FX2LP board into a PC USB port. If this is the first time, you should see popup messages asking you to install a USB driver. Navigate to:

   C:\Cypress\USB
   \CY3684_EZ-USB_FX2LP_DVK\1.0
   \Drivers\cyusbfx1_fx2lp

Select the folder corresponding to your Windows OS.

You can confirm a successful install by viewing the Device Manager:

Figure 1. FX2LP Board Driver Is Installed



4. Launch the Cypress USB Control Panel at:

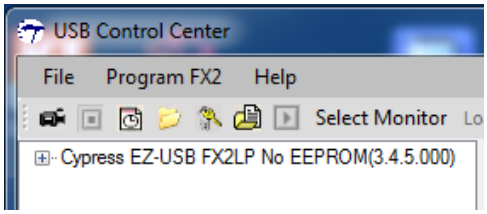
   C:\Cypress

   \Cypress Suite USB 3.4.7

   \CyUSB.NET\bin\CyControl.exe

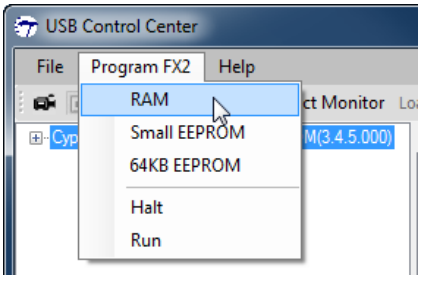
5. You should see the FX2LP board along with other connected USB devices listed in the left panel. To simplify this display so that it shows only the FX2LP board, click the “Device Class Selection” tab in the right panel and uncheck everything except the “Devices served by the CyUSB.sys driver (or a derivative)” item. The left panel should look like this:

Figure 2. USB Control Center Finds the FX2LP Board



6. Now we’re ready to load the FX2LP custom device app into the board. Click on the Cypress device entry to highlight it, then select Program FX2/RAM.

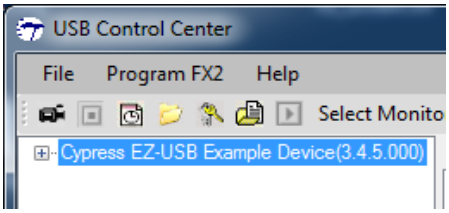
Figure 3. Load the New Device Code into RAM



7. Navigate to **C:\Cypress\Custom_Requests\Output** and select the Custom_Requests.hex file. Note that there is another hex file in **C:\Cypress\Custom_Requests**, “mon-ext-sio1-c0.hex”, which we will explain later. Click OPEN.

8. Watch the USB Control Center left panel. If your PC sound is on, you will hear the “USB disconnect” sound immediately followed by the “USB connect” sound, and a new device will appear (see Figure 4). As a sanity check, the seven-segment decimal point should be lit, and the four LEDs at the top of the board should be OFF.

Figure 4. The New USB Device



This is Cypress ReNumeration™ at work. The initial device (in Figure 1) is a USB loader that is hard-wired into the FX2LP device. The “NO EEPROM” switch setting ensures that the loader ID information in the FX2LP chip is used, rather than anything stored in external EEPROM. After your code is loaded into FX2LP RAM, the loader electrically disconnects and reconnects as the new USB device—the one defined by your custom code. As you develop code, you will repeat steps 6 and 7 for every new code build to be tested.

Note: You can also make your code a permanent addition to the FX2LP board by moving the EEPROM ENABLE switch to its “EEPROM” (up) position, moving the EEPROM SELECT switch to its “LARGE EEPROM” (up) setting, then selecting “64KB EEPROM” in step 6. The file to load is Custom_Requests.iic. After programming, disconnect and re-connect to USB and your program will automatically load and run.

The code we just loaded implements a custom USB device that responds to the custom (vendor) requests shown in Table 5. Note that the fields in the left column are exactly the ones shown in Table 3, but our code assigns these fields to suit the purposes of the custom USB device. Also note that there are many “don’t cares” (the x entries) that are not required by the requests. You could easily define parameters for your device in these fields.

You are free to create as many vendor requests as you wish and to define the fields in Table 5 in any way that suits your application. This is the power of vendor requests.

Table 5. Our Custom Vendor Requests

	Upload	**Renum**	**I2C Rate**	**7 Seg**	**LED**	**EEPROM Write**
bRequest SETUPDAT [1]	**0xa2**	**0xa3**	**0xa4**	**0xa5**	**0xa6**	**0xa7**
wValueL SETUPDAT [2]	addrL	x	0:100 1:400	digit	L[3:0]	addrL
wValueH SETUPDAT [3]	addrH	x	x	x	x	addrH
wIndexL SETUPDAT [4]	0:EEPROM 1:RAM	x	x	x	x	x
wIndexH SETUPDAT [5]	x	x	x	x	x	x
wLengthL SETUPDAT [6]	LenL	x	x	x	x	LenL
wLengthH SETUPDAT [7]	LenH	x	x	x	x	LenH

The code defines six custom (vendor) request columns. The particular request (Upload, Renum, etc.) is selected by the bRequest field. For example, bRequest=0xa5 sends a hex digit to the FX2LP board seven-segment readout for display. Use the USB Control Center to test this request as follows.

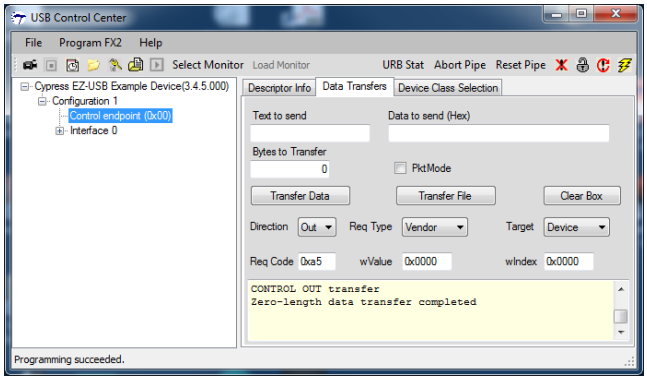
Configure the USB Control Center “Data Transfers” tab as shown in Figure 5. Make sure the “Control endpoint (0x00)” item is selected in the left panel and make the following settings:

- “Bytes to Transfer” to 0
- Direction to “Out”
- Req Type to “Vendor”
- Req Code to 0xa5

The “Target” is unimportant. Leave wValue and wIndex = 0x0000, then press the Transfer Out button. The digit “0” should appear in the FX2LP board’s seven-segment readout. Change wValue to 7 and press “Transfer Data” again. The readout updates to the digit 7. Now try wValue=0x0F. Then try wValue=0x10. Any value outside the range 0x00 - 0x0F lights only the decimal point.

You may wonder why the Bytes To Transfer field is set to 0 even though one byte of data is sent to the seven-segment readout. This is because all the data we need for the readout fits into the seven Standard Request bytes, so a separate data packet is not necessary. This will be true of many vendor requests that simplify the transfer (and code).

Figure 5. Data Transfers Tab Setup

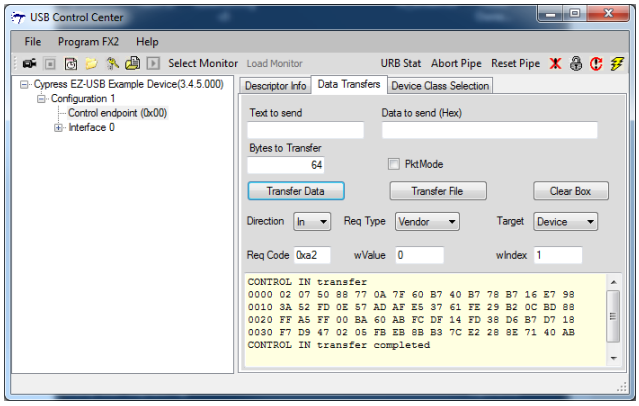


Now test the LED Request (Table 5, last column) by setting Req Code to 0xa6, wValue to 5, and pressing Transfer Data. You should see D4 and D2 illuminate. The wValue number is an LED bit map with bit3=D5, bit2=D4, bit1=D3 and bit0=D2. Therefore, 5 = 0101 = off-on-off-on.

For testing the “Upload” request, keep the USB control center fields as in Figure 6 and press “Transfer Data”. The first 64 bytes of program RAM starting from the address 0x0000 will be uploaded and displayed.

To upload the data from the EEPROM, the wIndex value should be kept as 0.Other fields can be kept same as in the Figure 6. Also make sure that the EEPROM enable is in “EEPROM” (up) position and EEPROM SELECT switch is in “LARGE EEPROM” (up) position.

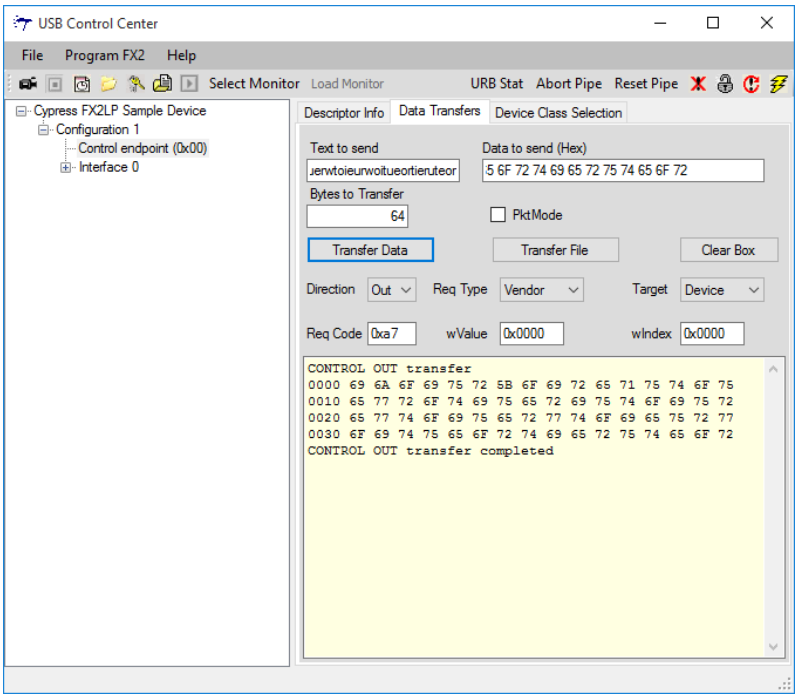
Figure 6. Inspect Onboard RAM Contents



This request uses an IN packet in addition to the standard request packet to transfer data. This is because the fields in a standard request are “write-only”, sending requests to a device but not accepting data on their own. The “Bytes To Transfer” field should contain the number of bytes that you want read and “Direction” field should be changed to IN.

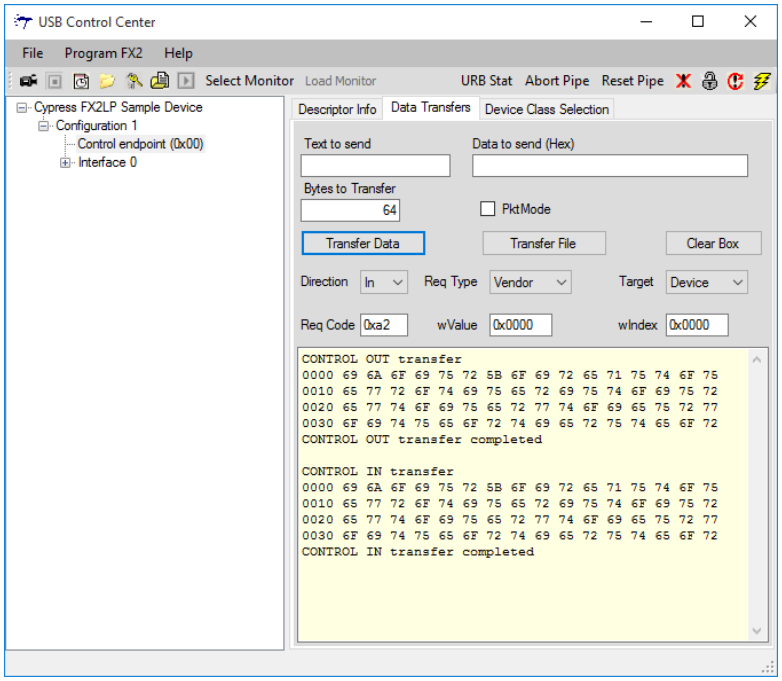
Finally, test the “EEPROM Write” request by filling out the fields as shown in the Figure 7 and press “Transfer Data” after providing the data in the “Text to Send” field. This will perform EEPROM write and the count provided in the “Bytes to transfer” will be written to the large EEPROM through I^2C writes.

Figure 7. EEPROM Write



The “EEPROM Write” command can be verified by performing the “Upload” request which will read the EEPROM data and display in the control center. To perform this read operation, set the fields as provided in the Figure 8 and press “Transfer data”.

Figure 8. Verifying “EEPROM Write”

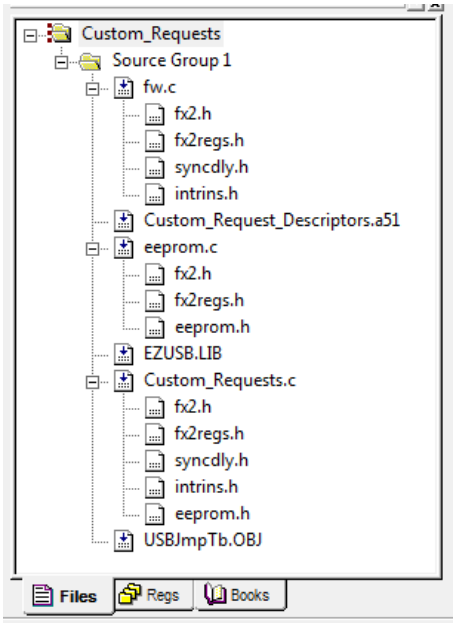


The vend_ax code example provided in the FX2LP DVK also demonstrates the implementation of different vendor commands. This can be accessed from the following path once the FX2LP DVK is installed. <CY3684 DVK installation path>\Cypress\USB\CY3684_EZ-USB_FX2LP_DVK\1.1\Firmware\Vend_ax

5 How It Is Done

Launch the Keil project for this FX2LP code by navigating to C:\Cypress\Custom_Requests and double-clicking on the Keil project file called Custom_Requests.Uv2. This brings up the Keil integrated development environment (IDE) and displays the data in the “Files” tab, as shown in Figure 9.

Figure 9. Keil Project Files



5.1 fw.c

This file, written by Cypress, is called the USB Frameworks. All USB standard commands are processed by this file so your code does not have to do it. Code is included to handle USB standard requests, including the one we’re investigating − vendor requests. Usually you do not need to modify the fw.c frameworks file. However, you can trace the firmware in this file to see how the device responds to the various Chapter 9 requests during enumeration.

5.2 Custom_Request_Descriptors.a51

This assembly language file defines USB descriptors and contains all USB-related definitions. In this file, you can make the required modifications to configure USB resources. Typical modifications include changing the VID/PID (Vendor and Product ID), endpoint configuration, interface class, and informative strings.

5.3 EEPROM.C

This file includes EEPROM-related functions. This file shows how to perform an EEPROM read/write and how to get page size information.

5.4 EZUSB.LIB

The EZ-USB library is an 8051 .LIB file that implements functions common to many firmware projects (such as I2C read/Write routines, delay, and USB disconnect functions). Because these functions do not need to be modified, they are provided in library form. However, the FX2LP development kit includes the source code for the library in the event that you need to modify a function or if you just want to know how something is done.

5.5 Custom_Requests.c

You can study this file to learn how vendor commands work. It can be used as a starting point to create your own vendor commands. Six vendor commands are defined in this file. Note that Cypress reserves vendor commands 0xa0-0xaf for purposes such as code download and app notes, such as this one.

5.6 USBJmpTb.OBJ

This is an object file that contains the ISR jump table for USB and GPIF interrupts.

6 Code Analysis

Creating a vendor command is a simple two-step process. Refer to the example Custom_Requests.c code. The first step is to give the vendor commands symbolic names:

```
#define VR_UPLOAD           0xa2
#define VR_RENUM            0xa3
#define VR_I2C_RATE         0xa4
#define VR_7SEG             0xa5
#define VR_LEDS             0xa6
#define VR_EEPROM_WRITE     0xa7
```

When the FX2LP USB Frameworks code receives a vendor request, it calls the function:

BOOL **DR_VendorCmnd**(void)

The heart of this function is a switch statement with one case entry per custom request. Here is the seven-segment update case (with abbreviated comments):

```
switch(SETUPDAT[1])
{
case VR_7SEG:
    while (I2CS & bmSTOP); // wait for STOP
    I2CS = bmSTART;  // set START bit
    I2DAT = 0x42;// I2C address
    while (!(I2CS & bmDONE));      // wait for done
    val = SETUPDAT[2];      // wValueL=digit 0x00-0x0F
    if(val<=0x0F)
    I2DAT = ~LOOKUP7[SETUPDAT[2]];// 0-F
    else   I2DAT = 0x7F;    // d.p. only
    while (!(I2CS & bmDONE));      // wait for done
    I2CS = bmSTOP;          // set the STOP bit
    break;
```

The switch statement examines SETUPDAT[1], which is the bRequest field of the Standard Request (Table 3). Note that the USB Frameworks code saved the work of examining the SETUPDAT[0] field in the SETUP packet that just arrived. Precisely because the bmRequestType byte (Table 3) indicated a vendor request, the Frameworks called the DR_VendorCmnd function. Simply fill in code for the various bRequest (SETUPDAT[1]) values we defined for our custom requests in Table 5.

The first four lines in the case statement (a) wait for the FX2LP I2C controller to be free; (b) set the START bit; (c) send the I2C address of 0x42, which is the I2C expander chip on the FX2LP board that connects to the seven-segment readout; and (d) wait for the address byte to transmit.

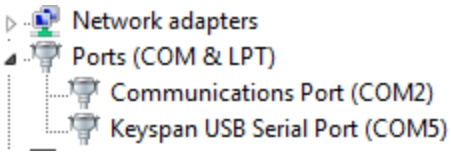
Then the code examines SETUPDAT[2], which is the wValueL byte in the request (Table 5). We defined this byte to be a hex digit from 0x00 to 0x0F. The code looks up the segments if the value is in range. Otherwise, it displays only the decimal point. Finally, the code sends the I2C STOP signal and we're done.

This request has bytes SETUPDAT[3-7] "empty"; you could think of other uses for these bytes to transmit to your custom device.

7 Debugging the Code

This example uses about 3 kbytes of code memory, which is inside the 4-kbyte limit of the Keil demo tools. You can modify the code as long as you do not exceed the limit. Of course, any serious FX2LP project requires a full tool set; contact Keil for that.

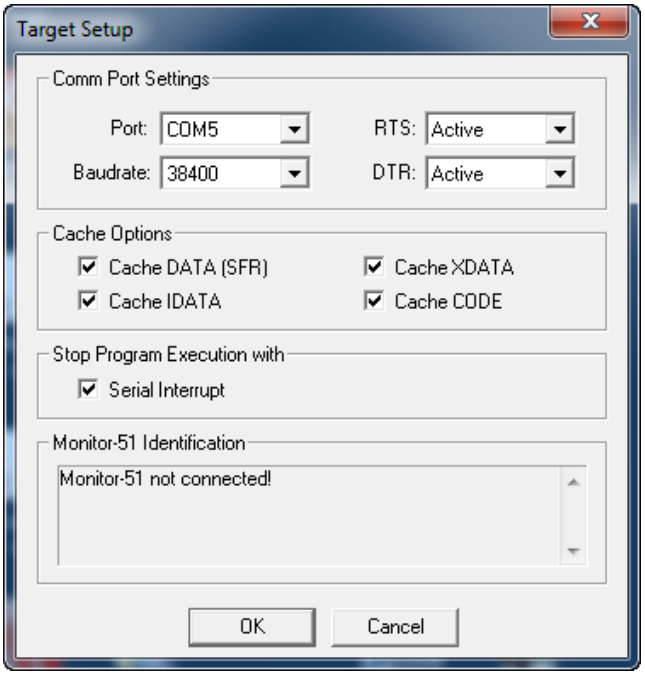
To use the debug capability of the FX2LP board and Keil tools, attach the FX2LP board to a PC serial port. This probably requires a USB-to-serial converter, such as the Keyspan USA-19HS (Amazon). Install the converter according to the vendor instructions, then use the Windows Device Manager to find its COM port number. For example:



Plug the serial adapter into SIO-1 (top) DB-9 connector. Here's how to tell the Keil IDE to use this port:

1. In the Files tab, right-click on the top item, "Custom_Requests", and select "Options for Target Custom_Requests"
2. Select the Debug tab
3. Click the "Settings" button in the upper right
4. Choose your COM port in the "Port" dropdown
5. Set the baud rate to 38400
6. Make sure "Serial Interrupt" is checked (see Figure 10)

Figure 10. Comm Port Settings



The last step is to download a serial debugger, which runs alongside your development code. Go back to your USB Control Center, select 'Program FX2/RAM", and navigate to the Cypress\Custom_Requests folder. You should see the file "mon-ext-sio1-c0.hex; click on it to download it onto the FX2LP board. A green LED near the USB connector turns ON to indicate that the monitor is running.

Note: In the file “mon-ext-sio1-c0.hex, ‘mon’ means monitor, ‘ext’ means use external RAM, “sio1” means use the SIO-1 serial connector (next to the USB connector), and “c0” specifies the load type. AN42499 − Setting Up, Using, and Troubleshooting the Keil(TM) Debugger Environment” has details.

Select Debug/Start-Stop Debug Session or click the magnifying glass to start the session. Run the code by selecting Debug/Go. After the code is running, the USB Control Center can send vendor commands as before.

To stop the debugger, select Debug/Stop Running; to exit the debugger, first stop running, then select Debug/Start-Stop Debug Session.

If you are interested in debugging your code by printing messages on HyperTerminal or Tera Term, then please refer to AN58009 − Serial (UART) Port Debugging of FX1/FX2LP Firmware.

8 Customizing the USB Control Center

Here is the path to the Control Center folder:

C:\Cypress\Cypress Suite USB 3.4.7

\CyUSB.NET\examples\Control Center

This folder contains Visual Studio solution folders to allow customization of the C# code. For example, you may want to change the default panel settings (to Vendor Req Type, Out direction, and 0 transfer bytes) so you do not have to do it every time you start the app to test the custom device. At a deeper level, the source code can serve as a basis for fully custom Windows apps that talk to custom USB devices of your design. For more details on how to design the host application, please refer to the application note, AN70983 - Designing a Bulk Transfer Host Application for EZ-USB® FX2LP™/FX3™.

9 The Last Step

The “glue” that bridges Windows code to your FX2LP Custom Device is a Cypress-written Windows driver called “cyusb.sys” and its companion information file (cyusbfx1_fx2lp.inf). Cypress makes this driver available for end customers that are making products based on FX2LP by providing Cypress VID and custom PID; please create a tech support case to get more details.

10 Resources and Additional Information

- AN65209 – Getting Started with FX2LP™
- Detailed information about using the KEIL debugger environment is in the app note AN42499 − Using and Troubleshooting the KEIL Debugger Environment.
- Detailed information about serial port debugging is in AN58009 − Serial (UART) Port Debugging of FX1/FX2LP Firmware.
- Bootloader information is in AN50963 − EZ-USB FX1/FX2LP Boot Options.
- To find good reference materials and examples of USB Vendor Command Design and HID Class Usage, go to Jan Axelson’s Lakeview Research website.
- USB Complete, by Jan Axelson, is a book that describes USB details clearly and completely.

11 Summary

This application note explained a built-in USB feature called vendor requests. The requests allow you to create your own USB commands. The example firmware provided with this application note implements six custom commands to interact with the FX2LP development board. A Windows app, written in C#, demonstrates Windows communication, and allows full testing of the custom device. Source code and step-by-step instructions teach you how to run and debug the code on the FX2LP.

Document History

Document Title: AN45471 - Create Your Own USB Vendor Commands Using FX2LP™

Document Number: 001-45471

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	2505120	LIP	01/16/2009	New specification.
*A	3155612	LIP	01/27/2011	Added Resources and Additional Information. Updated links across the document.
*B	4109959	RSKV	08/30/2013	Updated attached Associated Project. Updated Vendor Commands: Updated description. Updated Vendor Command Example Code: Updated description. Removed Vendor Command Testing. Added Test Drive. Added How It Is Done. Added Code Analysis. Added Debugging the Code. Added Customizing the USB Control Center. Added The Last Step. Updated Test Drive. Added links to useful materials across the document. Updated to new template.
*C	4235752	LIP	01/03/2014	Updated to new template. Completing Sunset Review.
*D	5180445	MDDD	03/30/2016	Added link in “Related Resources” in page 1. Added High-Speed code examples link. Updated Abstract. Updated Resources and Additional Information: Added “AN65209 – Getting Started with FX2LP™” in the list. Updated to new template.
*E	5673611	HYLE / HPPC	08/07/2017	Updated attached Associated Project (Added EEPROM Write command). Updated Abstract. Updated Introduction: Updated description. Updated Vendor Commands: Updated 02: Vendor Request: Updated description. Updated Test Drive: Updated Table 5; updated description; and also added Figure 7 and Figure 8. Updated How It Is Done: Updated Custom_Requests.c: Updated description. Updated Code Analysis (Added “#define VR_EEPROM_WRITE 0xa7”). Updated Summary. Updated to new template. Completing Sunset Review.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

PSoC is a registered trademark and PSoC Creator is a trademark of Cypress Semiconductor Corporation.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

© Cypress Semiconductor Corporation, 2009-2017. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.