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.

AN78446

Interrupt Handling in EZ-USB® FX2LP™

Author: Prajith Cheerakkoda

Associated Project: Yes

Associated Part Family: CY7C68013A/CY7C68014A/CY7C68015A/CY7V68016A

Software Version: None

Related Application Notes: None

More code examples? We heard you.

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

Are you looking for USB 3.0 peripheral controllers?

To access USB 3.0 product family, please visit our USB 3.0 Product Family webpage

AN78446 explains how three USB-specific interrupts and external interrupts are handled in EZ-USB® FX2LP™. It also provides example code for the interrupts.

Contents

1 Introduction 1
2 Endpoint Interrupts 2
3 IBN Interrupts 2
4 Ping NAK Interrupts 2
5 External Interrupts 3
5.1 ISRs 4
6 Firmware 4
6.1 Endpoint Interrupts 7
6.2 IBN Interrupts 8
6.3 Ping NAK Interrupts 9
6.4 External Interrupts 10
7 Hardware Connections 11
8 Testing the Project 11
9 Summary 13

1 Introduction

EZ-USB FX2LP incorporates 13 interrupt sources in its interrupt architecture: five standard 8051 interrupts and eight additional EZ-USB interrupts.

Standard 8051 interrupts:

- IE0(INT0): External Interrupt 0
- IE1(INT1): External Interrupt 1
- RI_0 and TI_0: UART 1 Interrupt
- TF0: Timer 0 Overflow
- TF1: Timer 1 Overflow

Additional EZ-USB interrupts:

- TF2: Timer 2 Overflow
- PF1: Wakeup Pin (WU2)
- RI_1 and TI_1: UART 1 Transmit and Receive
- USBINT(INT2): USB-Specific Interrupt
- I2CINT(INT3): I^2C Bus Interrupt
- IE4(INT4): External Interrupt 4
- IE5(INT5): External Interrupt 5
- IE6(INT6): External Interrupt 6

The USBINT (INT2) interrupt is shared among 27 USB-specific autovectored interrupts. EZ-USB FX2LP provides an advanced version of the vectored interrupts that are available, called “autovectored Interrupts.” Autovectoring is a mechanism employed in EZ-USB FX2LP to allow an interrupt service routine (ISR) to be automatically invoked when a corresponding interrupt occurs. For more details on autovectoring concepts and USB-specific interrupts, review the EZ-USB Technical Reference Manual (TRM), section 4.5, “USB-Interrupt Autovectors.”

The following USB-specific interrupts are covered in this application note:

- Endpoint interrupts
- In-Bulk-NAK (IBN) interrupts
- Ping NAK interrupts

EZ-USB FX2LP integrates three new external interrupts—INT4, INT5, and INT6—in addition to the standard 8051INT0 and INT1 external interrupts. This application note illustrates the use of the three USB-specific interrupts and all the external interrupts. It assumes that you are familiar with the EZ-USB TRM, chapter 4, Interrupts,” and that you have a basic understanding of 8051 interrupts.

2 Endpoint Interrupts

Table 1 shows the available endpoint interrupts with their priority and INT2VEC value. For an OUT endpoint, the interrupt request signifies that OUT data has been sent from the host, validated by the EZ-USB FX2LP, and is in the endpoint buffer memory.

For an IN endpoint, the interrupt request signifies that the data previously loaded by the EZ-USB into the IN endpoint buffer has been read and validated by the host, making the IN endpoint buffer ready to accept new data.

Table 1. EZ-USB Endpoint Interrupts

Interrupt Name	Priority	INT2VEC Value	Comment
EP0IN	9	20	EP0-IN is ready to be loaded with data.
EP0-OUT	10	24	EP0-OUT has USB data.
EP1IN	11	28	EP1-IN is ready to be loaded with data.
EP1-OUT	12	2C	EP1-OUT has USB data.
EP2	13	30	IN: Buffer is available. OUT: Buffer has data.
EP4	14	34	IN: Buffer is available. OUT: Buffer has data.
EP6	15	38	IN: Buffer is available. OUT: Buffer has data.
EP8	16	3C	IN: Buffer is available. OUT: Buffer has data.

3 IBN Interrupts

When the host requests an IN packet from the EZ-USB FX2LP Bulk endpoint, the endpoint NAKs (returns the NAK packet) until the endpoint buffer is filled and committed for transfer. At this point, the EZ-USB FX2LP responds to IN with a data packet.

Until the endpoint is committed, a flood of IN-NAKs can tie up bus bandwidth. Therefore, if the IN endpoints are not always kept full and armed, it is useful to know when the host is “knocking at the door,” requesting IN data. The IBN interrupt provides this notification. It fires whenever a Bulk endpoint NAKs an IN request.

4 Ping NAK Interrupts

When operating at full speed, every host OUT transfer consists of the OUT PID and the endpoint data, even if the endpoint is NAKing (not ready). While the endpoint is not ready, the host repeatedly sends all the OUT data; if it is repeatedly NAK’d, bus bandwidth is wasted. The USB 2.0 specification introduces a mechanism, PING, which makes better use of the bus bandwidth for “unready” Bulk OUT endpoints.

At high speed, the host can ping a Bulk OUT endpoint to determine if it is ready to accept data, holding off the OUT data transfer until it can be accepted. The host sends a PING token, and the EZ-USB FX2LP responds with either of the following:

- An ACK to indicate that there is space in the OUT endpoint buffer
- A NAK to indicate “not ready, try later.”

The PING interrupt indicates that an EZ-USB FX2LP Bulk OUT endpoint returned a NAK in response to a PING.

Note: PING applies only to high speed (480 Mbps).

Table 2 shows the PING interrupts available, with their priority and INT2VEC value. Interrupt enables for the individual interrupts are in the NAKIE register; interrupt requests are in the NAKIRQ register.

Table 2. EZ-USB PING Interrupts

Interrupt Name	Priority	INT2VEC value	Comment
EP0 PING	19	48	EP0 was pinged and it NAK’d.
EP1 PING	20	4C	EP1 was pinged and it NAK’d,
EP2 PING	21	50	EP2 was pinged and it NAK’d.
EP4 PING	22	54	EP4 was pinged and it NAK’d.
EP6 PING	23	58	EP6 was pinged and it NAK’d.
EP8 PING	24	5C	EP8 was pinged and it NAK’d.

5 External Interrupts

Figure 1 shows and Table 3 lists the external interrupts in the EZ-USB FX2LP chip.

Figure 1. External Interrupts

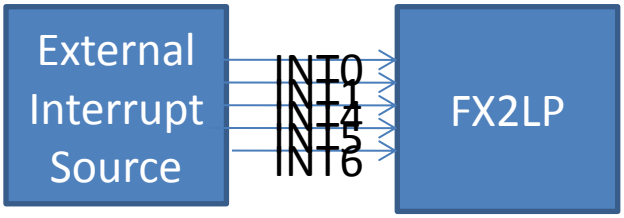


Table 3. External Interrupts

Interrupt	Interrupt Enable	Interrupt Pin	Priority Control	Natural Priority	Interrupt Request Flag	Interrupt Type	Interrupt Type Controlling Bit
INT0	IE.0	PA.0	IP.0	1	TCON.1	Level or edge sensitive, active low	[TCON.0]
INT1	IE.2	PA.1	IP.2	3	TCON.3	Level or edge sensitive, active low	[TCON.2]
INT4	EIE.0	See note 1	EIP.2	8	EXIF.4	Edge sensitive, active high	–
INT5	EIE.1	See note 1	EIP.3	9	EXIF.5	Edge sensitive, active low	–
INT6	EIE.4	PE.5	EIP.4	12	EICON.3	Edge sensitive, active high	–

Notes:

- The INT4 and INT5 have dedicated pins only in the 100 and 128 packages (CY7C68014A-128AXC, CY7C68013A-128AXI, CY7C68013A-100AXC, and CY7C68013A-100AXI). The pin for INT4 is shared among the GPIF/FIFO/INT4 interrupts; setting the INTSETUP register INTSETUP.1 to '0' enables INT4 operation. The default *USBJmpTb.a51* has an autovectoring option for INT4. To disable it, the following lines are commented in the interrupt vector table in the assembly source file *USBJmpTb.a51*:

```
CSEG    AT 53H

USB_Int4AutoVector    equ    $ + 2

ljmp    USB_Jump_Table
```

- IE, EIE, IP, EIP, TCON, EXIF, and EICON are special function registers (SFRs). For a description of these SFRs, refer to the EZ-USB TRM.
- Active low interrupts are falling-edge triggered, and active high interrupts are rising-edge triggered.

5.1 ISRs

In the example project, the ISRs for the external interrupts are defined in the C file *isr.c*. For example, the format to define the ISR for INT0 is as follows:

```
void ISR_EXTR0 (void) interrupt 0

{

//The code to execute within the ISR;

}
```

This function serves as the ISR for INT0. It must never be called by a C or assembly function. It is automatically executed when INT0 occurs.

"interrupt 0" tells the compiler to look for the ISR at address 0x0003. Similarly, for INT5, the ISR is as follows:

```
void ISR_EXTR5 (void) interrupt 11

{

// The code to execute within the ISR;

}
```

6 Firmware

The project available with this application note illustrates how to enable and handle these interrupts. The firmware is written in such a way that it performs a data loopback operation based on the USB-specific interrupts. The Bulk endpoint pairs EP1OUT-EP1IN, EP2OUT-EP6IN, and EP4OUT-EP8IN perform a data loop operation based on the endpoint interrupt, IBN interrupt, and Ping NAK interrupt, respectively. For external interrupts, the firmware toggles the DVK LEDs and Port C pins with half the frequency of the interrupt requests on the pins.

The project associated with this application note uses a bulkloop operation to demonstrate how the USB-specific interrupts mentioned previously work. Bulkloop is a data loopback operation using Bulk OUT and Bulk IN transfers on the EZ-USB FX2LP endpoints. During a Bulk OUT transfer, the host sends data to the OUT endpoint if it is empty. The firmware copies data from the OUT endpoint to the empty IN endpoint and then "commits" the IN endpoint buffer. During a Bulk IN transfer, the host reads data from the committed IN endpoint. In brief, a bulkloop operation reads data from the host and sends it back to the host.

IN-OUT endpoint pairs create data loopback paths for the bulkloop operation. In the example project, three loopback paths are available for the bulkloop operation, as shown in

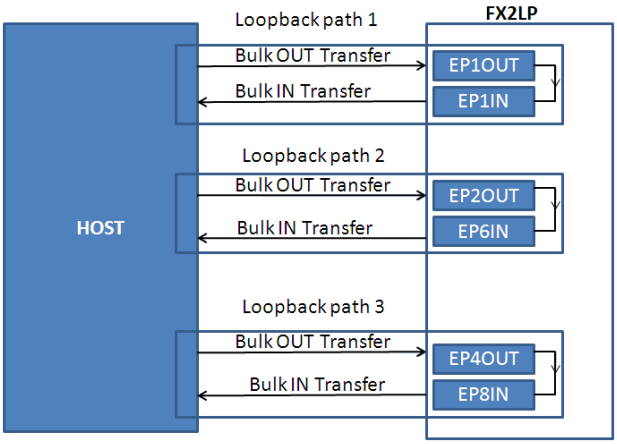
Figure 2, which means that the firmware has three IN-OUT endpoint pairs. The data flows through these paths based on USB-specific interrupts. Data transfer through path 1 is based on endpoint interrupts. Similarly, data transfer through path 2 and path 3 is based on the IBN and Ping NAK interrupts, respectively. Following are the steps included in the loopback operation:

1. Arm the OUT endpoint.
2. Copy data from the OUT endpoint to the IN endpoint.
3. Commit data in the IN endpoint.
4. Rearm the OUT endpoint.

Notes:

- Arming an OUT endpoint means making buffer space available for the Serial Interface Engine (SIE) to accept data from the host.
- Committing an IN endpoint makes the FIFO buffers available to the host for reading data from the endpoint.

Figure 2. External Interrupts



It is important in any USB ISR to clear the main USB interrupt before clearing the individual USB interrupt request latch. This is because as soon as the individual USB interrupt is cleared, any pending USB interrupt immediately tries to generate another main USB interrupt. If the main USB IRQ bit has not been previously cleared, the pending interrupt is lost. The structure for a typical USB ISR is as follows:

```
USB interrupt_ISR
{
; FIRST clear the USB (INT2) interrupt request
; Clear the USB interrupt request
; Service the interrupt here
}
```

The associated project is built on the framework available as part of the CY3684 EZ-USB FX2LP Development Kit (DVK) contents. The assembly source file *dscr.a51* defines three pairs of endpoints to the host for three data loop paths. Endpoints 2 and 4 are OUT endpoints. Endpoints 6 and 8 are IN endpoints. The EP1 OUT and EP1 IN endpoints are also defined. All endpoints are configured as Bulk.

The code execution begins from the main function in the file *fw.c*. The function TD_Init(),defined in the file *intr.c,* is called in the main function. This function initializes all the endpoints and enables the four interrupts mentioned previously.

The six endpoints defined in the assembly source file *dscr.a51* are configured in this function with the following statements:

```
EP1OUTCFG = 0xA0;
EP1INCFG = 0xA0;
SYNCDELAY;
EP2CFG = 0xA2;
SYNCDELAY;
EP4CFG = 0xA0;
```

```
SYNCDELAY;
EP6CFG = 0xE2;
SYNCDELAY;
EP8CFG = 0xE0;
SYNCDELAY;
```

Chapter 15 of the EZ-USB TRM contains the definition of the registers. The key characteristics of the endpoints are as follows:

- Endpoint 1 – IN, Bulk
- Endpoint 1 – OUT, Bulk
- Endpoint 2 – OUT, Bulk, double buffered
- Endpoint 4 – OUT, Bulk, double buffered
- Endpoint 6 – IN, Bulk, double buffered
- Endpoint 8 – IN, Bulk, double buffered

```
// out endpoints do not come up armed.  Arm EP1, EP2 and EP4 OUT endpoints
EP1OUTBC = 0x40;
// arm the EP1 OUT endpoint by writing to the byte count
// since the defaults are double buffered we must write dummy byte counts twice
SYNCDELAY;
EP2BCL = 0x80;
// arm EP2OUT by writing byte count w/skip.
SYNCDELAY;
EP2BCL = 0x80;
SYNCDELAY;
EP4BCL = 0x80;
// arm EP4OUT by writing byte count w/skip.
SYNCDELAY;
EP4BCL = 0x80;
```

The following two lines enable the interrupts of the EP1 endpoint defined for the project:

```
EPIE |= bmBIT3 ; // Enable EP1 OUT Endpoint interrupts
EPIE |= bmBIT2; // Enable  EP1 IN  Endpoint interrupts
```

The IBNIE register contains an individual interrupt-enable bit for each endpoint: EP0, EP1, EP2, EP4, EP6, and EP8. These bits are valid only if the endpoint is configured as a Bulk or Interrupt endpoint. The IBNIRQ register similarly contains individual interrupt request bits for the six endpoints. The EP6 IBN interrupt is enabled by setting the corresponding bit in the IBNIE register. The NAKIE/NAKIRQ registers each contain a single bit, IBN, which is the OR'd combination of the individual bits in IBNIE/IBNIRQ. For more information about these registers, see section 8.6.3.1 of the EZ-USB TRM. The firmware enables an interrupt by setting the enable bit high and clears an interrupt request bit by writing a '1' to it. The following code performs these functions:

```
// clear the global IBN IRQ
NAKIRQ = bmBIT0;
// enable the global IBN IRQ
NAKIE |= bmBIT0;
// clear any pending IBN IRQ
IBNIRQ = 0xFF;
```

```
// enable the IBN interrupt for EP6
IBNIE |= bmEP6IBN;
```

The following code in the TD_Init function clears any pending Ping NAK interrupts and enables the Ping NAK interrupt for EP4.

```
NAKIRQ |= ~bmIBN; // clear any pending PINGNAK IRQ

NAKIE |= bmEP4PING; // enable the PING-NAK interrupt for EP2 and EP4
```

In the associated project, the following register configurations are done in *intr.c* for setting up the external interrupts and Port C as the output port:

```
//INT0 and INT1
PORTACFG = 0x03;
// PA0 and PA1 are pins for INT0 and INT1 respectively.
TCON |= 0x05;
// INT0 and INT1 are configured as Edge triggered interrupts.
//INT4
INTSETUP &= ~0x02;   // If INTSETUP.1=0, then INT4 is supplied by the pin. Else, the
// interrupt is supplied internally  by FIFO/GPIF sources.
//INT5  is a dedicated  pin , available in the 100 and 128 pin packages.
//INT6
PORTECFG = 0x20; //PE5  is  INT6
OEE &= ~0x20;
//Enable External Interrupts
EIE |= 0x1C; // Enable
External Interrupts 4, 5 and 6
IE |= 0x05;//Enable External Interrupts 0 and 1
//Clear  Flags
EXIF &= 0xBF;  // Clear INT4 EXIF.6  Flag
EXIF &= 0x7F;  // Clear INT5 EXIF.7  Flag
EICON &= 0xF7; // Clear INT6 EICON.3  Flag
EA   = 1;     // Enable Global Interrupt
PORTCCFG = 0x00; // PORTC is is configured as an I/O, alternately
//it can output the lower address of enabled GPIF address pins
OEC  = 0xFF;        // PORTC is an output
IOC = 0xFF;  // Initialize PORTC to all LOW
```

The 8051 and USB-specific interrupts are handled differently. The USB interrupt vector generation is handled by the USB jump table (*USBJmpTb.obj*), and the 8051 interrupt vector generation is handled by the Keil compiler. The "#pragma noiv" statement in *intr.c* tells the Keil compiler to use the USB jump table to generate the interrupt vectors for the USB-specific interrupts, instead of its own interrupt vector scheme, which follows the traditional 8051 scheme.

6.1 Endpoint Interrupts

Endpoint ISRs ISR_Ep1in and ISR_Ep1out are associated with endpoints EP1IN and EP1OUT, respectively. The execution of these two ISRs enables the data flow through this path. The OUT endpoint ISR transfers data from EP1OUT to EP1IN, if it is available, and makes data inside EP1IN buffers available to the host. EP1IN ISR rearms the EP1OUT endpoint so that it can accept new data from the host. This cycle continues for new transfers from the host. Interrupt housekeeping, such as clearing the interrupt request, is also taken care of inside the ISRs.

The following code segment is the ISR that is serviced every time an EP1OUT transfer occurs. If endpoint EP1OUT has data (that is sent from the host), the capability of endpoint EP1IN to receive the data is checked. This is done by reading the EP1IN busy bit in the endpoint status register EP1INCS. If endpoint EP1IN is not full, then the data is transferred.

This decision is executed by the following statements:

```
if(!(EP1INCS & bmBIT1))
{// Checks EP1IN availability
```

The number of bytes to be transferred is read from the byte count register EP1OUTBC.The register's EP1OUTBC contains the number of bytes written into the FIFO buffer by the host.

```
count = EP1OUTBC;// The count value is loaded from the byte count register
```

Execution of the following loop carries out the data transfer:

```
for (i=0;i<count; i++)
{
      EP1INBUF[i]=EP1OUTBUF[i];
}
```

The following code commits the IN endpoint. It makes the buffer available to the host and clears the interrupt requests:

```
EP1INBC =count;
EZUSB_IRQ_CLEAR();//Clears the USB interrupt
EPIRQ = bmBIT3;//Clears EP1 OUT interrupt request
```

After the data is transferred, endpoint EP1OUT is rearmed to accept a new packet from the host. This is accomplished by the following code in the IN endpoint ISR:

```
EP1OUTBC = 64;
EZUSB_IRQ_CLEAR();//Clears the USB interrupt
EPIRQ = bmBIT2;//Clears EP1 IN interrupt request
```

Note: See the EZ-USB TRM, section 4.3.3, "Interrupt Latency," for information on the interrupt latency

6.2 IBN Interrupts

Loopback path 2, shown in

Figure 2, is based on an IBN interrupt. Unlike the endpoint interrupt, only one ISR is associated with the IBN interrupt. For example, there is one ISR for all Bulk IN endpoints. The endpoint that NAK'd is found using the IBNIRQ register. The IBNIRQ register contains individual request bits per endpoint. In this project, the IBN ISR executes whenever EP6 NAKs, as it is the only endpoint with IBN enabled.

The IBN ISR performs actions in the following order:

1. Clear the USB (INT2) interrupt request (by writing '0' to it) and disable the IBN interrupts for all endpoints so that the ISR execution is not interrupted in between.

   ```
   IBNIE = 0x00;
   // clear the global USB IRQ
   EZUSB_IRQ_CLEAR();
   ```

2. Inspect the endpoint bits in IBNIRQ to determine which IN endpoint just NAK'd. This step is relevant when IBN is enabled for more than one endpoint. Since IBN is enabled only for EP6IN, it is not required.
3. Take the required action and then clear the IBN bit in the IBNIRQ for the serviced endpoint (by writing '1' to it).

   The required action is data transfer. If endpoint 2 has data (that is sent from the host), it is checked by reading the endpoint 2 empty bit in the endpoint status register. This decision is executed by the following statements:

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

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

```
APTR1H = MSB( &EP2FIFOBUF );
// initializing the first data pointer
APTR1L = LSB( &EP2FIFOBUF );

AUTOPTRH2 = MSB( &EP6FIFOBUF );
// initializing the second data pointer
AUTOPTRL2 = LSB( &EP6FIFOBUF );

count = (EP2BCH << 8) + EP2BCL;
// The count value is loaded from the byte
count registers
```

Here the bulkloop operation is slightly different from the other two USB-specific interrupts. In ISR_Ibn, the first byte of received data is incremented and then written into the IN endpoint buffer. This is done with the following code:

```
EXTAUTODAT2 = EXTAUTODAT1+1;
```

The remaining data bytes are copied to the IN endpoint buffer without any modifications. Execution of the following loop carries out the data transfer:

```
for ( i = 0x0001; i < count; i++ )
   {
// setup to transfer EP2OUT buffer to EP6IN buffer using AUTOPOINTER(s)
       EXTAUTODAT2 = EXTAUTODAT1;
   }
```

The following statement transfers data from endpoint 2 to endpoint 6:

```
EXTAUTODAT2 = EXTAUTODAT1;
```

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

The following code commits the IN endpoint, that is, makes the buffer available to the host:

```
EP6BCH = EP2BCH;
SYNCDELAY;
EP6BCL = EP2BCL; // arm EP6IN

IBNIRQ = bmEP6IBN; // clear the IBN IRQ
IBNIE |= bmEP6IBN; // enable the IBN IRQ
SYNCDELAY;
```

After the data is transferred, endpoint 2 has to be rearmed to accept a new packet from the host.

```
EP2BCL = 0x80; // re(arm) EP2OUT
```

4. Clear the IBN bit in the NAKIRQ register (by writing '1' to it) and enable the IBN interrupt.

```
NAKIRQ = bmBIT0;// clear the global IBN IRQ

IBNIE = bmEP6IBN; // Enable IBN for EP6
```

6.3 Ping NAK Interrupts

EZ-USB FX2LP implements the Ping NAK interrupt as EP0PING, EP1PING, and so on, one for each endpoint. The EPxPING interrupt is asserted when the host PINGs an endpoint and the EZ-USB FX2LP responds with a NAK because the particular endpoint buffer memory is not available. The EZ-USB FX2LP firmware framework provides hooks for all the interrupts that it implements. The example project uses ISR_Ep4pingnak ISRs to handle the EP4PING interrupt. Following is the code for the EP4 ISR:

```
void ISR_Ep4pingnak(void) interrupt 0

{

SYNCDELAY; // Re-arm endpoint 4
```

```
EP4BCL = 0x80;
EZUSB_IRQ_CLEAR();
//clear the 4interrupt
NAKIRQ = bmEP4PING;
}
```

The ISR_Ep4pingnak discards the previous data that is stored in one of the buffers of endpoint 4 by rearming the endpoint (that is, EP4BCL = 0x80). Therefore, EP4 can now receive the data that is currently being sent by the host because there is space available in one of its buffers.

It then clears the interrupt by setting a particular bit in NAKIRQ because it has been serviced.

6.4 External Interrupts

In the associated project, Port C has been configured as the DVK LED D2 toggle. When an INT0 interrupt occurs, PC.0 and PC.1, D3/PC.4 and D4/PC.5, and D5/PC.6 are toggled. The code for toggling the LEDs and Port C pins is written inside the ISR of these interrupts.

In the example project, the ISRs for all the external interrupts are defined in *isr.c*. For example, the format to define the ISR for INT0 is as follows:

```
void ISR_EXTR0 (void) interrupt 0
{
//The code to execute within the ISR;
}
```

This function serves as the ISR for INT0. It must never be called by a C or assembly function. It is automatically executed when INT0 occurs.

“interrupt 0” tells the compiler to look for the ISR at address 0x0003. Similarly, for INT5, the ISR is as follows:

```
void ISR_EXTR5 (void) interrupt 11

{

// The code to execute within the ISR;

}
```

Example ISR code for INT4 in *isr.c* is as follows:

```
void ISR_EXTR4(void) interrupt 10

{
EXIF &= 0xBF;// Clear INT4 EXIF.6  Flag
IOC ^= 0x10;// Toggle pin 4 of PortC
}
```

7 Hardware Connections

Refer to Table 3 in the Getting Started with FX2LP™ document for the default jumper settings on the CY3684 DVK. Table 4 gives the port/jumper names on which the required pins for testing external interrupts are available on the CY3684.

Table 4. EZ-USB Jumper Instruction

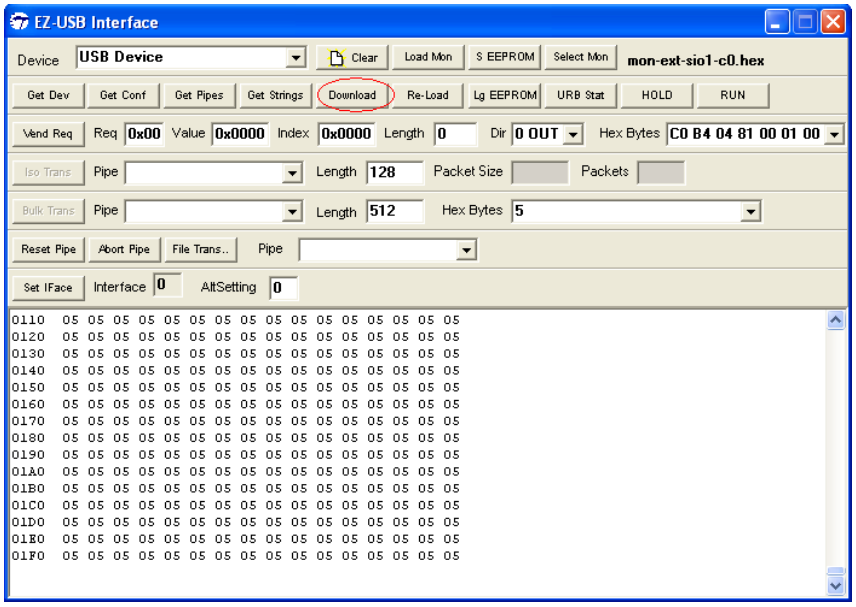
Pin Name	Port/Jumper Name on CY3684
Port C	P3
LEDs	JP3
INT0	P2.19
INT1	P2.18
INT4	P6.5
INT5	P6.4
INT6	PE.5

8 Testing the Project

This section describes how to test the functionality of three data loopback paths and the toggling of Port C pins and DVK LEDs based on external interrupts. To do the test, follow these steps:

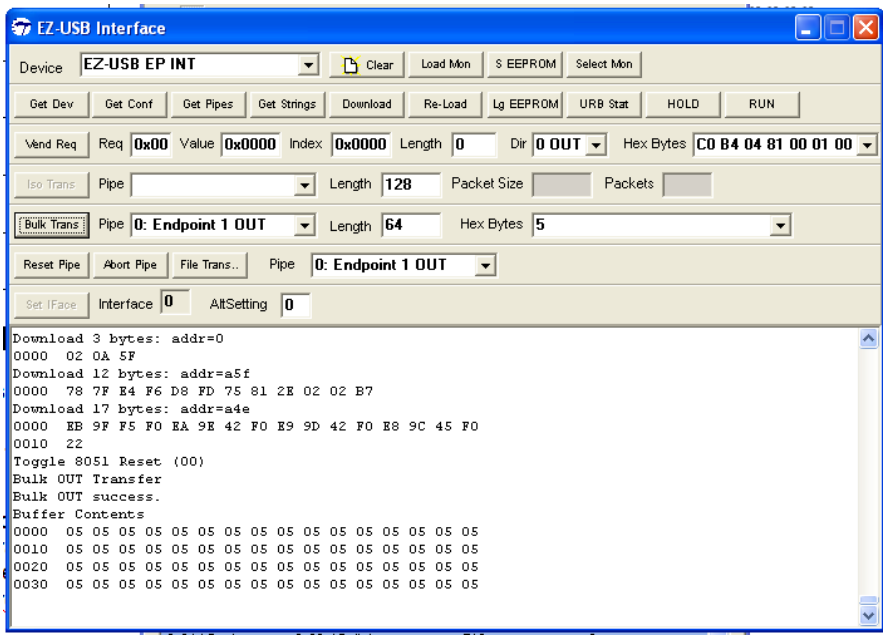
1. Download SuiteUSB 3.4 and install it. This installs the CyConsole utility.
2. Connect the CY3684 board to the PC with the EEPROM ENABLE switch in the “No EEPROM” position. The board enumerates with the default internal descriptor. Use the *CyUSB.inf* file available in the folder *Associated_project\driver* to bind with the device. For help with binding the driver, see *MatchingDriverToUSBDevice.htm* or the “Matching Devices to the Driver” section of the *CyUSB.pdf* file in the drivers folder.
3. Open the CyConsole utility. Go to **Start** > **All Programs** > **Cypress** > **USB** > **CyConsole EZ-USB**.
4. Download the compiled hex file *extr_intr.hex* to RAM by clicking the **Download** button, as shown in Figure 3, and selecting the path where the hex file is located. When the download is complete, it prompts for a driver.
5. Add new a VID/PID (the one used in the *dscr.asm* file, PID-04B4 VID-1004) to *CyUSB.inf* and then bind it with the device.

Figure 3. Endpoint_Interrupts.hex Download



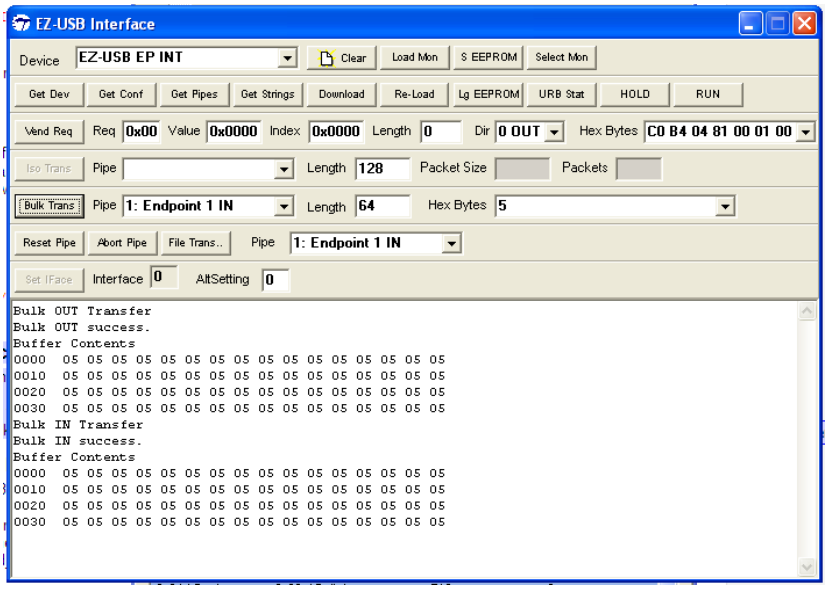
6. For testing data path 1, send 64 bytes of user-defined data from the host to endpoint EP1OUT using CyConsole. For example, select **0: Endpoint 1 OUT** in the **Pipe** field, specify the **Length** as 64 and **Hex Bytes** as 5, and then click the **Bulk Trans** button, as shown in Figure 4.

Figure 4. Bulk OUT Transfer



7. This data can be read back from endpoint EP1IN using CyConsole. For example, select **1: Endpoint 1 IN** in the **Pipe** field, specify the **Length** as 64, and then click the **Bulk Trans** button, as shown in Figure 5.

Figure 5. Bulk IN Transfer



The procedures for testing other interrupts are as follows:

- **IBN interrupts:** For OUT and IN data transfers through data loopback path 2, select endpoints EP4OUT and EP8IN respectively, instead of EP1OUT and EP1IN. These endpoints have a maximum packet size of 512 bytes. Do a data loopback as described previously for data path 1 and verify that the first byte of the received data has been incremented and the remaining bytes are the same.
- **Ping NAK interrupts**: In these interrupts, the endpoint EP4OUT-EP8IN endpoint pair with a maximum packet size of 512 bytes performs the data transfer. For OUT and IN data transfers, select endpoints EP4OUT and EP8IN, respectively. Do a data loopback and verify that the received and transmitted data are the same.

  The code can also be tested by continuously sending data to the EP4 without reading the data out of the EP8. Because the PING-NAK ISR rearms the endpoint, you can continuously transmit data to EP4, and the transfer always succeeds. The data in the EP4 buffers at any point of time is the latest two packets of data sent from the host.
- **External interrupts:** These types of interrupts trigger the interrupts using an external source, for example, a function generator. The function generator can be set to generate a square wave of known frequency (use low frequency, for example, a 100-Hz signal to view LED toggling). When the respective interrupts are triggered, the LED toggle appears. When an INT0 interrupt occurs, PC.0 and D2 are toggled. Similarly, on INT1/ INT4/ INT5/ INT6, PC.1 and D3/PC.4 and D4/PC.5 and D5/PC.6 are toggled. The Port C pin toggling can be checked by connecting those pins to the DSO.

9 Summary

This application note serves as a quick start guide for developing applications using EZ-USB FX2LP USB-specific and external interrupts. Example code demonstrates how three USB-specific interrupts and all external interrupts are handled. With the help of this document and the EZ-USB TRM, chapter 4, “Interrupts,” you can easily develop code for other USB-specific interrupts.

About the Author

Name: Prajith Cheerakkoda

Title: Applications Engineer

Document History

Document Title: AN78446 - Interrupt Handling in EZ-USB® FX2LP™

Document Number: 001-78446

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	3610040	PRJI	05/09/2012	New Spec.
*A	4803684	MDDD	06/19/2015	Updated template Sunset review
*B	5705323	AESATP12	04/26/2017	Updated logo and copyright
*C	5840394	MDDD	08/01/2017	Added More code examples section

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 | PSoC 6

Cypress Developer Community

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

Technical Support

cypress.com/support

All other trademarks or registered trademarks referenced herein are the property of their respective owners.

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

© Cypress Semiconductor Corporation, 2012-2017. 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.

.