Streaming Data Through Isochronous/Bulk Endpoints on EZ-USB *FX2™*

Introduction

Applications like audio and video that require continuous data flow without interruption use streaming-through-endpoints to transfer data over USB. In order to maintain the smoothness of flow and to minimize glitches in the application because of lost data, it is very important that the design be architected in such a manner that the data transfer rate meets what is required by the application. Failure of the device and host to maintain the required rate of data transfer may affect performance, adversely affecting the quality of images or sound in the case of video and audio applications, respectively.

This application note provides brief background information on what is involved while designing for a streaming application using the EZ-USB *FX2*™ part. It provides information on streaming data through bulk endpoints, isochronous endpoints and high-bandwidth isochronous endpoints along with pitfalls to consider and avoid while using the *FX2* for designing high-bandwidth applications. The application note also provides streaming example firmware that demonstrates how to setup the endpoint for streaming data through isochronous or bulk endpoints. This application note is included as a part of the CYStream Reference Design Package. The reference design also includes a host application and driver that enables streaming data to the *FX2* running the streaming example firmware.

Streaming Applications

Video devices (video compression engines, TV tuners, MPEG encoders), audio devices (MP3 players), and telephony devices are a few of the numerous applications that use streaming transfers. Although isochronous endpoints are used for designing streaming applications, bulk endpoints can also be used to design the same type of application. Depending on the data throughput and bandwidth required, the designer may opt for either type of endpoint.

Isochronous endpoints have a prenegotiated amount of USB bandwidth with a prenegotiated delivery latency. The pre-negotiated bandwidth is defined by setting the *wMaxPacketSize* field of the endpoint descriptor. The latency is defined by setting the *bInterval* field of the endpoint descriptor. This is discussed further in the *High-bandwidth Transfers* section of this application note. Isochronous endpoints have a guaranteed bandwidth but not guaranteed data delivery. The timely delivery of isochronous data is ensured at the expense of potential transient losses in the data stream. No retries of data transfer are ever done for isochronous data. This is the nature of data transfers involving isochronous endpoints. Isochronous endpoints can also be configured to design a high-bandwidth transfer application.

Bulk endpoints on the other hand, cannot be configured for high-bandwidth transfers. However, bulk endpoints can provide a greater maximum throughput than isochronous endpoints can with a trade off that the bandwidth is not guaranteed. Data is transferred through this type of endpoint as bandwidth is available. Bulk data traffic can become bursty when there is additional data traffic on the USB. Bulk endpoints use ACK/NAK hand shaking to ensure error-free data transfers.

The designer must make a decision what endpoint type to use based on the bandwidth and throughput requirements of their application. While designing any streaming application the designer must consider the following:

- Data throughput required on the peripheral end.
- Data throughput required on the USB end to meet the data rates required by the host application.
- Buffering requirements of the system.
- Endpoint type suitable to meet the required bandwidth and data rates.
- High-bandwidth requirement and how many packets per microframe required.
- Driver support for high-bandwidth transfers.
- Does the target OS for this application support high-bandwidth transfer.

It is the intent of the CYStream Reference Design Package to help the reader understand streaming applications in general and evaluate the reference design accordingly. This application note provides background USB related technical information on designing *FX2* firmware for a streaming application using bulk or isochronous endpoints by presenting the major highlights of the code example included in the reference design.

High-Bandwidth Transfers

High-bandwidth transfers are only defined for high-speed transfers using periodic endpoints: isochronous and interrupt. A high-speed endpoint that requires more than 1024 bytes per microframe is called a high-bandwidth endpoint. A high-bandwidth endpoint can transfer multiple packets, up to three packets of 1024 bytes each, per microframe. This yields a maximum transfer rate of 24 MBps. This application note covers information on high-bandwidth transfer through isochronous endpoints only.

The number of transfers per microframe is defined in the *wMaxPacketSize* field of the endpoint descriptor. Bits 12...11 of this field specify the number of additional transactions per microframe and can be set to the following:

- 00 = None (one transaction per microframe of 0—1024 bytes)
- 01 = 1 additional (two transfers per microframe, 0–2048 bytes per microframe)
- 10 = 2 additional (three transfers per microframe, 0–3072 bytes per microframe)
- 11 = Reserved.

Per the USB 2.0 specification, an isochronous endpoint must specify its required bus access period. This is done by setting the *bInterval* field of the endpoint descriptor*.* The *bInterval* field of the endpoint descriptor defines the rate at which the endpoint will be polled by the host. This provides a mechanism for slowing down the rate at which the host will service the endpoints.

The *bInterval* value is used as the exponent for a $2^{(bInterval-1)}$ value; for example, a *bInterval* of three means a period of four ($2^{(3-1)}$). For full and high speed isochronous endpoints, this *bInterval* value must be in the range from 1–16(inclusive). The desired period is specified as $2^{(bInterval-1)} \times F$ (frame/microframe) time. F is in units of 125 µs for high speed and 1 ms for full speed. This allows full- and high-speed isochronous transfers to have rates slower than one transaction per (micro)frame.

In the example described in this application note, the polling rate for all isochronous endpoint for each of the alternate setting is set to 1. This means that the host will poll this endpoint once ($2^{(1-1)}$) every microframe. Further details on this endpoint descriptor setting are presented in the *Streaming Firmware Example* section.

To keep track of the data packet transfer during the same microframe, high-bandwidth isochronous transfer uses a mechanism called the PID sequencing. While designing with high-bandwidth endpoints, it is very important that the design architecture take the data PID sequencing into consideration as described in the next section.

Data PID Sequencing

For synchronization of data packets during the same microframe a technique called data Packet ID (PID) Sequencing is used for all high-bandwidth transfers. This is similar to the data toggle synchronization used for bulk and interrupt endpoints, except that there are four different PID used: MDATA, DATA0, DATA1, DATA2. One of these PIDs is used for each of the data packet transferred during the same microframe.

For an IN transfer, the host will expect the number of packets per microframe specified in the device descriptor. However, the device can send less packets than what is specified in the device descriptor. To do this, the device must correctly set the data PID in the first IN packet to identify the number of packets the device can send during that microframe. The host reads the returned data PID to determine the number of additional IN tokens the host may send during the same microframe. The host must accept the data PID value returned from the device and not send more IN tokens than what the device can support during that microframe.

Consider a device descriptor that defines an IN endpoint with three transfers per microframe. For a device to send three packets of data per microframe, the device must respond to the first IN token from the host with a data PID of DATA2. The host, seeing a data PID of DATA2, knows that there are two more data packets available and to send two more IN tokens in the current microframe. If the device can only send two packets of data during the microframe, it must respond to the first IN token from the host with a data PID of DATA1. This will let the host know that there is one more packet left to receive from the device and to send only one more IN token during the current microframe. If the device has only one packet worth of data, it must respond to the first IN token from the host with the a data PID of DATA0. This tells the host that this is the last packet in the microframe and not to send any more IN tokens in the current microframe. If the device has no data at all, it may either send a Zero Length Packet (ZLP) with data PID of DATA0 or not send any data at all. The case of ZLP is described further in the section "*DATA PID Mismatch Consideration.*" This is basically how the PID sequencing mechanism works for isochronous IN transfers.

Data PID sequencing used for a high-speed, high-bandwidth isochronous OUT endpoint is different from what is used for an IN endpoint. For OUT transfers there is an additional data PID called the MDATA (More Data) to indicate that more data packets will be sent during the current microframe. The data PID of the last packet sent lets the device know how many packets were sent by the host during the current microframe. The host issues a DATA0 data packet when there is a single transaction. When there are two transactions per microframe, the host issues a data PID of MDATA for the first transaction and a data PID of DATA1 for the second transaction. The device, seeing a data PID other than MDATA, knows that it is the last packet in the current microframe. Seeing a data PID of DATA1, the device knows that there were two packets sent during the current microframe. When there are three transactions per microframe, the host uses a data PID of MDATA for the first two transactions and a DATA2 PID for the third and last transaction. Seeing a DATA2 PID, the device knows that this was the last packet and there were three packets sent during this current microframe. For an OUT transfer, the host must not send more data packets than what is specified in the endpoint descriptor within the same microframe.

Data PID sequencing allows isochronous endpoints to detect if a packet was lost/damaged during a microframe. Although, there is no recovery mechanism involved as this is basically the nature of isochronous transfers. The table below shows the order of data packet PIDs that are used in subsequent transactions within a microframe for high-bandwidth isochronous IN and OUT transfers.

Table 1. PID Sequencing for Isochronous Endpoints

Number of Packets Available	Direction	DATA PID		
		Packet 1	Packet 2	Packet 3
3	IN	DATA2	DATA1	DATA0
2	IN	DATA1	DATA0	–
1	IN	DATA0	–	–
3	OUT	MDATA	MDATA	DATA2
2	OUT	MDATA	DATA1	–
1	OUT	DATA0	–	–

The EZ-USB *FX2* isochronous endpoints supports data PID sequencing. While designing for IN transfers, the designer must make sure that the *FX2* device is able to supply the number of packets per microframe as specified in the endpoint descriptor and the EPxISOINPKTS register of the *FX2*. For an OUT transfer, the host application/driver should be designed to provide the number of packets per microframe as stated in the endpoint descriptor. For an IN transfer, if the device is unable to supply the number of packets per microframe as specified in the endpoint descriptor, there may be a situation where data PIDs go out of sync and result in data PID mismatch. Further information is provided on this scenario in the "*Data PID Mismatch Consideration*" section.

Set-up for High-Bandwidth Transfer on the *FX2*

To configure an IN endpoint for high-bandwidth isochronous transfers, the number of packets per microframe must be defined in the EPxISOINPKTS register of the *FX2*.

EPxISOINPKTS

b7	b6	b5	b4	b3	b2	b1	b0
0	0	0	0	0	0	INPPF1	INPPF0
R	R	R	R	R	R	R/W	R/W

Bit 1-0 INPPF1:0 *IN Packets per Frame*

The EPxISOINPKTS (× = endpoint number: 2, 4, 6 or 8) register is used to set the number of packets per microframe that can be sent through endpoint ×. Bits 0 and 1 determine the number of packets per microframe when the device is in high-speed mode. The designer must make sure that the *FX2* is able to provide this number of packets every microframe stated in this register. If the *FX2* falls short of the number of packets per microframe as stated in this register, it may run into a scenario of data PID mismatch.

No specific registers in the *FX2* need to be set for transfers through isochronous OUT endpoints. The device must report the number of packets desired per microframe in the endpoint descriptor.

Data PID Mismatch Consideration

For an IN transfer, when the host requests data, the device is expected to return the data with the correct data PID. As explained above, the data PID of the first packet received from the device lets the host know how many packets to expect in the same microframe. The *FX2* core does not look ahead at the number of packets committed to the USB. What does this mean? Well, if the EPxISOINPKTS is set to 3, the *FX2* will always send the first packet with a data PID of DATA2 regardless of how many packets has already been committed to the USB core. If the EPxISOINPKTS is set to 2, the *FX2* will always send the first packet with a data PID of DATA1 regardless of how many packets have already been committed to the USB core.

So if the EPxISOINPKTS is set to 3 and only two packets have been committed to the USB and is available to the core to send to the host, the *FX2* will respond to the first IN token with data PID DATA2. Seeing this first packet with DATA2 PID, the host (driver) will expect two more data packets and send two more additional IN tokens in the same microframe. The device will respond to the second IN token with data PID of DATA1. The device will not respond to the third IN token as there is no more data available (only two packets were committed to the core during the current microframe). This is a data PID mismatch error and will result in a USB Turnaround Error. It is up to the host to determine how it will handle and recover from this error condition.

The same problem will occur if EPxISOINPKTS is set to 2 and the *FX2* core has only one packet and the first packet is sent with DATA1 PID. If the EPxISOINPKTS is set to 1, of course there is no contention as the core will always start the packet with the DATA0 PID and the host will know not to expect any more packets in the current microframe.

Since the *FX2* core does not look ahead as to how many packets are available and sets the data PID of the first packet based on the setting of EPxISOINPKTS, it is bound to run into a scenario where it starts the first transaction with the wrong data PID. Starting with the incorrect PID causes bus turnaround/time out error. The user host driver must be designed in such a way as to be able to overcome this error.

There may also be a situation when there is no data available during a microframe. If the *FX2* does not respond with any data during a microframe and the host is expecting data, there might be some potential error on the host end application. The driver must be designed so as to handle this case. An alternate solution is to make sure that the *FX2* sends a zero length packet when there is no data available. There is no mechanism to allow the *FX2* to respond with a ZLP automatically when there are no data packets available to meet the number set in EPxISOINPKTS. So, in this case, there are two options:

1. Have some external logic commit a ZLP by asserting the PKTEND pin appropriately. Hence meeting the number of packets set in the EPxISOINPKTS register.
2. Have the external peripheral clock in filler data to keep stream running and have the host end driver filter out this filler data from the stream.

The host driver should be designed so that it is able to handle either of the cases of no data at all during a microframe or less number of transaction per microframe than what is stated in the endpoint descriptor. Cypress's new and improved driver for the development kit handles the issue when the number of packets received by the device is less than what is stated in the endpoint descriptor, by propagating the error up to the client application. This driver CYdvk.sys (binary format only) is also included with the CYStream Reference Design Package, which is discussed further in the next section.

CYStream Reference Design Package

The CYStream Reference Design Package is a software and example firmware that does the following:

1. Demonstrate isochronous and bulk streaming performance over USB with the EZ-USB *FX2* development kits.
2. Provide tools and sample firmware that can be used by developers interested in USB isochronous or streaming applications.

This package includes a host application that demonstrates isochronous and bulk streaming using the existing EZ-USB development kits, CY3681, which demonstrate the EZ-USB *FX2*'s throughput performance for:

- High-speed high-bandwidth isochronous streams.
- High-speed isochronous and bulk streams.
- Support for full-speed isochronous and bulk streams.

This reference design includes a host application (CYStream.exe) that uses the CYdvk.sys driver to interface with the EZ-USB *FX2* development board. The device has multiple alternate settings using isochronous IN/OUT endpoints, bulk IN/OUT endpoints, and high-bandwidth isochronous IN/OUT endpoints. The host application allows you to change to different alternate settings. The application also shows the average throughput during the transfer and the average CPU utilization.

The source code for the *FX2* device is provided in this reference design package and the major highlights are briefly discussed in the next section.

Streaming Firmware Example

The example code included with the CYStream Reference Design Package demonstrates how to set up endpoints (bulk and isochronous) of the *FX2* for streaming data. This section covers the major routines of the code in detail. The example includes the following source files:

1. The frameworks file, fw.c.
2. The descriptor file, dscr.a51.
3. The file CyStream.c, which has all the initialization for the the endpoint buffers and other configuration register. It also initializes the FIFO buffers with a specific data pattern.

The *FX2* endpoints FIFO can be configured to either manual mode (default state) or auto mode. In manual mode, the 8051 is responsible for committing data; where as in auto mode the data is committed automatically once the number of bytes in the FIFO meets the level set in a specific *FX2* register known as EPxAUTOINLEN. Refer to the *FX2* technical reference manual for further information on this register setting. For an OUT transfer, in order to commit data to the external peripheral, the 8051 needs to write to the byte count register with the SKIP bit set to 0. If the SKIP bit is set to 1, data received from the host is simply ignored (discarded). Writing to the byte count register with any arbitrary value re-arms the endpoint.

The example presented in this application note uses the manual mode setting of the endpoint FIFO, where the 8051 is responsible for committing data. There is no external master involved in this example. Section *CYStream.c* presents the code section that perform data transfers over USB through IN/OUT endpoint for various alternate settings.

Descriptors File: dscr.a51

This file has the USB standard descriptors defined in the appropriate order. There are seven alternate settings defined in the descriptor table for interface 1 when the device is operating in high-speed. The descriptor defines four alternate settings for full-speed operation. Each alternate setting configures the endpoints for either bulk or isochronous transfer. Two alternate settings configure isochronous endpoints for high-speed high-bandwidth transfer. This section also discusses the endpoint descriptor setting for bulk endpoint, isochronous endpoint and high-bandwidth isochronous endpoint. Please refer to the source code for details on the field setting of the endpoint descriptor for various alternate setting. Following is a sequential list of the descriptor as defined in the dscr.a51 source file:

- Device descriptor
- Device qualifier
- High-speed configuration descriptor
  - — Interface descriptor for alternate setting 0
    - Endpoint descriptors
  - — Interface descriptor for alternate setting 1
    - Endpoint descriptors
  - — Interface descriptor for alternate setting 2
    - Endpoint descriptors
  - — Interface descriptor for alternate setting 3
    - Endpoint descriptors
  - — Interface descriptor for alternate setting 4
    - Endpoint descriptors
  - — Interface descriptor for alternate setting 5
    - Endpoint descriptors
  - — Interface descriptor for alternate setting 6
    - Endpoint descriptors
- Full-speed configuration descriptor
  - — Interface descriptor for alternate setting 0
    - Endpoint descriptors
  - — Interface descriptor for alternate setting 1
    - Endpoint descriptors
  - — Interface descriptor for alternate setting 2
    - Endpoint descriptors
  - — Interface descriptor for alternate setting 3
    - Endpoint descriptors
- String descriptor

Table 2 displays the endpoint configuration for various alternate settings for full-speed operation and *Table 3* defines the endpoint configuration for various alternate settings for high-speed operation.

Table 2. Different Alternate Setting in Full-speed Mode

Alternate Setting	Number of End-points	Endpoint Number Direction (Type)	Maximum Packet Size (Bytes)
0	1	2 IN (Bulk)	64
1	1	2 OUT (Bulk)	64
2	1	2 IN (Isochronous)	1023
3	1	2 OUT (Isochronous)	1023

Table 3. Different Alternate Setting in High-speed Mode

Alternate Setting	Number of End-points	Endpoint Number Direction (Type)	Maximum Packet Size (Bytes)
0	1	2 IN (Bulk)	512
1	1	2 OUT (Bulk)	512
2	2	2 IN (Bulk)	512
		6 OUT (Bulk)	512
3	1	2 IN (Isochronous)	3x1024
4	1	2 OUT (Isochronous)	3x1024
5	1	2 IN (Isochronous)	1x1024
6	2	2 IN (Isochronous)	1x1024
		6 OUT (Isochronous)	1x1024

An endpoint descriptor is a seven byte descriptor that defines the endpoint characteristics. The first byte of this descriptor defines the length of the descriptor (0x07) which is followed by the descriptor type (0x05). Both of these are fixed fields regardless of the type of endpoint. Field byte 2 defines the endpoint number and direction. Bit 7 of byte 2 defines the direction of the endpoint and bits 0..6 define the endpoint number. Byte 3 defines the endpoint type: bulk = 0x02, isochronous = 0x01. Field 4 is a word (bytes 4 and 5) that defines the maximum packet size of the endpoint. The last field (byte 6) defines the polling interval of the endpoint. A detailed description of the byte fields of the endpoint descriptor for bulk endpoint, isochronous endpoint and high-bandwidth isochronous endpoint is presented below. For details on the bit definition of each field of an endpoint descriptor please refer to section 9.6.6 of the USB 2.0 specification.

Bulk Endpoint Descriptor

This is a seven-byte descriptor that defines the endpoint characteristics as follows:

07H ;; Descriptor length

05H ;; Descriptor type

82H ;; Endpoint number 2 and direction IN

02H ;; Endpoint type (Bulk)

00H ;; Maximum packet size (LSB)

02H ;; Max packet size (MSB) 512 byte packets/uFrame

00H ;;Polling interval.

The endpoint type field (byte 3) is set to 0x02 which defines a bulk endpoint type. Bytes four and five, which define the *wMaxPacketSize* field, are set to 512 bytes. The polling interval is not applicable to bulk endpoints and is arbitrarily set to 0.

Isochronous Endpoint Descriptor

This seven byte descriptor defines the endpoint characteristics as follows:

07H ;; Descriptor length

05H ;; Descriptor type

82H ;; Endpoint number2 and direction IN

01H ;; Endpoint type (Isochronous)

00H ;; Maximum packet size (LSB)

04H ;; Max packet size (MSB) 1 x 1024 byte packets/uFrame

01H ;;Polling interval.

The endpoint type is set to 0x01 for an isochronous endpoint. Bytes four and five, which define the *wMaxPacketSize* field, are set to 1024 bytes. As this is not a high-bandwidth endpoint bits 12..11 in the *wMaxPacketSize* field of the endpoint descriptor are set to 00 binary, indicating one packet per microframe. The *bInterval* field, which is the polling interval, is set to 1. This means that the host will poll this endpoint once($2^{(1-1)}$) every frame when the device is operating at high-speed.

High-bandwidth Isochronous Endpoint Descriptor

This is a seven byte descriptor that defines the endpoint characteristics as follows:

07H ;; Descriptor length

05H ;; Descriptor type

82H ;; Endpoint number 2 and direction IN

01H ;; Endpoint type

00H ;; Maximum packet size (LSB)

14H ;; Max packet size (MSB) 3 x 1024 byte packets/uFrame

01H ;;Polling interval.

The endpoint type field is set to 0x01 for an isochronous endpoint. Bytes four and five, the *wMaxPacketSize* field, are set to 0x0014. As explained in the *High-bandwidth Transfers* section, the number of transfers per microframe is defined in the *wMaxPacketSize* field of the endpoint descriptor. Bits 12..11, which specify the number of additional transactions per microframe, are set to 10 binary; indicating two additional packets per microframe. The *bInterval* field, which is the polling interval, is set to 1. This means that the host will poll this endpoint once ($2^{(1-1)}$) every frame when the device is operating at high speed.

For full-speed isochronous endpoints, the maximum packet size is limited to 1023 bytes and the endpoint cannot be configured for high-bandwidth transfer. The above examples of isochronous endpoints are defined for high-speed only. The polling rate is used in the same manner as used for a high-speed isochronous endpoint as discussed above.

Frameworks: fw.c

This section of the firmware responds to USB requests from the host, and implements enumeration and reenumeration. Refer to the frameworks application note provided in the development kit (which can also be downloaded from the Cypress Semiconductor web site at www.cypress.com) for more specifics on the source code.

Source code: CYStream.c

The following are a few of the major functions that are implemented in this source file. We will refrain from mentioning all the functions, so please refer to the source code for details.

TD_Init()

This routine is invoked from the main() once at the start. All the initialization of the endpoints are done in this routine.

The TD_Init routine initializes the endpoint FIFOs and the buffer with incrementing data bytes. The default alternate setting of 0 defines only one endpoint: 2 (IN), 512 bytes, quad buffered. So this endpoint is enabled in this routine by setting the VALID bit in the EP2CFG register. All the four buffers are initialized with data pattern of incrementing bytes. The endpoint FIFOs are set to Ports Mode, requiring data to be provided to the USB core via firmware intervention.

Note. A commercial application would either use the slave FIFO mode or the GPIF mode to transfer data between the host and the external peripheral device interfacing with the *FX2*. The mode can be set using the IFCONFIG register of the *FX2*. Further information on the IFCONFIG register setting can be found in section 15.5.2 of the *Technical Reference Manual*. This IFCONFIG register defaults to the Ports Mode. In “Ports” mode, all the I/O pins are general purpose I/O ports. “GPIF master” mode and “Slave FIFO mode” uses the PORTB and PORTD pins as a 16-bit data interface to the four *FX2* endpoint FIFOs EP2, EP4, EP6 and EP8.

```
void TD_Init(void)                                   // Called once at startup
{
   int i,j;

  CPUCS = ((CPUCS & ~bmCLKSPD) | bmCLKSPD1) ; // Set the CPU clock to 48MHz
  SYNCDELAY;

   IFCONFIG |= 0x40;                               // Set the slave FIFO interface to 48MHz
   SYNCDELAY;

    // Using endpoint 2 only (alternate setting 0), zero the valid bit on all others

    EP1OUTCFG = (EP1OUTCFG & 0x7F);
   SYNCDELAY;
   EP1INCFG = (EP1INCFG & 0x7F);
   SYNCDELAY;
   EP4CFG = (EP4CFG & 0x7F);
   SYNCDELAY;
   EP6CFG = (EP6CFG & 0x7F);
   SYNCDELAY;
   EP8CFG = (EP8CFG & 0x7F);
   SYNCDELAY;
    EP2CFG = 0xE0;                                 // DIR=IN, TYPE=BULK, SIZE=512, BUF=4

   USBIE |= bmSOF;                                 // Enable SOF interrupts
   mycount = 0;

    // Prepare data
    for (i=1;i<5;i++)
    {
         EP2FIFOBUF[0] = LSB(mycount);
         EP2FIFOBUF[1] = MSB(mycount);
         EP2FIFOBUF[2] = USBFRAMEL;
         EP2FIFOBUF[3] = USBFRAMEH;
         EP2FIFOBUF[4] = MICROFRAME;
         for (j=5;j<1024;j++)
         {
              EP2FIFOBUF[j] = i;
         }
         EP2BCH = 0x02;
         EP2BCL = 0x00;
    }
   Rwuen = TRUE;                                   // Enable remote-wakeup
}
```

Figure 1. TD_Init Routine

In this example there is no external peripheral so the mode is set to Ports mode and the endpoint FIFOs are left to their default manual mode in which the 8051 is responsible for (re)arming endpoints and committing data.

A variable called “mycount” is initialed to zero. This variable maintains the number of times the TD_Poll function is called in the forever while loop (in main), once the device has enumerated with an alternate setting that defines an IN endpoint. The first five bytes of each packet of EP2 is filled with the value of “mycount,” the current frame number and microframe number. *Figure 1* shows the code for the TD_Init routine.

TD_Poll()

The function TD_Poll is called repeatedly once the firmware has re-enumerated. This routine basically updates the first five bytes of each packet for an IN endpoint with the “mycount” value, the current frame number and the current microframe number. For IN transfer, once the endpoint is filled with the number of bytes, it is committed to the host by the 8051’s setting the byte count register to the number of bytes in the endpoint FIFO. The code is displayed in *Figure 2* below.

```
void TD_Poll(void)                                   // Called repeatedly while the device is idle
{
if( EZUSB_HIGHSPEED( ) )                            // FX2 in high-speed mode
{      switch (AlternateSetting)                    // Perform USB activity based on selected
   {                                                 //    alternate setting
      case Alt0_BulkIN:
            if(!(EP2468STAT & bmEP2FULL))
             {                                       // Send data on EP2
                 EP2FIFOBUF[0] = LSB(mycount);
                 EP2FIFOBUF[1] = MSB(mycount);
                 EP2FIFOBUF[2] = USBFRAMEL;
                 EP2FIFOBUF[3] = USBFRAMEH;
                 EP2FIFOBUF[4] = MICROFRAME;
                 EP2BCH = 0x02;                      // Arm endpoint with 512 Bytes
                 EP2BCL = 0x00;
                 mycount++;
             }
            break;

      case Alt2_BulkINOUT:
            if(!(EP2468STAT & bmEP2FULL))
           {                                         // Send data on EP2
                 EP2FIFOBUF[0] = LSB(mycount);
                 EP2FIFOBUF[1] = MSB(mycount);
                 EP2FIFOBUF[2] = USBFRAMEL;
                 EP2FIFOBUF[3] = USBFRAMEH;
                 EP2FIFOBUF[4] = MICROFRAME;
               EP2BCH = 0x02;                        // Arm endpoint with 512 bytes
                EP2BCL = 0x00;
                 mycount++;
             }
             // Check EP6 EMPTY(busy) bit in EP2468STAT (SFR),
            // Core sets this bit when FIFO is empty
             if(!(EP2468STAT & bmEP6EMPTY))
             {
                 EP6BCL = 0x80;                      // Re(arm) EP6OUT
             }
             break;

        case Alt3_IsocIN:
        case Alt5_IsocIN:
             if(!(EP2468STAT & bmEP2FULL))
            {                                        // Send data on EP2
                 EP2FIFOBUF[0] = LSB(mycount);
                 EP2FIFOBUF[1] = MSB(mycount);
                 EP2FIFOBUF[2] = USBFRAMEL;
                 EP2FIFOBUF[3] = USBFRAMEH;
                 EP2FIFOBUF[4] = MICROFRAME;
                   EP2BCH = 0x04;                    // Arm endpoint with 1024 Bytes
                  EP2BCL = 0x00;
                mycount++;
              }
              break;

        case Alt1_BulkOUT:
        case Alt4_IsocOUT:
             // Check EP2 EMPTY(busy) bit in EP2468STAT (SFR),
            // Core sets this bit when FIFO is empty
             if(!(EP2468STAT & bmEP2EMPTY))
             {
                 EP2BCL = 0x80;                      // Re(arm) EP2OUT
             }
             break;

        case Alt6_IsocINOUT:
            {
             if(!(EP2468STAT & bmEP2FULL))
            {                                        // Send data on EP2
                 EP2FIFOBUF[0] = LSB(mycount);
                 EP2FIFOBUF[1] = MSB(mycount);
                 EP2FIFOBUF[2] = USBFRAMEL;
                 EP2FIFOBUF[3] = USBFRAMEH;
                EP2FIFOBUF[4] = MICROFRAME;
                EP2BCH = 0x04;                       // Arm endpoint with 1024 Bytes
                  EP2BCL = 0x00;
                mycount++;
            }
/       //Check EP6 EMPTY(busy) bit in EP2468STAT (SFR),
        // Core sets this bit when FIFO is empty
         if(!(EP2468STAT & bmEP6EMPTY))
        {
              EP6BCL = 0x80;                    // Re(arm) EP6OUT
         }
         break;
    }
    break;
}
 lse                                          // FX2 is in Full Speed

    switch (AlternateSetting)               // Perform USB activity based on the
  {                                            //    selected alternate setting
      case Full_Alt0_BulkIN:
          if(!(EP2468STAT & bmEP2FULL))
          {                                    // Send data on EP2
             EP2FIFOBUF[0] = LSB(mycount);
             EP2FIFOBUF[1] = MSB(mycount);
             EP2FIFOBUF[2] = USBFRAMEL;
             EP2FIFOBUF[3] = USBFRAMEH;
             EP2FIFOBUF[4] = MICROFRAME;
            EP2BCH = 0x00;                     // Arm endpoint with 64 Bytes
             EP2BCL = 0x40;
           mycount++;
          }
          break;
      case Full_Alt1_BulkOUT:
      // Check EP2 EMPTY(busy) bit in EP2468STAT (SFR),
         // Core set's this bit when FIFO isempty
      if(!(EP2468STAT & bmEP2EMPTY))
      {
          EP2BCL = 0x80;                       // Re(arm) EP2OUT
      }
       break;

       case Full_Alt2_IsocIN:
        if(!(EP2468STAT & bmEP2FULL))
        {                                      // Send data on EP2
           EP2FIFOBUF[0] = LSB(mycount);
           EP2FIFOBUF[1] = MSB(mycount);
           EP2FIFOBUF[2] = USBFRAMEL;
           EP2FIFOBUF[3] = USBFRAMEH;
           EP2FIFOBUF[4] = MICROFRAME;

          EP2BCH = 0x03;                       // Arm endpoint with 1023 Bytes
           EP2BCL = 0xFF;
          mycount++;
                 }
       break;

   case Full_Alt3_IsocOUT:
      // Check EP2 EMPTY(busy) bit in EP2468STAT (SFR),
        // Core set's this bit when FIFO isempty
      if(!(EP2468STAT & bmEP2EMPTY))
      {
          EP2BCL = 0x80;                    // Re(arm) EP2OUT
      }
      break;
   }
```

Figure 2. TD_Poll Routine

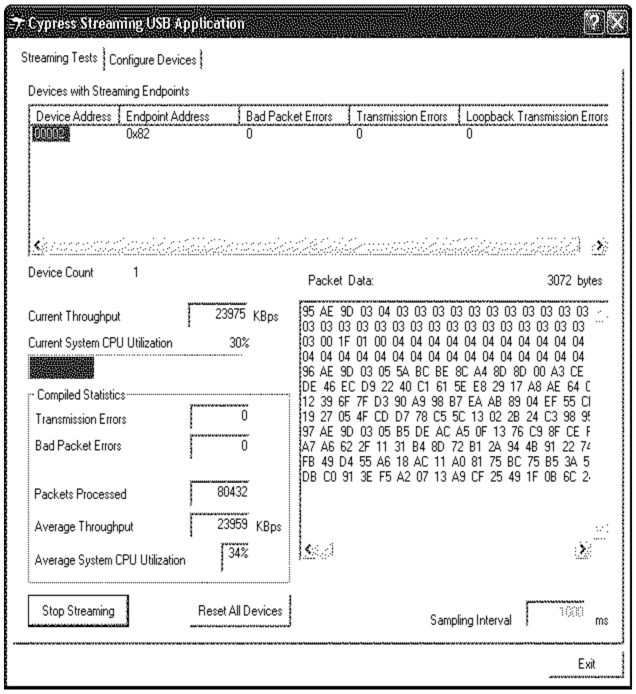
For an OUT transfer, in order to rearm the endpoint to allow the device to accept data from the host, the 8051 writes to the endpoint byte count register with any arbitrary value with the SKIP bit set. This is done after checking and making sure that the endpoint is not already busy receiving data from the host.

The soft copy of the entire code can be downloaded off the web. It is also included with the CYStream Reference Design Software Package.

Performance Analysis

The CYStream reference design can be used to evaluate the performance of a streaming application. The CYStream reference design software package includes a host application known as the CYStream.exe. This application is used to interface with the EZ-USB *FX2* running the CYStream firmware. In this section we will present information on the data throughput achieved using the CYStream firmware on the *FX2*. It is recommended that the reader refer to the *CYStream User's Guide* included in the reference design kit for information on how to set up the *FX2* development board to run the CYStream firmware. Once you have this set up, run the CYStream host application. Please note that the CYStream utility tool uses the CYDvk.sys driver to interface with the *FX2* board running the CYStream firmware. So you must have the device bound to the CYDvk.sys driver. The *CYStream User's Guide* provides step-by-step information on how to set up and run this application. The data throughput results will only be discussed here.

Following is a screen capture of the CYStream host application window while streaming data, followed by a description of the fields shown in the window displayed in *Figure 3*.



Devices with Streaming Endpoints

Device Address	Endpoint Address	Bad Packet Errors	Transmission Errors	Loopback Transmission Errors
00002	0x82	0	0	0

Device Count 1

Current Throughput 23975 KBps

Current System CPU Utilization 30%

Compiled Statistics

Transmission Errors 0

Bad Packet Errors 0

Packets Processed 80432

Average Throughput 23959 KBps

Average System CPU Utilization 34%

Sampling Interval 1000 ms

Packet Data: 3072 bytes

```
95 AE 9D 03 04 03 03 03 03 03 03 03 03 03 03 03
03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03
03 00 1F 01 00 04 04 04 04 04 04 04 04 04 04 04
04 04 04 04 04 04 04 04 04 04 04 04 04 04 04 04
96 AE 9D 03 05 5A BC BE 8C A4 8D 8D 00 A3 CE
DE 46 EC D9 22 40 C1 61 5E E8 29 17 A8 AE 64 C
12 39 6F 7F D3 90 A9 98 B7 EA AB 89 04 EF 55 C
19 27 05 4F CD D7 78 C5 5C 13 02 2B 24 C3 98 9
97 AE 9D 03 05 B5 DE AC A5 0F 13 76 C9 8F CE F
A7 A6 62 2F 11 31 B4 8D 72 B1 2A 94 4B 91 22 7
FB 49 D4 55 A6 18 AC 11 A0 81 75 BC 75 B5 3A 5
DB C0 91 3E F5 A2 07 13 A9 CF 25 49 1F 0B 6C 2
```

Figure 3. CYStream.exe Application Window Display

Current Throughput: Provides live updating of the current throughput performance of the USB bus and EZ-USB *FX2* over selected endpoints.

Current System CPU Utilization: Provides a visual indication of the utilization of the computer's CPU while streaming over USB.

Transmission Errors: Increments whenever there is an error reported in the transfer of a buffer.

Bad Packet Errors: Increments whenever there is an error with a specific packet of a transfer. Many times, both error fields will report the same error. Isochronous transfers may have bad packet errors that don't report as transmission errors since there is no CRC checking in isochronous transfers.

Packets Processed: Increments to show the total number of packets successfully transferred during the streaming test.

Average Throughput: Shows performance over the duration of the streaming test.

Average System CPU Utilization: Shows performance over the duration of the streaming test.

This application was run on a system with an Intel® USB 2.0 host controller running Windows® XP (SP1) with the current version of the Microsoft® driver (version: 5.1.2600). A CATC device was used to capture the data and determine the number of packets transferred per microframe for each of the alternate settings. *Table 4* shows the results obtained when the device is operating at high speed; *Table 5* shows the results obtained when the device is operating in full speed.

Table 4. Average Throughput at High Speed

Alternate Setting	Endpoint Number/Type	Average Throughput (CYStream) (MBps)	Number of Transfers per microframe (CATC)
0	2 IN (Bulk)	39.967	10
1	2 OUT (Bulk)	31.815	8
2	2 IN (Bulk)	28.100	6-7
	6 OUT (Bulk)	31.78	8
3	2 IN(ISO)	24.000	3
4	2 OUT (ISO)	23.975	3
5	2 IN (ISO)	8.000	1
6	2 IN (ISO)	7.997	1
	6 OUT (ISO)	8.000	1

Table 5. Average Throughput at Full-Speed

Alternate Setting	Endpoint Number/Type	Average Throughput (CYStream) (MBps)	Number of Transfers per microframe (from CATC)
0	2 IN (Bulk)	1.061	17
1	2 OUT (Bulk)	1.113	18
2	2 IN (ISO)	1.006	1
3	2 OUT (ISO)	1.006	1

Note that the numbers in column 3 are read from what is displayed by the CYStream host application. Due to some overhead on the host end (host application implementation) the numbers displayed may vary slightly from what is actually measured off the CATC.

The results show that the maximum throughput rate using isochronous endpoint is 24 MBps. The same rate can also be achieved using bulk endpoints when the endpoint is configured as a quad buffered endpoint, as illustrated by this example firmware.

Approved AN053 9/9/03 kkv