CY3682 Design Notes

Introduction

The SX2 USB interface works with any standard microprocessor or digital signal processor and adds USB 2.0 support for any peripheral design. The EZ-USB 3682 SX2 Development kit includes a Cypress FX 8051 processor and example firmware to master the SX2. This document describes the FX example firmware, fx2sx2. This firmware can be used as a starting point for development with the SX2 using another processor as external master.

This example firmware has the following sections: Initialization, Interrupt Service Routine, Read Register, Write Register, Write Descriptor, Data Loopback, and Endpoint 0. The firmware is written in the C language and structured so that most of it can be reused in any application.

The functions:

- low_level_command_write
- low_level_command_read
- low_level_fifo_write
- low_level_fifo_read

are specific to the FX processor. These functions generate the proper timing as described in the SX2 Datasheet for asynchronous command and FIFO reads and writes. These functions must be replaced by hardware specific routines for the particular external master used to control the SX2. **It is up to the system designer to change or replace these functions with their own hardware specific functions which generate the proper timing as outlined in the SX2 datasheet.**

Initialization

This example creates four 512-byte double buffered endpoints. The FX uses an asynchronous interface and default SX2 polarities and it enables all the SX2 interrupts. A list of the registers and values used in this example is shown in *Table 1*. The external master can perform other tasks while waiting for the SX2 to initialize. For more details, consult the SX2 datasheet. The description of the initialization process follows.

After any hardware or FX processor initialization, the FX brings the SX2 out of reset by sending a signal using an FX general purpose I/O pin. Then the FX waits until the SX2 is ready to accept commands; the READY interrupt signals when the SX2 is ready.

Once the SX2 is ready, the FX turns on LED0 and writes all of the SX2 register values it needs to setup the SX2 for the application. Next, the FX writes its descriptor string to the SX2. Note: if the EEPROM initialization is used, then the READY interrupt is replaced by the ENUM_OK interrupt, and the host processor can then write the SX2 registers.

After the SX2 has its descriptor information, it connects and enumerates. The FX firmware waits for the ENUM_OK interrupt then, after the FX receives the ENUM_OK interrupt, it turns on LED1 on the FX PCB.

Once the SX2 completes enumeration, the external master must check and see whether the SX2 enumerated at High or Full speed. After the FX checks the HSGRANT bit, it adjusts the IN PACKET LENGTH so that the SX2 knows when to automatically send packets to the USB host. The FX will turn on LED2 if the SX2 enumerated at High speed.

The last initialization task is for the FX to flush all of the SX2 FIFOs.

The initialization process is shown in *Figure 1* and *Figure 2*.

Initialization Code

```
OUTA = 0;                      //Reset the SX2 with a FX GPIO
EZUSB_Delay(1);                //Wait a minimum of 200 microseconds
OUTA = 1;                      //Bring the SX2 out of reset
EZUSB_Delay(1);                //Wait until SX2 is going

//This code is for self powered devices which do not use the EEPROM to enumerate
#ifndef BUSPOWER

while (!sx2_ready);            //Wait until SX2 gives us a ready interrupt

ledX_rdvar = LED0_ON;          //Light LED0 to indicate SX2 is ready

for (i = 0; i < sizeof(regs); i++)    //Setup SX2 with all default values
{
        WriteRegister (regs[i], regsvalue[i]);
}
```

```

    WriteDescriptor();            //Load entire descriptor into SX2

    #endif

    while (!enum_ok);             //Wait until the SX2 has enumerated

    ledX_rdvar = LED1_ON;         //Light LED1 to indicate that we're enumerated

    if (!(ReadRegister (0x2D) & 0x80))    //Check if we have not enumerated at Highspeed
    {
            WriteRegister (0x0A, 0x20);           //Set IN packet length to 64
            WriteRegister (0x0B, 0x40);
            WriteRegister (0x0C, 0x20);           //Set IN packet length to 64
            WriteRegister (0x0D, 0x40);
            WriteRegister (0x0E, 0x20);           //Set IN packet length to 64
            WriteRegister (0x0F, 0x40);
            WriteRegister (0x10, 0x20);           //Set IN packet length to 64
            WriteRegister (0x11, 0x40);
            ledX_rdvar = LED2_OFF;                //Dim LED2 to indicate full speed
    }
    else
    {
            ledX_rdvar = LED2_ON;                 //Light LED2 to indicate high speed
    }


    WriteRegister (0x20, 0xF0);                   //Flush the FIFOs to start
```

Number	Register	Value
0x01	IFCONFIG	0xC8
0x02	FLAGSAB	0x00
0x03	FLAGSCD	0x00
0x04	POLAR	0x00
0x06	EP2CFG	0xA2
0x07	EP4CFG	0xA0
0x08	EP6CFG	0xE2
0x09	EP8CFG	0xE0
0x0A	EP2PKTLENH	0x02
0x0B	EP2PKTLENL	0x00
0x0C	EP4PKTLENH	0x02
0x0D	EP4PKTLENL	0x00
0x0E	EP6PKTLENH	0x22
0x0F	EP6PKTLENL	0x00
0x10	EP8PKTLENH	0x22

Number	Register	Value
0x11	EP8PKTLENL	0x00
0x12	EP2PFH	0x81
0x13	EP2PFL	0x00
0x14	EP4PFH	0x81
0x15	EP4PFL	0x00
0x16	EP6PFH	0x81
0x17	EP6PFL	0x00
0x18	EP8PFH	0x81
0x19	EP8PFL	0x00
0x1A	EP2ISOINPKTS	0x01
0x1B	EP4ISOINPKTS	0x01
0x1C	EP6ISOINPKTS	0x01
0x1D	EP8ISOINPKTS	0x01
0x2E	INTENABLE	0xFF

Table 1.Initial Register Values From Header File

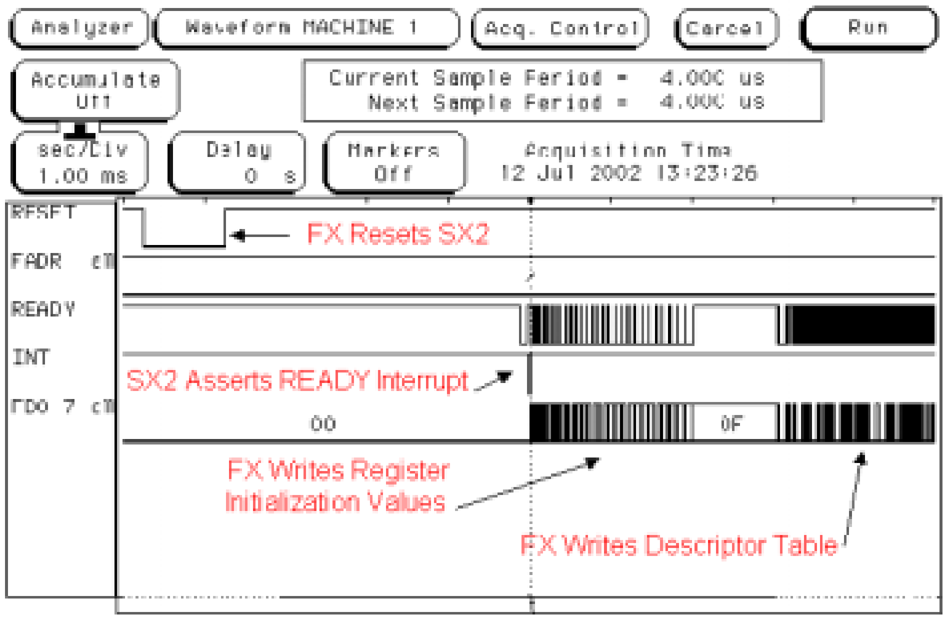

Analyzer
Waveform MACHINE 1
Acq. Control
Cancel
Run
Accumulate Off
Current Sample Period = 4.000 us
Next Sample Period = 4.000 us
sec/Div 1.00 ms
Delay 0 s
Markers Off
Acquisition Time 12 Jul 2002 13:23:26
RESET
FADR
READY
INT
FD0 7
FX Resets SX2
SX2 Asserts READY Interrupt
00
0F
FX Writes Register Initialization Values
FX Writes Descriptor Table

Figure 1. SX2 Initialization From Reset to Descriptor

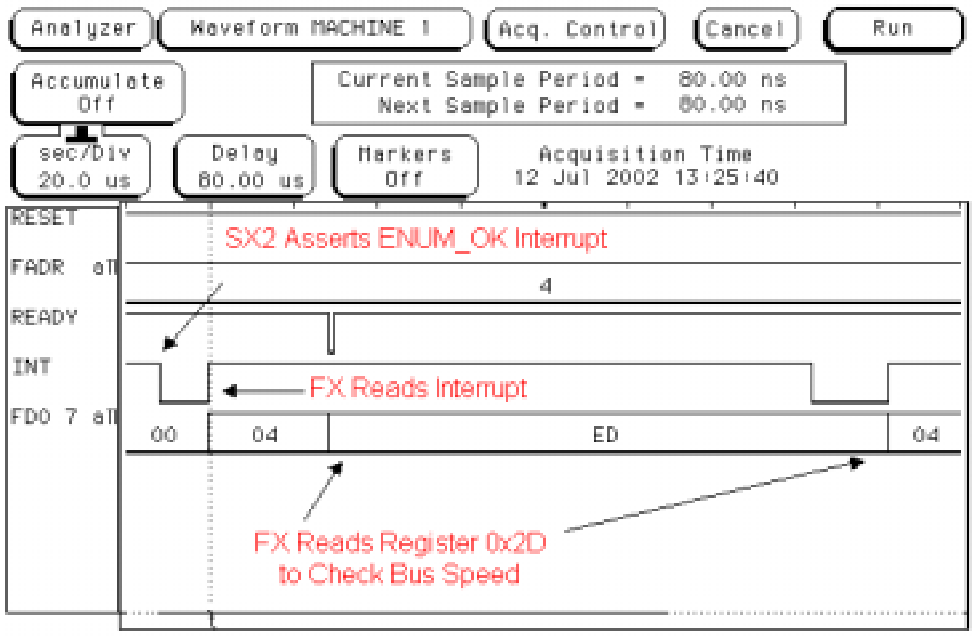

Analyzer
Waveform MACHINE 1
Acq. Control
Cancel
Run
Accumulate Off
Current Sample Period = 80.00 ns
Next Sample Period = 80.00 ns
sec/Div 20.0 us
Delay 80.00 us
Markers Off
Acquisition Time 12 Jul 2002 13:25:40
RESET
FADR
READY
INT
FD0 7
SX2 Asserts ENUM_OK Interrupt
4
FX Reads Interrupt
00
04
ED
04
FX Reads Register 0x2D to Check Bus Speed

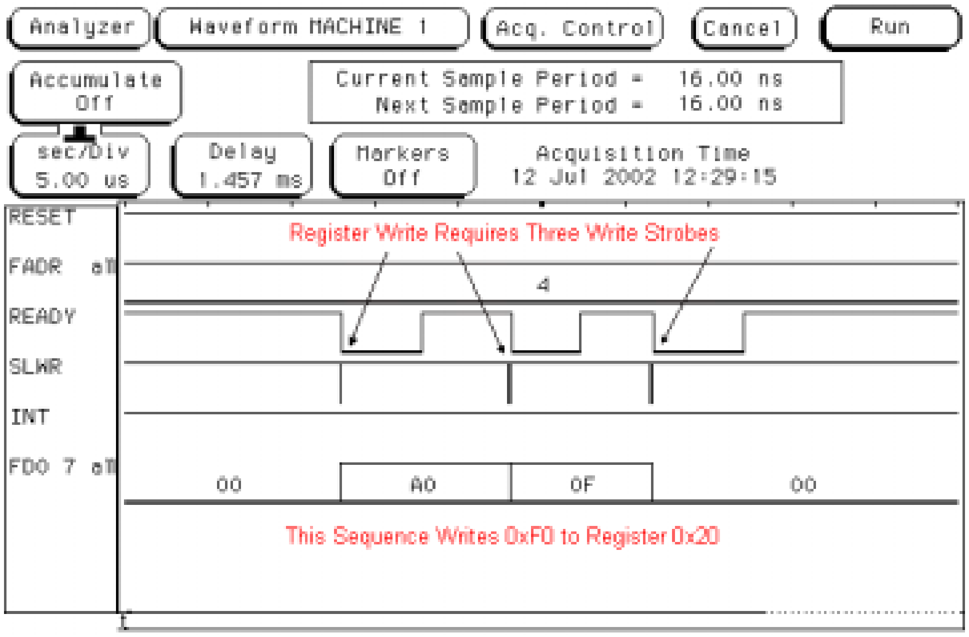
Figure 2. SX2 Initialization Continued

WriteRegister

The WriteRegister function calls the hardware specific function low_level_command_write. The first byte write is the 6-bit address of the register, with the most significant bit set and the next MSB cleared. The next write is the upper nibble of the data, with the four most significant bits cleared. The last write is the lower nibble of the data, with the four most significant bits cleared. The complete Write Register Function Sequence is shown in *Figure 3*.

WriteRegister Code

```
void WriteRegister (BYTE r, BYTE d)      //r = register number, d = data to be written
{
    low_level_command_write (0x04, (r | 0x80));         //Write request, bit7 = 1, bit6 = 0
    low_level_command_write (0x04, (d & 0xF0) >> 4);    //Write data high nibble
    low_level_command_write (0x04, (d & 0x0F));         //Write data low nibble
}
```



Current Sample Period = 16.00 ns
Next Sample Period = 16.00 ns

Acquisition Time
12 Jul 2002 12:29:15

Register Write Requires Three Write Strobes

This Sequence Writes 0xF0 to Register 0x20

Figure 3. Write Register Sequence

ReadRegister

The ReadRegister function calls the hardware specific function low_level_command_write and low_level_command _read. The first write is the address of the register, with the two most significant bits set. The FX waits until it receives an interrupt from the SX2 indicating that the data is ready. The FX then performs a hardware specific read and returns the data. The complete Read Register Sequence is shown in *Figure 4*.

ReadRegister Code

```
BYTE ReadRegister (BYTE r)                              //r = register number
{
    BYTE d;                                             //d holds read data
    read_interrupt = TRUE;
    low_level_command_write (0x04, (r | 0x80 | 0x40));  //Read request, bit7 = 1, bit6 = 1
    while (read_interrupt)                              //Wait until SX2 has data
           ;
    d = low_level_command_read (0x04);                  //Read Data
    return (d);
}
```

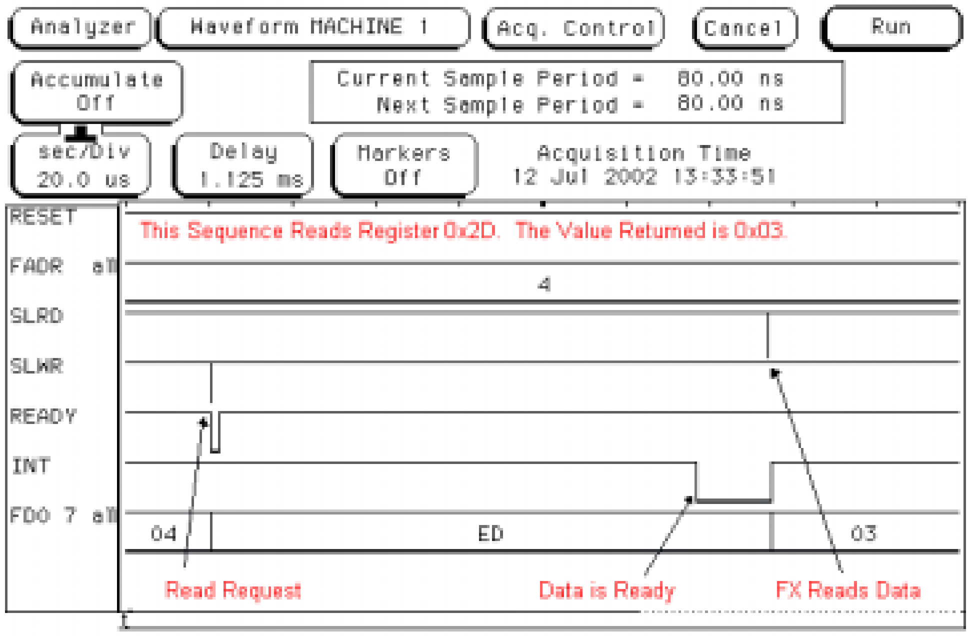


Figure 4. Read Register Sequence

Interrupt Service Routine

The Interrupt Service Routine distinguishes between the two types of interrupts (requested data ready or asynchronous interrupt) by examining the read_interrupt flag. If the ISR finds read_interrupt to be true, it clears the interrupt and does no further processing. This indicates to the ReadRegister function that the data is available to read.

If read_interrupt is FALSE, then this interrupt is an asynchronous interrupt that needs to be parsed. Depending on the source of the interrupt, some flags are set or toggled to tell the main loop what is happening.

The INTERRUPT pin on the SX2 is connected to a hardware interrupt source on the FX so that all SX2 interrupts are immediately serviced by the FX. Data may be immediately processed in the interrupt service routine or may be stored for background processing by the main loop.

Every time the INTERRUPT signal is asserted by the SX2, data is ready to be read, and the FX calls the low_level_command_read function . This is true whether it is requested data or an interrupt source.

ISR Code

```
void int0_isr (void) interrupt 0
{
    BYTE i;

    if (read_interrupt)                   //If we are expecting data
    {
            read_interrupt = FALSE;       //Clear flag
            return;                       //Don't go any further
    }

    i = low_level_command_read (0x04);
    switch (i)
    {
            case 0x01:                    //Ready
                    sx2_ready = TRUE;
                    break;
            case 0x02:                    //Bus Activity
                    no_activity = !no_activity;
                    break;
            case 0x04:                  //Enumeration complete
                    enum_ok = TRUE;
                    break;
            case 0x20:                  //Flags
                    got_out_data = TRUE;
                    break;
            case 0x40:                  //EP0Buf
                    ep0buf_ready = TRUE;
                    break;
            case 0x80:                  //Setup
                    got_setup = TRUE;
                    break;
        }
}
```

WriteDescriptor

The WriteDescriptor function uses the hardware specific function low_level_command_write to write the descriptor data to the SX2's 500 bytes of descriptor RAM. To load the descriptor, the WriteDescriptor function does the following:

1. Initiate a Command Address Transfer to register 0x30. This Command Address Transfer must conform to the Command Protocol specified in the SX2 Data Sheet–bit seven of the byte set to one, bit six set to zero, and the remaining bits zero through five indicating the address of the descriptor register (i.e., 0x30).
2. Perform four Command Data Transfers representing the LSB and MSB of the word value that defines the length of the entire descriptor about to be transferred. These Command Data Transfers and all other data transfers must conform to the Command Protocol–bit seven of each byte set to zero, bits four through six are don't cares, and bits zero through three are the lower or upper nibble of the byte being transferred.
3. Write the descriptor one byte at a time (perform two Command Data Transfers at a time), until complete.

   Note: the Command Address Transfer is only performed once.

WriteDescriptor Code

```
void WriteDescriptor (void)
{
    WORD len,i;

    len = sizeof(descriptor);

    low_level_command_write (0x04, (0x30 | 0x80));       //Write request, bit7 = 1, bit6 = 0
    low_level_command_write (0x04, (len & 0x00F0) >> 4); //Write length high nibble of lsb
    low_level_command_write (0x04, (len & 0x000F));      //Write length low nibble of lsb
    low_level_command_write (0x04, (len & 0xF000) >> 12);//Write length high nibble of msb
    low_level_command_write (0x04, (len & 0x0F00) >> 8); //Write length low nibble of msb
    for (i = 0; i < len; i++)
    {
            low_level_command_write (0x04, (descriptor[i] & 0xF0) >> 4);//Write data high nibble
            low_level_command_write (0x04, (descriptor[i] & 0x0F));     //Write data low nibble
    }
}
```

Endpoint 0

This section of code demonstrates how to handle endpoint 0 USB traffic. If the SX2 receives a setup request that it cannot handle automatically, it fires a SETUP interrupt. The FX ISR sets the got_setup flag when it sees a setup interrupt. This flag is checked in the main loop. For more information on SETUP requests, consult the USB Specification.

After reading a SETUP interrupt, the FX clears the flag and reads the eight bytes of setup data. For more information on the format of the setup data, consult the USB specification.

After receiving the setup data, the FX determines the direction, length, and type of request--standard, class, vendor, or unknown. Not all applications use every request type. This example acknowledges the vendor request 0xAA if it is a zero-length request. This vendor request could be used by the host application to indicate application specific status and is illustrated in *Figure 5*.

The example also responds to the vendor request 0xAB. This command can have a data stage. If the size is less than 64 bytes, the FX example loops back the data it receives.

The example also uses vendor requests 0xB6 and 0xB8 to signify a short packet, as described in the Data Loopback section.

All other requests are stalled. To stall a request, the external master initiates a write request for the SETUP register, 0x32, and writes any non-zero value to the register.

To complete endpoint zero data transfers, the ep0buf_ready flag is used. If the SX2 receives a setup request with a non-zero length, it fires the EP0BUF interrupt. For an IN request, this interrupt indicates that the EP0 buffer is available to be written to. For an OUT request, this interrupt indicates that a packet was transferred from the host to the SX2.

The FX firmware first clears the ep0buf_ready flag that was set in the ISR. If it's an IN request, the FX firmware writes data to the ep0buffer, then writes the byte count to the bytecount register. If it's an OUT request, the FX firmware reads the bytecount register to determine how much data to read, then reads the ep0buffer. This example only handles a maximum transfer of 64 bytes, but could be repeated for larger transfers. The SETUP and EP0 interrupts are illustrated in *Figure 6*.

Setup Code

```
if (got_setup)                                                  //Received setup interrupt
{
    got_setup = FALSE;                                          //Clear flag
    for (i = 0; i < 8; i++)
    {
           setup[i] = ReadRegister(0x32);                       //Read setup data
    }
    setupdirection = setup[0] & 0x80;                   //Find direction, In = 1, Out = 0
    setuplength = setup[6];                                     //Get length of setup
    setuplength |= setup[7] << 8;
    if ((setup[0] & 0x60) == 0)                                 //This is a standard request
    {
           //********TODO:  Handle or keep track of any standard requests
           switch (setup[1])
           {
                   case 0x01:                                   // *** Clear Feature
                          switch(setup[0])
                          {
                                  case 0x02:                    // End Point
                                         if(setup[2] == 0)
                                         {
                                                 switch(setup[4] & 0x7F)
                                                 {
                                                        case 2:
                                                                WriteRegister (0x06, 0xA2);
                                                                //Clear stall bit in EPxCFG
                                                                WriteRegister (0x33, 0);
                                                                //Ack this request
                                                                break;
                                                        case 4:
                                                                WriteRegister (0x07, 0xA0);
                                                                //Clear stall bit in EPxCFG
                                                                WriteRegister (0x33, 0);
                                                                //Ack this request
                                                                break;
                                                        case 6:
                                                                WriteRegister (0x08, 0xE2);
                                                                //Clear stall bit in EPxCFG
                                                                WriteRegister (0x33, 0);
                                                                //Ack this request
                                                                break;
                                                        case 8:
                                                                WriteRegister (0x09, 0xE0);
                                                                //Clear stall bit in EPxCFG
                                                                WriteRegister (0x33, 0);
                                                                //Ack this request
                                                                break;
                                                        default:
                                                                WriteRegister (0x32, 0xFF);
                                                                //Stall the request
                                                 }
                                         }
```

```
                                        else
                                                WriteRegister (0x32, 0xFF);     //Stall the request
                                        break;
                        }
                        break;
                case 0x03:                                                      // *** Set Feature
                        switch(setup[0])
                        {
                                case 0x02:                                      // End Point
                                        if(setup[2] == 0)
                                        {
                                                switch(setup[4] & 0x7F)
                                                {
                                                        case 2:
                                                                WriteRegister (0x06, 0xA6);
                                                                //Set stall bit in EPxCFG
                                                                WriteRegister (0x33, 0);
                                                                //Ack this request
                                                                break;
                                                        case 4:
                                                                WriteRegister (0x07, 0xA4);
                                                                //Set stall bit in EPxCFG
                                                                WriteRegister (0x33, 0);
                                                                //Ack this request
                                                                break;
                                                        case 6:
                                                                WriteRegister (0x08, 0xE6);
                                                                //Set stall bit in EPxCFG
                                                                WriteRegister (0x33, 0);
                                                                //Ack this request
                                                                break;
                                                        case 8:
                                                                WriteRegister (0x09, 0xE4);
                                                                //Set stall bit in EPxCFG
                                                                WriteRegister (0x33, 0);
                                                                //Ack this request
                                                                break;
                                                        default:
                                                                WriteRegister (0x32, 0xFF);
                                                                //Stall the request
                                                }
                                        }
                                        else
                                                WriteRegister (0x32, 0xFF);     //Stall the request
                                        break;
                        }
                        break;
        }
}
else if ((setup[0] & 0x60) == 0x20)                     //This is a class request
{
        //********TODO:  Handle or keep track of any class requests
}
else if ((setup[0] & 0x60) == 0x40)                     //This is a vendor request
{
        switch (setup[1])
        {
                //**********TODO:  Add specific cases that you handle
                case 0xAA:                              //We handle this vendor command if zero length
                        if (!setuplength)
                                WriteRegister (0x33, 0);        //Ack this request
                        else
                                WriteRegister (0x32, 0xFF);     //Stall the request
                        break;
```

```
                   case 0xAB:
                          if (setuplength > 64)                 //We want to handle vendor 0xAB
                                  WriteRegister (0x32, 0xFF);   //But only if less than 64
                          else
                          {
                                  while (!ep0buf_ready);        //Wait for buffer to become available
                                  ep0buf_ready = FALSE;         //Clear the flag
                   //This example loops back any data that we receive with a 0xAB vendor request
                   //Note: this example only handles 64 bytes
                                  if (setupdirection)           //In request
                                  {
                                         for (i = 0; i < setuplength; i++)
                                         {
                                                 WriteRegister( 0x31, setupdata[i] );
                                                 //Write data to buffer
                                         }
                                         WriteRegister (0x33, len);            //Write bytecount
                   //Note:  This routine can be modified for multiple packets of 64
                                  }
                                  else                                         //Out request
                                  {
                                         len = ReadRegister (0x33);            //Read the bytecount
                                         for (i = 0; i < len; i++)
                                         {
                                                 setupdata[i] = ReadRegister (0x31);
                                         }
                   //Note:  This routine can be modified for multiple packets of 64
                                  }
                          }
                          break;
                   case 0xB6:
                          ep6shortpacket = TRUE;
                          //Set a flag so that the main loop commits what data it has
                          WriteRegister (0x33, 0);                             //Ack this request
                          break;
                   case 0xB8:
                          ep8shortpacket = TRUE;
                          //Set a flag so that the main loop commits what data it has
                          WriteRegister (0x33, 0);                             //Ack this request
                          break;
                   default:                                     //We don't recognize the request
                          WriteRegister (0x32, 0xFF);                          //Stall the request
                          break;
           }
    }
    else                                                        //Reserved or undefined request
           WriteRegister (0x32, 0xFF);                          //Stall the request
}
```

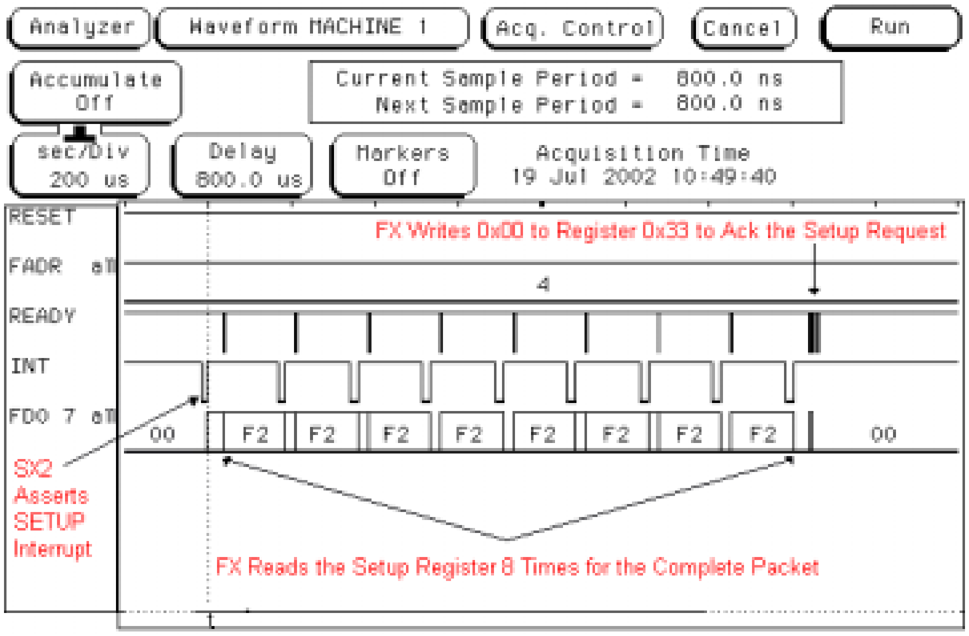

Analyzer
Waveform MACHINE 1
Acq. Control
Cancel
Run
Accumulate Off
Current Sample Period = 800.0 ns
Next Sample Period = 800.0 ns
sec/Div 200 us
Delay 800.0 us
Markers Off
Acquisition Time 19 Jul 2002 10:49:40
RESET
FADR all
READY
INT
FD0 7 all
FX Writes 0x00 to Register 0x33 to Ack the Setup Request
4
00
F2
00
SX2 Asserts SETUP Interrupt
FX Reads the Setup Register 8 Times for the Complete Packet

Figure 5. Setup Interrupt and Ack

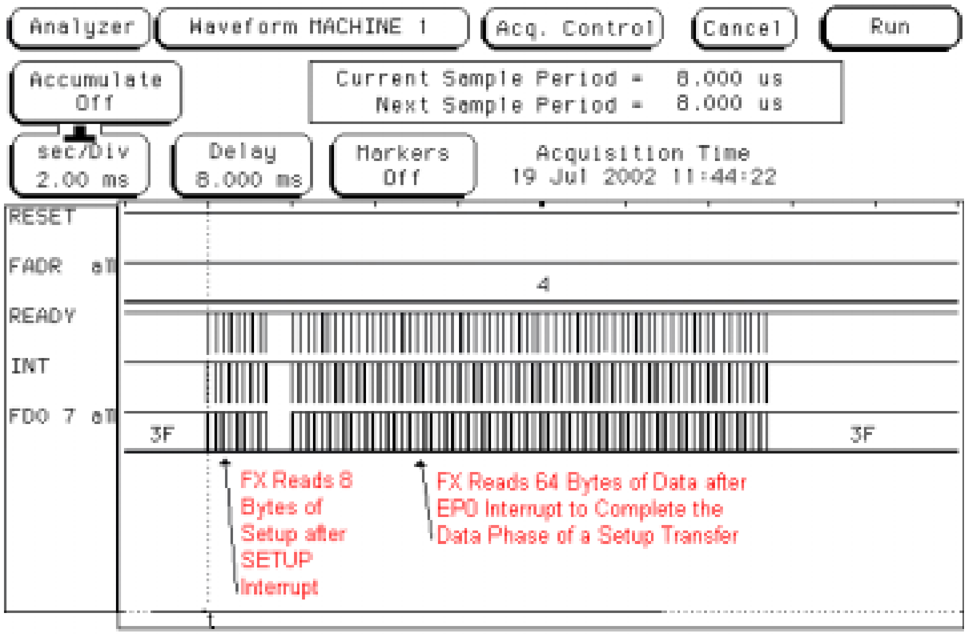

Analyzer
Waveform MACHINE 1
Acq. Control
Cancel
Run
Accumulate Off
Current Sample Period = 8.000 us
Next Sample Period = 8.000 us
sec/Div 2.00 ms
Delay 8.000 ms
Markers Off
Acquisition Time 19 Jul 2002 11:44:22
RESET
FADR all
READY
INT
FD0 7 all
4
3F
3F
FX Reads 8 Bytes of Setup after SETUP Interrupt
FX Reads 64 Bytes of Data after EP0 Interrupt to Complete the Data Phase of a Setup Transfer

Figure 6. Setup and EP0 Interrupts and Data

Data Loopback

If there is bus activity, then the FX firmware checks to see if the SX2 asserted the FLAGS interrupt. This indicates that the host sent data to one of the OUT endpoints, EP2 or EP4. If there is OUT data available, the FX firmware reads the EP24FLAGS register. This register checks the empty status of EP2 and EP4. If one of the endpoints contains data, the FX firmware reads data out of the endpoint, one byte at a time, and subsequently writes the bytes into one of the endpoints used for USB IN data. EP2 data is looped into EP6, and EP4 data is looped into EP8. The FX firmware continues to read and write data until the endpoint becomes empty.

This is a very simple data loopback example and can be exercised using the EZ-USB Control Panel PC program. Bulk Transfers of data can be sent OUT on EP2 or EP4 and then requested IN on EP6 or EP8.

The FX firmware also has an example of how to send an IN data packet which is less than the configured IN packet size. To do this, use the vendor request 0xB6 or 0xB8 which will write to the INPKTEND register, thereby committing any data already in EP6 or EP8 to the USB, regardless of the configured IN packet size.

This can also be exercised using the EZ-USB Control Panel program. First, send a small amount of data to EP2 (e.g., 30 bytes). Then, request a larger amount of data from EP6 (e.g., 64 bytes). The SX2 will not send the 30 bytes because the configured IN packet size has not been received. To send the 30 bytes, use the Control Panel to send the vendor request 0xB6 to the SX2. The FX firmware will see this request and commit the short packet.

The FX firmware checks the no_activity flag in its main loop. This flag is toggled in the ISR. If TRUE, the device has either been unplugged (self-powered), or suspended (bus-powered). If the device is bus-powered, then the FX firmware should put the SX2, then the FX into a low-power mode. Another BUSACTIVITY interrupt will wake up the FX. If the device supports remote wakeup, then the FX can wakeup the SX2 through a general purpose I/O pin instead of waiting for the host to resume.

Data Loopback Code

```
if (!no_activity)                               //If we are not suspended or unplugged
{
        ledX_rdvar = LED3_ON;
        if (got_out_data)                       //The FLAGS int tells us we have out data
        {
                got_out_data = FALSE;
                temp =  ReadRegister (0x1E);    //Read EP24 Flags Register
                if (!(temp & 0x02))             //If EP2 is NOT empty (has data)
                {
                        do
                        {
                                if (low_level_data_read (0x00, &dataloopback))
                                //If there is data to read from FIFO2
                                        while (!low_level_data_write (0x02, dataloopback));
                                        //Loop it back into FIFO6
                                temp =  ReadRegister (0x1E);  //Read EP24 Flags Register
                        }
                        while (!(temp & 0x02));
                        //Keep reading data out of EP2 until it is empty
                }
                if (!(temp & 0x20))             //If EP4 is NOT empty (has data)
                {
                        do
                        {
                                if (low_level_data_read (0x01, &dataloopback))
                                //If there is data to read from FIFO4
                                        while (!low_level_data_write (0x03, dataloopback));
                                        //Loop it back into FIFO8
                                temp =  ReadRegister (0x1E);  //Read EP24 Flags Register
                        }
                        while (!(temp & 0x20));
                        //Keep reading data out of EP4 until it is empty
                }
        }
        if (ep6shortpacket)
        //Sometimes we need to send an amount of data < the autoinlength
        {
                ep6shortpacket = FALSE;
                WriteRegister (0x20, 0x06);
                //Write EP6 packet end bit to INPKTEND register
                //Alternatively, the FX hardware could strobe the INPKTEND pin after setting
                //the address pins to endpoint 6
        }
```

```
                if (ep8shortpacket)
                //Sometimes we need to send an amount of data < the autoinlength
                {
                        ep8shortpacket = FALSE;
                        WriteRegister (0x20, 0x08);
                        //Write EP8 packet end bit to INPKTEND register
                        //Alternatively, the FX hardware could strobe the INPKTEND pin after setting
                        // the address pins to endpoint 8
                }
        }
        else            //If we are bus powered, then power down the SX2 and ourselves
        {
                ledX_rdvar = LED3_OFF;
//              temp = ReadRegister (0x01);             //Read the IFCONFIG register
//              WriteRegister (0x01, temp | 0x04);      //Set the SX2 standby bit
//              while (no_activity)
//                      {
//                              sleep();                //Stay asleep until the host resumes us
//                      }
                                //*********If the device supports remote wake-up, then it can
                                //wake up the SX2 instead of waiting for resume
        }
```

Conclusion

The SX2 is a powerful, intelligent peripheral chip that is easy to use and fits in a number of applications. This example shows standard initialization and data handling for a typical SX2 application. Most of the non-hardware-specific C code can be reused in other SX2 projects, making this example an excellent starting point for SX2 development.