`

THIS SPEC IS OBSOLETE

Spec No: 001-14558

Spec Title: Implementing an SPI Master on EZ-USB® FX2LP™ - AN14558

Sunset Owner: DBIR

Replaced By: None

AN14558

Implementing an SPI Master on EZ-USB® FX2LP™

Author: Sonia Gandhi, Nikhil Naik
Associated Project: Yes
Associated Part Family: CY7C6801XA
Software Version: None

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.

Introduction

EZ-USB FX2LP is a single-chip solution to implement high-speed USB 2.0 peripherals. FX2LP integrates a USB controller and an 8051 MCU to handle USB transfers and to create interfaces to other system devices. Although the 8051 contains two UARTs, it does not include a hardware SPI (Serial Peripheral Interface) unit. However it is possible to implement an SPI master using one of two methods. First, an SPI master can be implemented using GPIO pins and writing "bit-bang" code to directly control the GPIO pins. Second, one of the 8051 UARTs may operate as an SPI master.

Figure 1. Basic SPI.

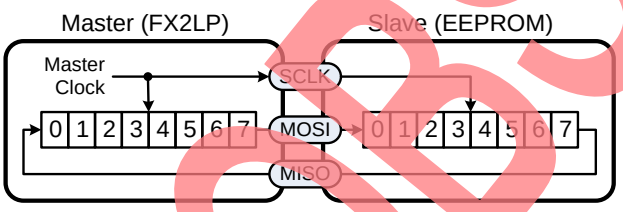


Figure 1 illustrates the basics of SPI. Two shift registers-one in the master and the other in the slave connected in a ring. The master supplies the common clock SCLK to both shift registers. The data pins are named MOSI for Master Out/Slave In, and MISO for Master In/Slave Out. Data is transmitted MSB-first. After eight clocks, the bytes in the master and slave exchange places.

Most SPI slave devices include an input pin named Chip Select (CS#) or Slave Select (SS#). This active-LOW signal enables SPI transfers. When the master de-asserts SS#, the slave tristates its MISO output driver and ignores SCLK transitions. Some SPI slaves use the de-assertion of the CS# signal to reset internal logic.

There are four possible combinations of SCLK polarity and phase, abbreviated as CPOL and CPHA. The SPI Mode is abbreviated as (CPOL, CPHA). Most SPI devices, including the EEPROM used in this note, operate equivalently in modes (0, 0) and (1, 1). In these modes the data is sampled on the rising edge of the SCLK. The only difference is the quiescent state of SCLK: LOW in mode (0, 0) and HIGH in mode (1, 1).

SPI prescribes no protocol. For example, there is no provision to address different slaves on the bus. Instead, the master provides individual CS# signals to each slave device. For this reason, the designer must consult the data sheet for specific SPI device protocol.

A common SPI slave is an EEPROM. EEPROMs are available in various densities, and over time they have adopted a common protocol. This application note shows how to connect FX2LP to a 16-Kb (2048x8) EEPROM. These EEPROMs are available from multiple vendors, usually with "160" in the part number to indicate the density. For example, this note uses a Microchip 25AA160B device for testing, which Atmel offers as an AT25160B, ON as a CAT25160, Renesas as an R1ECX25016, ST as an M95160, Rohm as a BR25L160, etc. There are minor differences between manufacturers and devices in features such as page size, so be sure to consult the data sheet for the device you choose.

These EEPROMs operate using a wide supply voltage range, all of them compatible with 3.3 V, so they can be directly connected to FX2LP pins.

Test Hardware

The example code in this note runs on a Cypress FX2LP Development board, included as part of the FX2LP Evaluation Kit, available from Cypress website. Installing the software in the kit installs the Windows driver needed for the testing.

The 'Bit-Bang' Approach

In this approach, the 8051 directly controls ("bit-bangs") GPIO pins to communicate with an SPI EEPROM.

Figure 2. FX2LP to EEPROM Connections

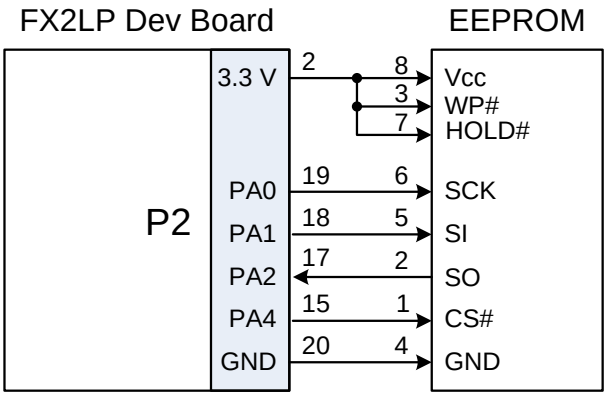
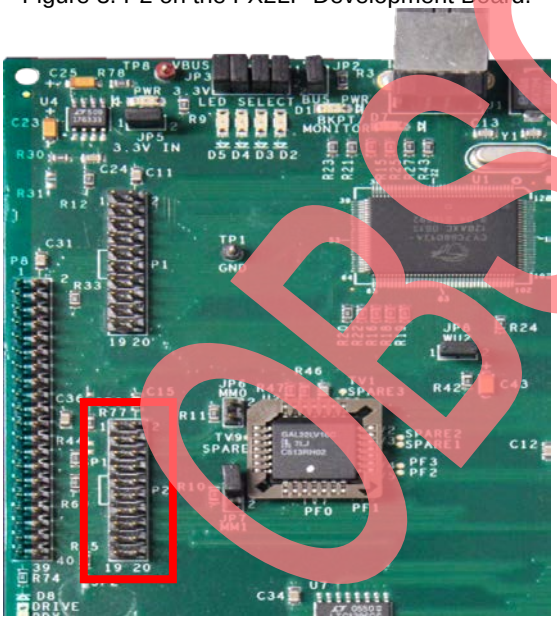


Figure 2 shows the connection between the FX2LP Development Board and a 25AA160B EEPROM. Figure 3 highlights P2 on the left edge of the FX2LP Development Board. Note that pin 1 is upper-left, and pin 20 is lower-right on J2.

Figure 3. P2 on the FX2LP Development Board.



SPI Byte Write

Code 1 shows the C code to write an SPI byte.

Code 1. C Code to Bit-Bang an SPI Write

```
void SPIByteWrite(unsigned char b)

// caller manages SPI_CS signal
{
  SPI_CLK = 0;
  if(b & 0x80) MOSI = 1;else MOSI= 0;
  SPI_CLK = 1;     SPI_CLK = 0;
  if(b & 0x40) MOSI = 1;else MOSI= 0;
  SPI_CLK = 1;     SPI_CLK = 0;
  if(b & 0x20) MOSI = 1;else MOSI= 0;
  SPI_CLK = 1;     SPI_CLK = 0;
  if(b & 0x10) MOSI = 1;else MOSI= 0;
  SPI_CLK = 1;     SPI_CLK = 0;
  if(b & 0x08) MOSI = 1;else MOSI= 0;
  SPI_CLK = 1;     SPI_CLK = 0;
  if(b & 0x04) MOSI = 1;else MOSI= 0;
  SPI_CLK = 1;     SPI_CLK = 0;
  if(b & 0x02) MOSI = 1;else MOSI= 0;
  SPI_CLK = 1;     SPI_CLK = 0;
  if(b & 0x01) MOSI = 1;else MOSI= 0;
  SPI_CLK = 1;     SPI_CLK = 0;
}
```

Writing the code inline instead of using a program loop makes the SPI transfer as quick as possible. Note that the 8051 MCU has bit addressing, so the MOSI variable (defined as the bit PA1) can be directly set or cleared.

Code 2 shows the protocol to write one EEPROM byte. It is one **case** in a **switch** statement that decodes USB Vendor Requests described later in this note. After enabling the EEPROM by setting its Chip Select pin LOW, the code sends the "Enable Write" command. Then, it toggles CS# pin HIGH to LOW to latch the command into the EEPROM. Next, the code sends the EEPROM Write command followed by two address bytes and one data byte. Returning CS# HIGH triggers the write cycle. The address and data bytes arrive over USB in a SETUP Data packet, which the code accesses in the 8-byte SETUPDAT buffer.

Code 2. Code to write one EEPROM byte.

```
case VR_WRITE_SPI_BB:
  SPI_CS = 0;
  SPIByteWrite(EE_ENABLE_WRITE);
  SPI_CS = 1;//set write-enable latch
  SPI_CS = 0;//ready to write
  SPIByteWrite(EE_WRITE);//command
  SPIByteWrite(SETUPDAT[3]);//addrH
  SPIByteWrite(SETUPDAT[2]);//addrL
  SPIByteWrite(SETUPDAT[4]);//data
  SPI_CS = 1;//trigger the write
break;
```

Figure 4. Logic Analyzer Trace of the Bit-Banged Byte-Write Operation

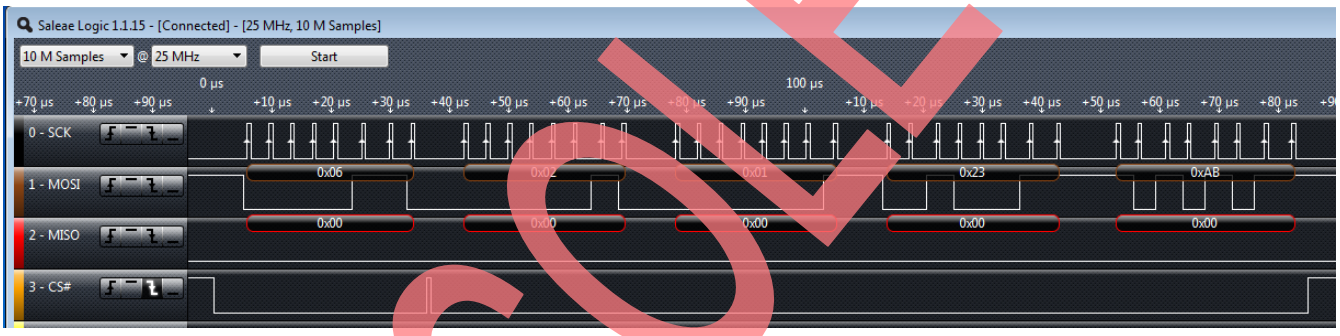


Figure 4 shows the five SPI byte write operations. Writing an EEPROM byte takes about 180 µs with an FX2LP 12 MHz clock. The MOSI line shows the output data, while the MISO line is “don’t care” since no EEPROM data is read. The first byte 0x06 is the ENABLE_WRITE command, followed by CS# toggling from HIGH to LOW. The second byte 0x02 is the WRITE command, followed by address 0x0123 and data 0xAB.

EEPROMs can accept burst writes, whereby a string of bytes follow the data byte (0xAB in Figure 4). This improves write times by requiring the first four bytes to be sent only once per burst. Consecutive writes increment an internal EEPROM address pointer.

SPI Byte Read

Code 3 shows the C code to read an SPI byte.

Code 3. C code to bit-bang an SPI Read

```
unsigned char SPIByteRead(void)
// caller manages SPI_SS signal
{
  unsigned char val = 0;
  SPI_CLK= 0;
  SPI_CLK=1;
if(MISO == 1) val|=0x80; SPI_CLK=0;
  SPI_CLK=1;
if(MISO == 1) val|=0x40; SPI_CLK=0;
  SPI_CLK=1;
if(MISO == 1) val|=0x20; SPI_CLK=0;
  SPI_CLK=1;
if(MISO == 1) val|=0x10; SPI_CLK=0;
  SPI_CLK=1;
if(MISO == 1) val|=0x08; SPI_CLK=0;
  SPI_CLK=1;
if(MISO == 1) val|=0x04; SPI_CLK=0;
  SPI_CLK=1;
if(MISO == 1) val|=0x02; SPI_CLK=0;
  SPI_CLK=1;
if(MISO == 1) val|=0x01; SPI_CLK=0;
  return val;
}
```

As with the write operation, inline coding each bit test gives the fastest possible read operation.

Code 4 shows the protocol to read one EEPROM byte. The code selects the EEPROM by setting CS#=0, and then checks to make sure any previous write operation has completed. It does this by sending a READ STATUS command, then looping until bit 0 in the STATUS byte is 0, indicating that no write operation is in progress. Then the code sends the address bytes as before, finally reading the data byte into the first byte of the ENDPOINT Zero data buffer EP0BUF[0]. Finally, the code arms the IN data for USB transfer by writing a byte count of one. Note that the MCU arms FX2LP USB transfers by writing the lower byte count register EP0BCL, so the upper byte count register EP0BCH should always be loaded first.

Code 4. Code to Read One EEPROM Byte

```
case VR_READ_SPI_BB:
  SPI_CS = 0;
  SPIByteWrite(EE_READ_STATUS);
  //wait until write complete
  while (SPIByteRead() & 0x01);
  SPI_CS = 1;
  SPI_CS = 0;// enable READ
  SPIByteWrite(EE_READ);
  SPIByteWrite(SETUPDAT[3]);// addrH
  SPIByteWrite(SETUPDAT[2]);// addrL
  EP0BUF[0] = SPIByteRead();// data
  SPI_CS = 1;
  EP0BCH=0;
  EP0BCL=1;// arm EP0-IN transfer
break;
```

Figure 5. Logic Analyzer Trace of the Bit-Banged Byte Read Operation

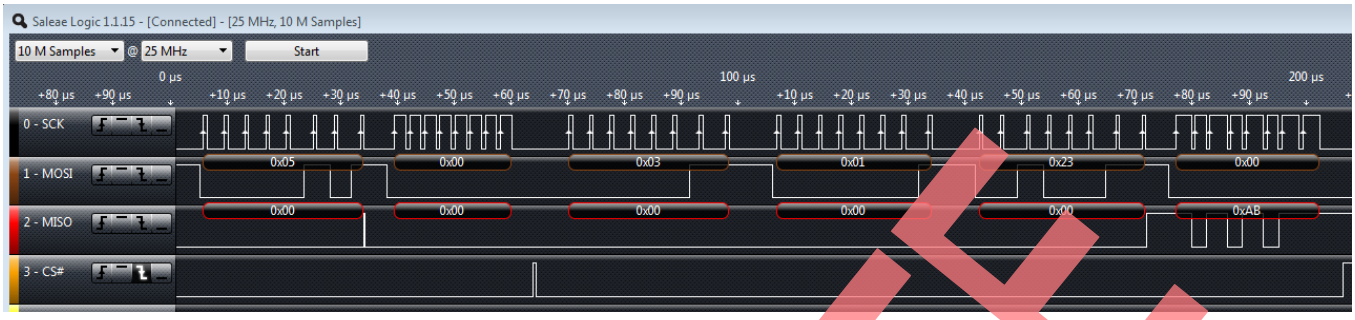


Figure 5 shows a five-byte SPI read operation. The first byte 0x05 is the READ_STATUS command, followed by reading the status byte on the MISO pin. Because bit 0 is 0 (no writes in progress), the sequence proceeds with the READ command 0x03, the two address bytes 0x0123, and finally reading back the data byte 0xAB. Similar to the writes, the MCU code can also read consecutive bytes in a burst.

Serial Port Mode 0 Approach

Both FX2LP UARTs have a Mode 0 that provides an output clock in addition to the serial data input and output pins. This mode can be used to implement an SPI master.

Similar to the bit-bang approach, a GPIO pin provides the EEPROM CS# signal. This example uses UART0 as an SPI master.

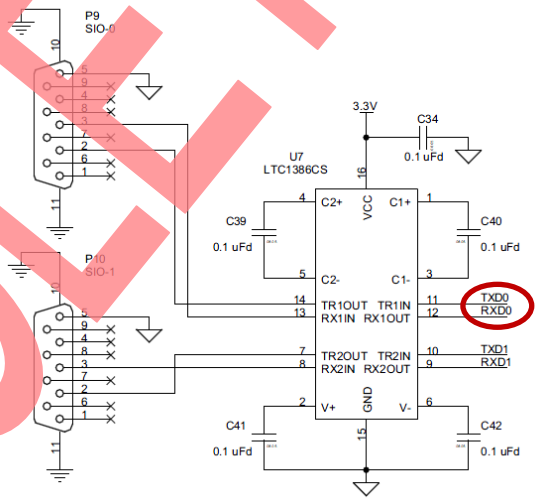
The UART signals are remapped in Mode 0 as shown in Table 1 for UART0:

Table 1. SPI Remapping for UART0 Mode 0

UART0 Signal	SPI Signal	Pin Number on FX2LP DVK
TXD0	SCK	U7-11
RXD0	MISO	U7-12
RxD0OUT	MOSI	PORTE.3 (P6-16)
--	CS#	PORTC.2 (P3-17)

Any GPIO pin can be used for CS#; this example uses PORTC bit 2. The “Available” column shows where the signals can be found on the FX2LP Development Board. As Figure 6 shows, the TXD0 and RXD0 signals must be taken from the input side of the RS-232 level converter U7 to conform to the 3.3 V logic levels required by the EEPROM. This requires soldering wires to U7 pins 11 (TXD0/SCK) and 12 (RXD0/MISO).

Figure 6. Location of RXD0 and TXD0 Signals on the FX2LP Development Board



UART0 is initialized for Mode 0 operation as shown in Code 5. PORT E bit 3 is configured to be the UART output (MOSI), and the PORTC bit 4 driver is turned on to provide the CS# output.

Code 5. UART0 Mode 0 Initialization Code

```
// SPI-UART Initialization
SCON0 = 0x13;      // See Figure 7
PORTECFG = 0x08;   // make PE3->MOSI
SPI_CS_U = 1;      // deselect EEPROM
OEC = 0x04;        // CS# is PORTC.2
```

The 8051 clock speed is set by the CLKSPD[1:0] bits in the CPUCS register. This example uses the default setting of 00, giving a 12 MHz CPU clock. The UART clock is set by bits in the SCON0 register (See Figure 7).

Figure 7. SCON0 Register Bits

SM0	SM1	SM2	REN	TB8	RB8	TI	RI
b7	b6	b5	b4	b3	b2	b1	b0
0	0	0	1	0	0	1	1

SM0-SM1 = 00 to select UART Mode 0. SM2 in Mode 0 selects the baud rate divisor of the CPU clock (0=divide by 12, 1=divide by 4). For this example, SM2=0, for an SCK output of 12 MHz/12=1 MHz. REN=1 enables the receiver. TB8 and RB8 are not used in Mode 0. TI and RI are set to 1 to initialize the UART transmit and receive flags to their ready states.

The SPI byte write and byte read functions are shown in Code 6. The function names are as before, but with a "U" suffix to indicate the UART.

Code 6. UART Mode0 Write and Read Code

```
void SPIByteWriteU (BYTE d)
 {
 TI = FALSE;       //Clear flag
 SBUF0 = swap1[d]; //Write byte
 while (!TI);      //Wait until
transmit done
}

BYTE SPIByteReadU (void)
{
 RI = FALSE;     //Clear flag
 while (!RI);    //Wait until Rx done
 return (swap1[SBUF0]); //Return byte
}
```

Figure 8. Logic Analyzer Trace of the UART Mode0 Byte Write

Figure 8 shows the five SPI byte write operation. The data is identical to Figure 4, but there are three differences compared with the bit-bang example:

1. With a 1 MHz clock, the writes are about twice as fast, with the write cycle taking about 90 microseconds. Setting SM2=1 (Figure 11) would triple this clock rate. Before doing this, make sure that the EEPROM can take a 3-MHz clock rate.
2. The UART clock quiescent state is HIGH instead of LOW. This is an example of SPI mode (1, 1) being equivalent to mode (0, 0). Both modes strobe and sample the data on the clock rising edge.
3. The UART outputs bytes LSB first, so the C code must flip each byte end-to-end. This can be done in a loop with shift and mask instructions. For faster execution than the loop, this example uses a 256-byte lookup table.

Testing

Cypress provides an application called **USB Control Center** as part of its **Suite USB**, available as a free download on the Cypress web site. Complete Microsoft Visual Studio source code is provided to allow study and modification. For convenience, the companion folder to this application note contains a binary version of **CyControl.exe**.

This note tests the SPI read and write routines using USB *Vendor Requests*. Vendor Requests allow you to create your own USB commands. Cypress application note **AN45471** discusses Vendor Requests in detail. This note summarizes AN45471 topics to help understand how Vendor Requests are used to test the SPI byte write and read functions.

This example creates six USB Vendor requests using the request numbers shown in Table 2. The example code uses the Cypress **USB Frameworks**, which is a C application that creates a fully operational USB device that can be customized. The USB Frameworks code routes USB requests to predefined placeholder functions. For example, to respond to Vendor Requests, you write code in the provided **DR_VendorCmd** (**D**evice **R**equest, **Vendor C**ommand) function in the **Vend_SPI.C** file. The first two requests in Table 2 are from AN45471, and the remaining four are unique to this application note.

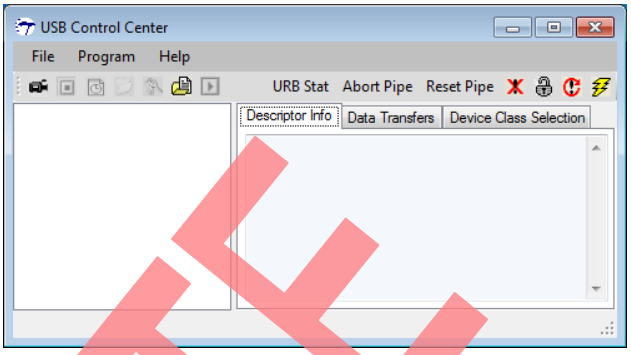
Table 2. USB Vendor Commands for This Application Note

Request	Purpose
0xA5	Update 7-segment readout
0xA6	Update four LEDs
0xA7	Write SPI byte (bit-bang)
0xA8	Read SPI byte (bit-bang)
0xAA	Write SPI byte (UART)
0xAB	Read SPI byte (UART)

1. Start CyControl.exe and you see the screen in Figure 9.

Note: If your version of the USB control center shows many connected USB devices in the left panel, you can simplify the display by clicking the "Device Class Selection" tab and selecting only the first checkbox to display only Cypress devices.

Figure 9. USB Control Center Startup Screen



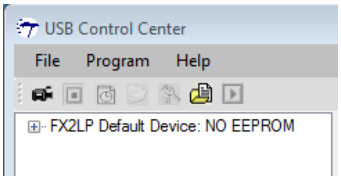
2. On the FX2LP Development Board, move the two slide switches at the lower left to the "down" position. This disables program loads from onboard EEPROMs and enables the USB loader. Verify that the jumpers are set to the states shown in Table 3.

Table 3. FX2LP Development Board Jumper Settings

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

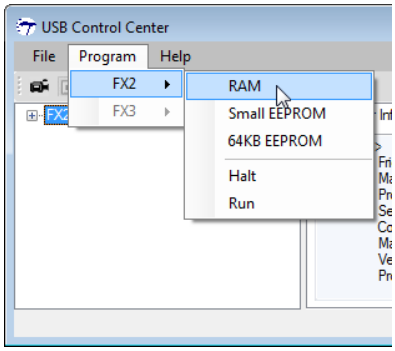
3. Plug the FX2LP Development Board into a USB port. The board appears as in Figure 10. "Default Device" means that FX2LP has enumerated as a default USB device, capable of loading hex code into its internal RAM for execution. This default USB device is hard-wired into FX2LP, requiring no code.

Figure 10. FX2LP Develop Board Detected



4. Select the device entry, and then select **Program** > **FX2** > **RAM** (See Figure 11).

Figure 11. Download Hex Code Into FX2LP RAM

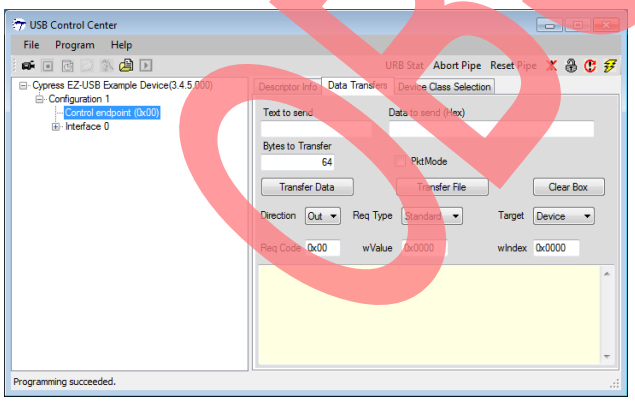


5. Navigate to the **Vend_SPI.hex** file in the VEND_SPI example folder provided with this application note. Double-click the file to load it into FX2LP RAM. You will hear the USB disconnect sound on the PC, followed by the reconnect sound as FX2LP enumerates as a new device. This new device is the one defined by the loaded file.

Note: You can review and modify the example **Vend_SPI** project using the Keil tools provided in the Cypress download. *Every time you modify and rebuild your code, creating a new **.hex** file, you must press the RESET button on the FX2LP Development Board to reconnect the board as the USB code loader.*

6. Expand the device tree and select the **"Control endpoint(0x00)** entry. Then select the Data Transfer tab. The screen should look like Figure 12.

Figure 12. USB Control Center is Ready to Conduct EP0 Transfers



The fields in the USB control center allow you to specify bytes in a SETUP packet, and receive bytes over Endpoint 0. A USB SETUP packet has the format shown in Table 4.

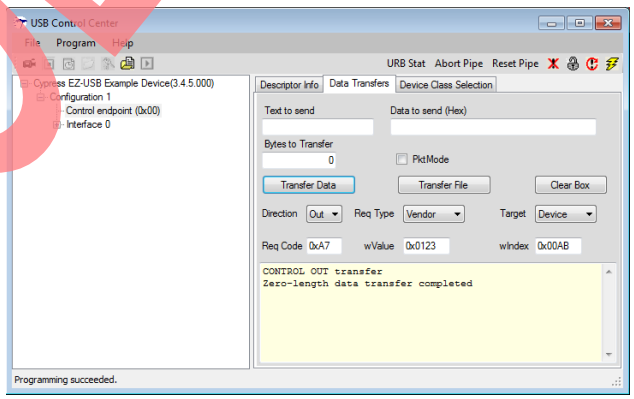
Table 4. The Eight Bytes in a SETUP Packet

Index	Field	Meaning
0	bmRequestType	Direction, Type, and Target
1	bRequest	Specific Request (Refer Table 2)
2	wValueL	Command-specific value (we choose EEPROM address).
3	wValueH	
4	wIndexL	Command-specific index (we choose EEPROM data in low byte).
5	wIndexH	
6	wLengthL	Number of bytes if the request includes a data stage
7	wLengthH	

The USB Control Center uses dropdown lists to fill in the required SETUP bytes. By selecting **Req Type** to "Vendor", the Value and Index fields can be anything you define them to be. In the SPI test, you can designate wValue to be the EEPROM address and wIndex to be the EEPROM data to write.

To write the byte 0xAB to EEPROM location 0x0123 using the bit-bang approach, set the USB Control Center settings as in Figure 13.

Figure 13. Settings for a Bit-Bang Byte Write



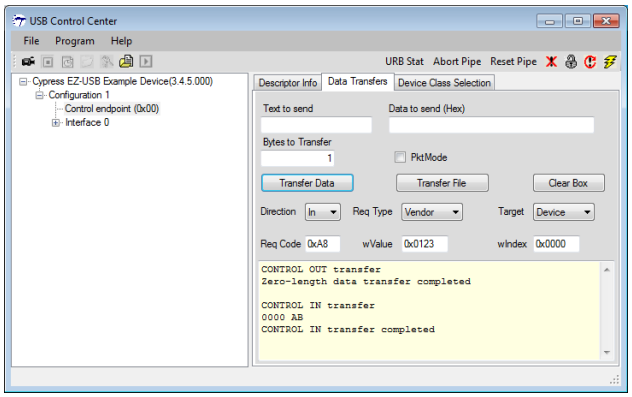
Specifically:

1. Set the **Bytes To Transfer** field to 0. All the Vendor Request bytes are contained in the SETUP packet, so no data stage is required.
2. **Direction** to OUT
3. **Req Type** to Vendor
4. **Target** to Device
5. **Req Code** to 0xA7 (Table 2)
6. **wValue** to the address 0x0123
7. **wIndex** to the data byte 0x00AB

Press the **Transfer Data** button. The text in the yellow box indicates that the transfer was successful. These settings produced the logic analyzer trace in Figure 4.

To read back the EEPROM value, change the settings to match Figure 14.

Figure 14. Settings for a Bit-Bang Read



Specifically:

1. Change **Bytes To Transfer** to 1. This tells the USB Control Center to expect one byte in the IN transfer, which is the byte read from the EEPROM.
2. Change Direction to IN.
3. Change Req Code to 0xA8 (See Table 2).
4. Change wIndex to 0x0000. This is not mandatory, but it makes it clear that the returned byte did not come from this field.

Press the **Transfer Data** button. The byte you wrote (0xAB) is returned. These settings produced the logic analyzer trace in Figure 8.

To repeat the tests using the UART code, use the above instructions, but substitute 0xAA for the write and 0xAB for the read from the SPI EEPROM (See Table 2).

Selecting the Approach to Use

To evaluate which approach to use, consider the following factors:

- Pin assignment. The bit-bang approach gives the freedom to assign any GPIO pins. This may help with FX2LP resource allocation and PCB routing.
- UART utilization. If the application requires two UARTs, use the bit-bang approach.
- Code size. If the UART approach uses a lookup table to flip the bytes, its code size is larger than the bit-bang approach.
- Interrupts. If you want the SPI operations to be interrupt-driven, use the UART which has interrupt provision for send and receive. The bit-bang approach does not.

Conclusion

This application note presented two methods for implementing an FX2LP SPI master. The bit-bang approach uses GPIO pins, and the UART approach uses one of the internal UARTS. An example Keil project contains C code with functions to implement both approaches. The code is tested using the Cypress USB Control Center, which can initiate USB Vendor (custom) requests.

References

- EZ-USB® Technical Reference Manual (Document #001-13670)
- CY7C68013A/CY7C68014A/CY7C68015A/CY7C68016A EZ-USB FX2LP™ USB Microcontroller High Speed USB Peripheral Controller Data Sheet (Document #38-08032)
- AN65209-Getting Started with FX2LP
- AN45471 - Create Your Own USB Vendor Commands Using FX2LP™
- SPI Serial Flash (25AA160B) Data Sheet

Document History

Document Title: Implementing an SPI Master on EZ-USB® FX2LP™ - AN14558

Document Number: 001-14558

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	968201	DBIR	04/13/2007	New Application Note.
*A	3170872	DBIR	02/11/2011	Updated in new template.
*B	3182387	DBIR	02/25/2011	No technical updates.
*C	3610200	DBIR	05/07/2012	Updated in new template. No technical updates. Completing sunset review.
*D	3916605	OSG / NIKL	02/28/2013	Updated Document Title to read as “Implementing I/O Bit-Bang SPI Interface with EZ-USB FX2LP™ - AN14558”. Changed “SPI Device” to “FLASH” in all instances across the document. Removed “Code for SpiFileWrite() and SpiFileRead() Sequence”. Added a project and a test procedure to run the project.
*E	3931944	NIKL	03/13/2013	Updated Projects (Attached files along with this document).
*F	4311684	RSKV	03/17/2014	Merged AN67442 Updated Projects (Attached files along with this document).
*G	4631096	DBIR	01/19/2015	Obsolete document.

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

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc cypress.com/go/plc
Memory	cypress.com/go/memory
Optical Navigation Sensors	cypress.com/go/ons
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 5

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Technical Support

cypress.com/go/support

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

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com