AN58764

Implementing a Virtual COM Port using FX2LP

Author: Prajith Cheerakkoda
Associated Project: Yes
Associated Part Family: CY7C68013/14/15/16
Software Version: None
Related Application Notes: AN65209

AN58764 explains how to implement a virtual COM port device using FX2LP. This eases migration from UART to USB. This document contains example code with the required descriptors, class-specific request handling, and the INF file required for enumeration.

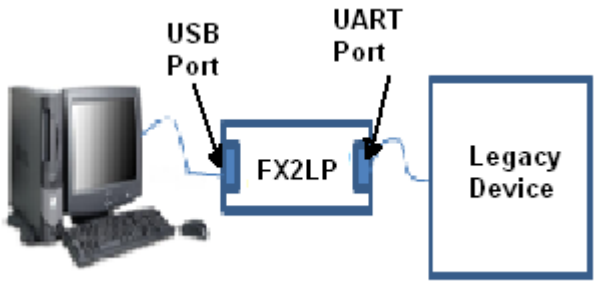
Contents

Introduction

The EZ-USB FX2LP™ is a high-speed USB peripheral controller. The programmability and flexibility of FX2LP allows easy implementation of USB device classes such as CDC (Communication Device Class), MSC (Mass Storage Class) and HID (human Interface Device). This application note discusses the implementation of virtual COM port device using FX2LP. This is done without any additional driver effort by using the default Windows serial driver (usbser.sys). Virtual COM port can coexist with other high speed interfaces and can be used for debug and manufacturing purpose.

Familiarity of Technical Reference Manual (TRM) chapter "14.Timers/Counter and Serial Interface" is assumed.

Figure 1. FX2LP as VirtualCOM Port.



Virtual COM Port

There are many PCs and laptops in the market that do not have a legacy COM port. The CDC device class of USB gives a way to fill the void of a COM port. Microsoft Windows natively comes with a serial driver *usbser.sys,* which does the UART to USB translations and vice versa. This driver allows legacy applications such as HyperTerminal to communicate with legacy devices.

Figure 2. PC with COM Port

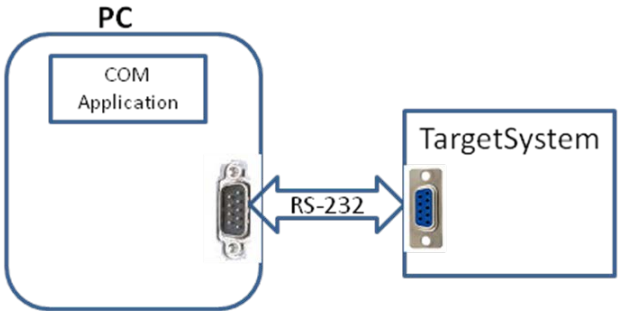
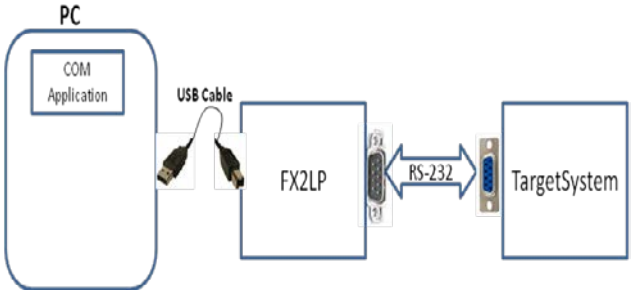


Figure 3. PC with Virtual COM Port



Communication Device Class Specification

USB communication device class (USB CDC), which defines architecture for emulating Telecommunication and Network Devices on USB ports, is a composite USB device class. Although it is a single device class, there may be more than one interface implemented such as a custom control interface, data interface, audio, or mass storage–related interfaces.

The CDC specifications are available through USB-IF website, http://www.usb.org/developers/devclass_docs/usbcdc11.pdf. This class specification supports five basic models of communication and each has one or more subclasses.

POTS: Devices that communicate through ordinary phone lines and generic COM-port devices.

ISDN: Communication through phone lines with ISDN interfaces

Networking model: Communication through Ethernet or ATM

Wireless mobile communication: Cell phones that support voice and data communication

Ethernet emulation model: An efficient way for devices to send and receive Ethernet frames

In this project for serial emulation we implement ACM (Abstract Control Model) subclass, which comes under plain old telephone system (POTS) model.

Abstract Control Model

The Abstract Control Model (ACM) can bridge the gap between legacy modem devices and USB devices. The ACM in the USB CDC Specification (Section 3.6.2.1) details the serial enumeration requirements, class specific requests, and serial notification required to establish a communication protocol.

Abstract Control Model requires two interfaces for communication devices.

Communication Class Interface: This is a device management interface for controlling operations of the device. In addition to standard USB requests this interface conveys class specific requests to the device. These requests are covered in Class Specific Request section.

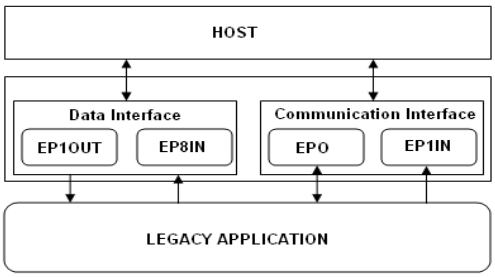
This interface has an optional notification component as well. It requires one interrupt IN endpoint to notify the host about the status of the control signals. Even though notification element is an optional component of communication interface it is required in standard PC.

Data Class interface: All data transfer between the host and the device is handled in this interface. It requires two BULK Endpoints, IN Endpoint for receiving data and an OUT Endpoint for sending data.

Data Transfer

In ACM subclass interfaces are implemented as in figure shown below.

Figure 4. CDC Interface



In the attached project, Host sends data from the legacy application to legacy device through EP1OUT bulk endpoint of FX2LP. Similarly data received from the legacy device is committed to EP8IN bulk Endpoint, Host polls the IN endpoint to receive data.

Device Management

Management component of communication interface accomplishes Device management through the default

control endpoint EP0. There are class specific requests for CDC device management.

Class Specific requests

To support certain types of legacy applications, two problems need to be addressed, supporting specific legacy control signals and state variables. Class-specific requests are defined to support these requirements

Set_line_coding: Sets asynchronous serial parameters: bit rate, number of stop bits, parity, and number of data bits.

Bit rate: This parameter sets the baud rate of communication i.e. number of bits transmitted per second.

Parity bit: This is used for error handling. Here we don't use parity so set it as "None" in Host application.

Number of stop bits: Bits following actual data bits.

Number of data bits: Number of bits transferred for each byte of data.

Get_line_coding: Requests asynchronous serial parameters: bit rate, number of stop bits, parity, and number of data bits. Host requests the present device configuration.

Set_control_line_state: Sets RS-232 signals, RTS, and DTR. This is optional but is required if these signals are used. Since these signals are not present in FX2LP, attached project doesn't use these signals.

The device returns notifications through interrupt IN endpoint. The host issues periodic IN tokens to the endpoint. The endpoint returns the notification if available and NAKs if it does not have any notification. In the firmware, SERIAL STATE notification sends the state of CD, DSR, Break, and RI. Sending this notification is optional.

Table 1. SET_LINE_CODING request format

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100001B	SET_LINE_CODING	Zero	Interface	Size of structure	Line Coding Structure

Table 2. GET_LINE_CODING request format

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100001B	GET_LINE_CODING	Zero	Interface	Size of structure	Line Coding Structure

Table 3. Line coding structure

Offset	Field	Size	Value	Description
0	dwDTERate	4	Number	Data terminal rate (in bits per second)
4	bCharFormat	1	Number	Stop bits 0 - 1 stop bit 1 - 1.5 Stop bits 2 - 2 Stop bits
5	bParityType	1	Number	Parity 0 - None 1 - Odd 2 - Even 3 - Mark 4 - Space
6	bDataBits	1	Number	Data bits 5, 6, 7, 8, or 16

Table 4. SET_CONTROL_LINE_STATE request format

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100001B	SET_CONTROL_LINE_STATE	Control Signal Bitmap	Interface	Zero	None

Table 5. Control Signal Bitmap Values for SetControlLineState

Bit Position	Description
D15….D2	RESERVED(Reset to zero)
D1	Carrier control for half duplex modems. This signal corresponds to V.24 signal 105 and RS-232 signal RTS. 0 - Deactivate carrier 1 - Activate carrier The device ignores the value of this bit when operating in full duplex mode.
D0	Indicates to DCE if DTE is present or not. This signal corresponds to V.24 signal 108/2 and RS-232 signal DTR. 0 - Not Present 1 - Present

Firmware

This Virtual COM Example Code contains three files:

Fw.c: Firmware framework from Cypress. It contains a common functionality with hooks for the user-specific functionality.

Virtual.c: Contains definitions of the functions and ISRs used for implementing the USB peripheral functionality as a virtual COM device.

DSCR.A51: Assembly source that contains the communication class and device class interfaces.

Since CDC is a composite USB device class it has two interfaces, Data class interface and Communication class interface. Communication interface includes class specific descriptors (which is specific to communication class devices), called Functional descriptors. Functional descriptors describe class-specific information within Communication class Interface descriptor. Data class interface doesn't hold any class specific descriptors.

Functional Descriptors

These descriptors hold class specific information within communication class interface descriptor. Following are functional descriptors.

- Header Functional Descriptor
- ACM Functional Descriptor
- Union Functional Descriptor
- Call Management Functional Descriptor

The descriptors are declared in the assembly source `DSCR.a51`. This assembly code is a part of attached project to this application note.

```
                                        ;; Header Functional Descriptor
 db    05H                              ;; Descriptor Size in Bytes (5)
 db    24H                              ;; CS_Interface
 db    00H                              ;; Header Functional Descriptor
 dw    1001H                            ;; bcdCDC


                                        ;; ACM Functional Descriptor
db    04H                               ;; Descriptor Size in Bytes (5)
db    24H                               ;; CS_Interface
db    02H                               ;; Abstarct Control Management Functional Desc
db    02H                               ;; bmCapabilities


                                        ;; Union Functional Descriptor
db    05H                               ;; Descriptor Size in Bytes (5)
db    24H                               ;; CS_Interface
db    06H                               ;; Union Functional Descriptor
db    00H                               ;; bMasterInterface
db    01H                               ;; bSlaveInterface0


                                        ;; CM Functional Descriptor
db    05H                               ;; Descriptor Size in Bytes (5)
db    24H                               ;; CS_Interface
db    01H                               ;; CM Functional Descriptor
db    00H                               ;; bmCapabilities
db    01H                               ;; bDataInterface
```

Header Functional Descriptor

Functional descriptor starts with Header functional descriptors. This is used to specify CDC version on which device is implemented.

ACM Functional Descriptor

It describes the commands supported by Communication Class Interface. Device supports the request combination of Set_Line_Coding, Set_ Control_Line_State, Get_Line_Coding and the notification Serial_State.

Union Functional Descriptor

Union Functional Descriptor describes the relationship among the interfaces which forms a single functional unit. It sets one of the interfaces in the group as a Master which functions as a controlling interface for the group and others as Slaves. Here Communication Class Interface acts as master and Data Class Interface as Slave. Host and device exchange requests and notifications via Communication class interface. Standard and class specific requests for the group are sent to this interface through EP0 control endpoint. Group sends notifications through EP1IN interrupt endpoint to host.

Call Management Functional Descriptor

Describes how device handles call management. Here device doesn't handle call management. Field bmCapabilities is a bitmap which shows information about device call management and interface used for exchange of call management information.

For more information on Functional descriptors see section 5.2.3 Functional Descriptors of CDC specification

The VID/PID can be changed according to the customer requirement in the Device Descriptor.

EndPoints

Communication Interface uses default endpoint EP0 for device management. Device uses EP0 for all standard and communication specific requests. Notification element is implemented by an interrupt endpoint, EP1IN.

Data interface endpoints are restricted to being either Bulk or Isochronous and should be pairs of same type. In the attached project, Data interface uses EP1OUT and EP8IN for data out and data in respectively, both as bulk.

Data Transfer

DATA OUT PATH

Following code implements data transfer from the Host to device.

```
if (!(EP1OUTCS & 0x02)) // Check if EP1OUT
is available

  {
   int bcl,i;
   bcl=EP1OUTBC;
   for(i=0;i<bcl;i++)
      WriteByteS0(EP1OUTBUF[i]);
   EP1OUTBC = 0x04;// Arms endpoint
  }
```

The BUSY bit (EP1OUTCS.2) indicates the status of the endpoint's OUT Buffer EP1OUTBUF. The USB core sets BUSY=0 when the host data is available in the OUT buffer. The statement `WriteByteS0(EP1OUTBUF[i])` passes each bytes in EP1OUT buffer to WriteByteS0 () function. The function `WriteByteS0` checks last byte has been transmitted successfully and then copy data byte to SBUF0.

TI is Transmit Flag. Set when a byte has been completely transmitted. RI is Receive Flag. Set when a byte has been completely received.

The line of code `EP1OUTBC = 0x04;` in the firmware arms the endpoint for receiving next byte from Host.

```
void WriteByteS0(char ch )  // Send the data
recieved from USB through SBUF0
{

   while (TI == 0) ;// Checks last byte
//has been transmitted or not.
   TI = 0 ;
   SBUF0 = ch ;

}
```

DATA IN PATH

Following code accomplishes device to host data transfer.

```
// recieve the data from UART and send it
out to the USB Host
if((EP2468STAT & bmEP8EMPTY))   // check if
EP8 is empty
  {
   if (RI)//Checks if new data is received
      {
   EP8FIFOBUF [0] = ReadByteS0();
   EP8BCH = 0;
             SYNCDELAY;
               EP8BCL = 1; // Commits
endpoint
             SYNCDELAY;

    }

  }
```

Above code calls ReadByteS0() function if EP8IN endpoint is availability and receive interrupt flag is asserted. Function `ReadByteS0` reads data from serial buffer SBUF and returns it. Returned value is copied to EP8IN endpoint and sent to host.

```
BYTE   ReadByteS0 (void)     // Copy the
recieved Byte from SBUF to the variable
RxByte0
{

         RI = 0;      // with reg320 syntax
but same address
         RxByte0 = SBUF0;   //RxByte;
        return RxByte0;

}
```

Communication Management

CDC defines class specific requests and notification component for device management. SET_LINE_CODING, GET_LINE_CODING and SET_CONTROL_STATE are the class specific requests. Their values are defined by following codes.

```
#define SET_LINE_CODING (0x20)
#define GET_LINE_CODING (0x21)
#define SET_CONTROL_STATE (0x22)
```

Following code identifies request type and sets asynchronous serial parameters or sends the asynchronous serial parameters to Host.

SET_LINE_CODING *case* copies data packet of the control transfer to LineCode array and passes it to Serial0Init() function. This data packet contains the asynchronous serial parameters sent by host. Its byte structure is given in Table 3. The `Serial0Init()` function parses the contents of LineCode array and sets corresponding baud rate.

```
switch(SETUPDAT[1])
   {
      case SET_LINE_CODING:
         Len = 7;
         EUSB = 0 ;
         SUDPTRCTL = 0x01;
         EP0BCL = 0x00;
            SUDPTRCTL = 0x00;
         EUSB = 1;

          while (EP0BCL != Len);
   SYNCDELAY;
          for (i=0;i<Len;i++)
   LineCode[i] = EP0BUF[i];
            Serial0Init();
   break;
```

GET_LINE_CODING *case* copies LineCode contents to EP0BUF and commits it. As per CDC specification the device is required to report default line coding settings in response to GET_LINE_CODING request from the Host. Therefore this case statement copies line coding data to *EP0BUF* from *LineCode* array and commits it.

```
case GET_LINE_CODING:

SUDPTRCTL = 0x01;
Len = 7;
for (i=0;i<Len;i++)
EP0BUF[i] = LineCode[i];
EP0BCH = 0x00;
SYNCDELAY;
EP0BCL = Len;
   SYNCDELAY;
while (EP0CS & 0x02);
   SUDPTRCTL = 0x00;
    break;
```

Since serial port of FX2LP doesn't have DTR and DTE pins this application doesn't include SET_CONTROL_STATE.

```
case SET_CONTROL_STATE:
 break;
```

Baud Rate Selection

Below code checks the host sent baud rate in LineCode[0] and LineCode[1] and then sets that particular baud rate. Following are the baud rates supported in the associated project 2400, 4800, 9600, 19200, 28800, 38400, 57600, 115200 and 230400 bits per second.

The `Serial0Init()` function configures the serial pot of FX2LP for the baud rate requested by host.

```
void Serial0Init() // serial UART 0 with Timer 2 in mode 1 or high speed baud rate generator
{

        if ((LineCode[0] == 0x60) && (LineCode[1] == 0x09 ))      // 2400
    {
                SCON0  = 0x5A; //   Set Serial Mode = 1, Recieve enable bit = 1
                T2CON  = 0x34;  //  Int1 is detected on falling edge,
                              // Enable Timer0, Set Timer overflow Flag
                RCAP2H = 0xFD;  //  Set TH2 value for timer2
                RCAP2L = 0x90;  //  baud rate is set to 2400 baud
                TH2    = RCAP2H;//  Upper 8 bit of 16 bit counter to FF
                TL2    = RCAP2L; //  value of the lower 8 bits of timer set to baud rate
    }

   else if ((LineCode[0] == 0xC0) && (LineCode[1] == 0x12 ))       // 4800
    {
                SCON0  = 0x5A;      //Set Serial Mode = 1, Recieve enable bit = 1
                T2CON  = 0x34;      //Int1 is detected on falling edge, Enable
                             //Timer0, Set Timer overflow Flag
                RCAP2H = 0xFE;             //Set TH2 value for timer2
                RCAP2L = 0xC8;             //baud rate is set to 4800 baud
                TH2    = RCAP2H;           //Upper 8 bit of 16 bit counter to FF
                TL2    = RCAP2L; // value of the lower 8 bits of timer set to baud rate
    }

   else if ((LineCode[0] == 0x80) && (LineCode[1] == 0x25 ))       // 9600
    {
                SCON0  = 0x5A;   // Set Serial Mode = 1, Recieve enable bit = 1
                T2CON  = 0x34;   // Int1 is detected on falling edge,
                            //Enable Timer0, Set Timer overflow Flag
                RCAP2H = 0xFF;             //  Set TH2 value for timer2
                RCAP2L = 0x64;             //     baud rate is set to 9600 baud
                TH2    = RCAP2H;           //  Upper 8 bit of 16 bit counter to FF
                TL2    = RCAP2L;// value of the lower 8 bits of timer set to baud rate
    }

   else if ((LineCode[0] == 0x00) && (LineCode[1] == 0x4B ))       // 19200
    {
```

```
            SCON0  = 0x5A; //Set Serial Mode = 1, Recieve enable bit = 1
            T2CON  = 0x34; //Int1 is detected on falling edge,
                           //Enable Timer0, Set Timer overflow Flag
            RCAP2H = 0xFF;             //  Set TH2 value for timer2
            RCAP2L = 0xB2;             //     baud rate is set to 19200 baud
            TH2    = RCAP2H;           //  Upper 8 bit of 16 bit counter to FF
            TL2    = RCAP2L; // value of the lower 8 bits of timer set to baud rate
      }
   else if ((LineCode[0] == 0x80) && (LineCode[1] == 0x70 ))        // 28800
      {
            SCON0  = 0x5A;   // Set Serial Mode = 1, Recieve enable bit = 1
            T2CON  = 0x34; //   Int1 is detected on falling edge,
                             //Enable Timer0, Set Timer overflow Flag
            RCAP2H = 0xFF;             //  Set TH2 value for timer2
            RCAP2L = 0xCC;             //     baud rate is set to 28800 baud
            TH2    = RCAP2H;           //  Upper 8 bit of 16 bit counter to FF
            TL2    = RCAP2L; // value of the lower 8 bits of timer set to baud rate
      }

else if ((LineCode[0] == 0x00) && (LineCode[1] == 0x96 ))    // 38400
            {
            SCON0  = 0x5A;//    Set Serial Mode = 1, Recieve enable bit = 1
            T2CON  = 0x34;                // Int1 is detected on falling edge,
                                  //Enable Timer0, Set Timer overflow
            RCAP2H = 0xFF;             //  Set TH2 value for timer2
            RCAP2L = 0xD9;             //     baud rate is set to 38400 baud
            TH2    = RCAP2H;           //  Upper 8 bit of 16 bit counter to FF
            TL2    = RCAP2L;//  value of the lower 8 bits of timer set to baud rate
      }

else if ((LineCode[0] == 0x00) && (LineCode[1] == 0xE1 ))    // 57600
      {
            SCON0  = 0x5A;  //  Set Serial Mode = 1, Recieve enable bit = 1
            T2CON  = 0x34;                // Int1 is detected on falling edge,
                                 //Enable Timer0, Set Timer overflow Flag
            RCAP2H = 0xFF;             //  Set TH2 value for timer2
            RCAP2L = 0xE6;             //     baud rate is set to 57600 baud
            TH2    = RCAP2H;           //  Upper 8 bit of 16 bit counter to FF
            TL2    = RCAP2L; //value of the lower 8 bits of timer set to baud rate
      }

else if ((LineCode[0] == 0x00) && (LineCode[1] == 0x84 ))     // 230400
      {
         PCON |= 0X80;
         UART230 |= 0x03;

      }
else
      {

         SCON0  = 0x5A;// Set Serial Mode = 1, Recieve enable bit = 1
         T2CON  = 0x34;
         RCAP2L = 0xF3;
         RCAP2H = 0xFF;
         TH2    = RCAP2H; //  Upper 8 bit of 16 bit counter to FF
         TL2    = RCAP2L;//  value of the lower 8 bits of timer set to baud rate*/

      }
 }
```

INF File

The virtual COM port enumeration requires an INF file. This file points the VID/PID of the device to the default Windows COM port driver, usbser.sys. The INF file is included along with this application note.

Test Procedure

Hardware Requirements

- CY3684 EZ-USB FX2LP Development Kit.
- PC with USB and UART ports (UART to USB cable can be used if the PC does not have a UART port).
- USB cable to connect to the PC from the FX2LP USB port.
- UART cable to connect to the PC from the FX2LP serial Port0.

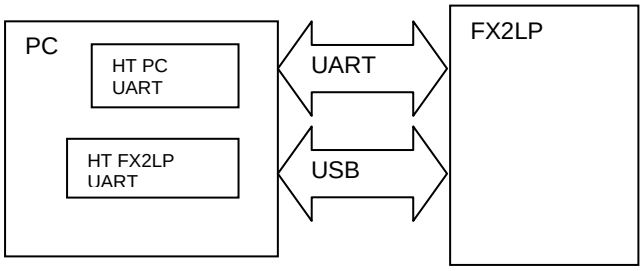
Software Requirements

- Cypress USB console (CyConsole) to download firmware into FX2LP. This can be downloaded and installed from www.cypress.com.
- HyperTerminal or equivalent software (Teraterm) must be available in the PC.

Hardware Setup

Figure 5 shows block diagram of the hardware test setup.

Figure 5. Hardware Test Setup



HW Connections

1. Connect the USB port of FX2LP DVK to the PC using the USB cable.
2. Connect the UART port of FX2LP DVK to the PC using the UART cable (or UART to USB cable if the PC does not have a UART port).

Procedure

Use the example code that accompanies this application note. This code is in the virtual COM Example Code folder.

To make the required Virtual COM port firmware hex file, open the *Virtual.Uv2* file using the Keil uVision tool suite. After you open the project, go to the **Project** tab and click **Rebuild all target files**. A *.hex* file (*VirtualCom.hex*) is generated. You will use this compiled hex file later in this application note.

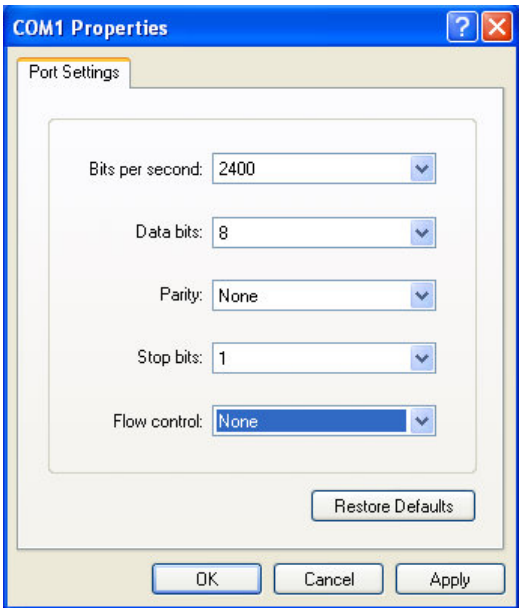
Hyper Terminal Settings

Note: The project included with this application note is tested and works correctly on the 32-bit Windows XP (32 and 64 bit), Windows 7 (32 and 64 bit), and Vista (32 and 64 bit) operating systems. Attached INF file can be used for all of the above mentioned PCs.

1. On the host PC, go to **Start** > **All Programs** > **Accessories** > **Communications** and click **HyperTerminal**.
   This opens a new connection and prompts for the connection description in a separate dialog box.
2. Enter a name for the connection ('PC UART') and click **OK**.

3. In the **Connect Using** drop-down box, select the COM port to which FX2LP DVK serial port is connected (COM6 in this example). Click **OK**.

4. The ‘COMx properties’ window will appear. Since this example is written for 2400, 4800, 9600, 19200, 28800, 38400, 57600, 115200 and 230400 Bits per second baud rates select any of these baud-rate values. **Change** the ‘Flow Control’ to ‘None’. **Do not change** ‘data bits’, ‘parity’ or ‘stop bits’. Click **OK**.

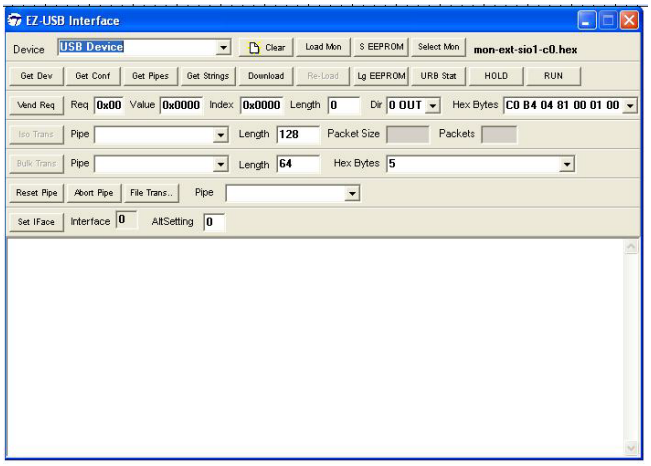


Firmware Download

The Cypress USB utility CyConsole uses USB to download the compiled *.hex* file.

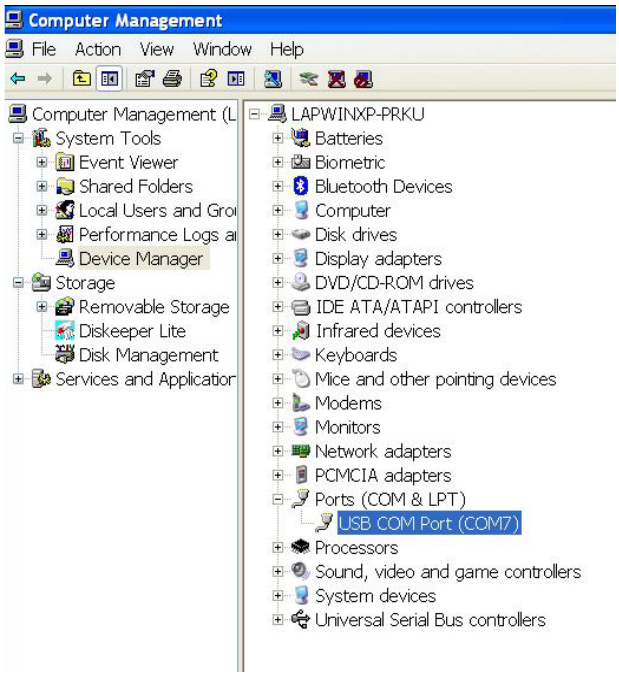
1. Plug the CY3684 EZ-USB® FX2LP Development Kit board into the PC using a USB cable.
2. Launch the Cypress USB console (CyConsole).
3. Go to **Start** > **All Programs** > **Cypress** > **USB** > **CyConsole EZ-USB**.

The application launches and the following window is displayed.

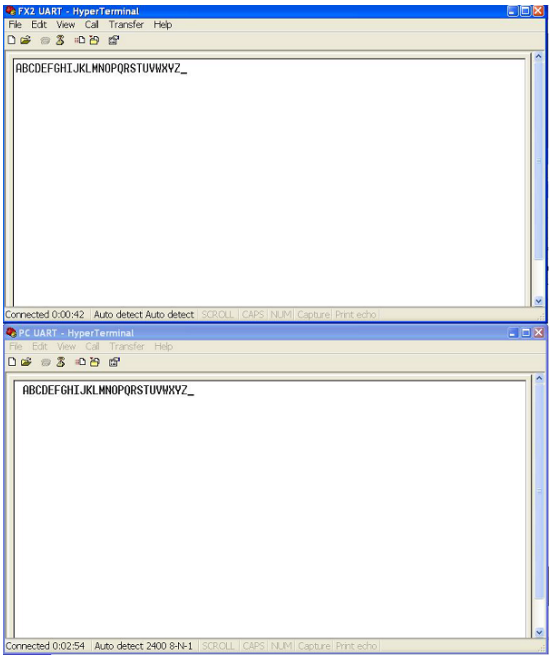


5. Download the compiled hex file by clicking the **Download** tab and selecting the path where the hex file is located. When the download is complete, it prompts for a driver.
6. Select the driver as the *CyUsbCom.inf* file attached with this application note.

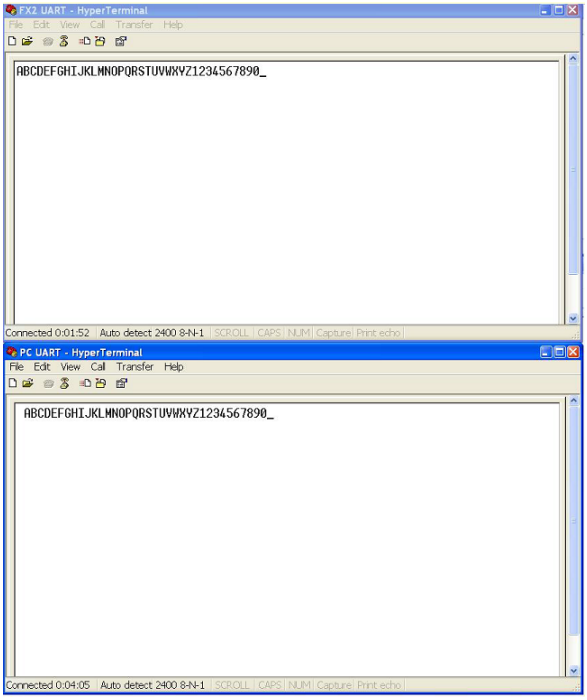
After installation, the compiled hex file appears in the device manager under Ports.



7. Follow the procedure described earlier for the HyperTerminal settings with this virtual COM port. However, change the name PC UART to FX2LP UART.
   Select the same baud rate which is selected for PC Hyper terminal. Now the two HyperTerminals (PC UART and FX2LP UART) are ready to communicate with each other.
8. Send characters and numbers from FX2LP UART by typing the characters in the FX2LP UART HyperTerminal. They are received in the PC UART Terminal.



9. Now send characters and numbers from PC UART by typing some numbers in the PC UART HyperTerminal. They are received in the FX2LP UART Terminal.



Following are the steps to download firmware onto EEPROM.

a) Download SuiteUSB 3.4 and install it. This installs the CyConsole utility.
b) Connect the CY3684 board to PC with EEPROM ENABLE switch in “No EEPROM” position. Now the board enumerates with the default internal descriptor. Use *CyUSB.inf* available in the folder vcp\INF file\Driver to bind with the device. For help with binding the driver, see “Matching Devices to the Driver” section of Cypress CyUsb.sys Programmer's Reference.
c) Open CyConsole utility. Go to Start > All **Programs** > Cypress > USB > **CyConsole EZ-USB**.
d) On DVK set SW1 to large EEPROM and SW2 to EEPROM position. Download the compiled iic file ‘*VirtualCom.iic’* to EEPROM by clicking the **Lg EEPROM** tab and selecting the path where the iic file is located. When the download is complete reset DVK, it prompts for a driver.
e) Add new VID/PID (one used in dscr.asm file) to *CyUsbCom.inf* and then bind it with the device.

Reference

Serial Port Complete II by Jan Axelson is a good reference for virtual COM port–related development.

Summary

This application note explains Serial port emulation on USB port using the standard Windows driver. This application note provides overview of CDC class and shows how to implement it in FX2LP. This document includes detailed explanation of firmware and step by step procedure to test the application.

Related Application Notes

Getting Started with FX2LP - AN65209.

About the Author

Name:	Prajith Cheerakkoda
Title:	Applications Engineer
Contact:	prji@cypress.com

Document History

Document Title: Implementing a Virtual COM Port using FX2LP – AN58764

Document Number: 001-58764

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	2845294	PRKU	01/18/2010	New application note
*A	3174314	SSJO	02/15/2011	Updates per review
*B	3242361	SSJO	04/27/2011	Added information about 2400 baud rate. Updated code.
*C	3411330	PRJI	10/17/2011	Minor text edits. Updated template.
*D	3660417	PRJI	07/05/2012	Updated Introduction. Updated Communication Device Class Specification. Updated Firmware (Added EndPoints, Data Transfer, Communication Management, Baud Rate Selection). Updated Test Procedure. Updated Summary. Updated in new template.

Worldwide Sales and Design Support

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

Products

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

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 5

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Technical Support

cypress.com/go/support

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

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