Application Note

Z8F6482 MCU USB Peripheral

AN041101-0318

Abstract

This application note demonstrates the setup and use of the Z8F6482 microcontroller's Universal Serial Bus (USB) peripheral to create a USB device to communicate with a Windows PC.

Note: The source code file associated with this application note, AN0411-SC01.zip, is available free for download from the Zilog website. The source code in these files has been tested with ZDS II version 5.2.2 for the Encore! XP Series of MCUs. Subsequent releases of ZDS II may require you to modify the code supplied with this application note.

Features

The Z8F6482 MCU USB module provides USB full-speed device functionality with eight USB endpoints. Features include:

- Full-speed (12 Mbps) USB device
- IN endpoint 0 and OUT endpoint 0 control endpoints
- IN endpoints 1–3 and OUT endpoints 1–3 capable of bulk and interrupt transfers
- USB Suspend, host-initiated Resume, and device-initiated Resume (remote wake-up)
- USB clock of 48 MHz from internal PLL or external clock source
- 512 bytes of dedicated USB endpoint buffer memory; each endpoint buffer memory can be configured as 8, 16, 32, or 64 bytes
- Integrated full-speed USB PHY with integrated pull-up resistor
- Support for two DMA channels

Discussion

The Z8F6482 MCU's USB module is a USB 2.0 compliant full-speed device (12 Mbps) with an integrated Physical Layer Transceiver (PHY) and dedicated buffer memory space. The USB module handles serial-to-parallel conversion for received data and parallel-to-serial conversion for transmitting data.

This USB module requires an accurate 48 MHz clock, which can be supplied through the Z8F6482 MCU's internal PLL with an external clock source. Ensure that the clock is con-

figured correctly as described in the Clock System section of the Z8 Encore XP F6482 Series Product Specification (PS0294).

Terminology

Host (or USB Host). Acts as the *master* in a USB communication (i.e., PC). All communications are initiated by the host.

Device (or USB Device). Acts as the *slave* in a USB communication (i.e., mouse, keyboard)

USB Module. Refers to the Z8F6482 MCU's USB peripheral

Interrupts

The Z8F6482 MCU's interrupt controller is responsible for handling all interrupts triggered by each peripheral, including the following two interrupts for the USB module:

- USB Interrupt Request (USBI)
- USB Resume Interrupt Request (USBRI).

Additionally, the USB module contains its own interrupt registers to describe the source of the USB interrupt.

USBRI is triggered by a resume event from the USB module, which can be either a host-initiated resume or a device-initiated resume. A host-initiated resume is always enabled and cannot be disabled by software whereas a device-initiated resume can be enabled or disabled by software. Table 1 lists the registers that affect a USB resume interrupt.

Table 1. USB Resume Interrupt Registers

Register	Bit	Description
IRQ0ENH/L	[1] USBRENH/L	USB Resume Interrupt Request Enable High/Low Bit
IRQ0	[1] USBRI	An interrupt request from USB Resume is awaiting service
USBIRQCTL	[1] RIRQE	Enable or disable device-initiated resume

USBI is triggered by several sources from the USB module, including:

- Error-free Setup Data Packet Received (SUDAVIRQ)
- Start of Frame Received (SOFIRQ)
- Setup Token Received (SUTOKIRQ)
- USB Suspend Detected (SUSPIRQ)
- USB Reset Bus State Detected (URESIRQ)
- IN/OUT Endpoint 0–3 Interrupt (INxIRQ, OUTxIRQ)

Table 2 lists the registers that affect a USB interrupt.

Table 2. USB Interrupt Registers

Register	Sub-Register	Bit	Description
IRQ0ENH/L		[2] USBENH/L	USB Interrupt Request Enable High/Low Bit
		[1] USBRENH/L	USB Resume Interrupt Request Enable High/Low Bit
IRQ0		[2] USBI	USB Interrupt Request
		[1] USBRI	USB Resume Interrupt Request
USBIEN	2Eh	[4] URESIEN	Enable or disable interrupt on USB reset bus state detected
		[3] SUSPIEN	Enable or disable interrupt on USB suspend detected
		[2] SUTOKIEN	Enable or disable interrupt on USB setup token received
		[1] SOFIEN	Enable or disable interrupt on USB start of frame packet received
		[0] SUDAVIEN	Enable or disable interrupt on Error Free Setup Data Packet received
USBINIEN	2Ch	[3] IN3IEN	Enable or disable interrupt on IN3 endpoint
		[2] IN2IEN	Enable or disable interrupt on IN2 endpoint
		[1] IN1IEN	Enable or disable interrupt on IN1 endpoint
		[0] IN0IEN	Enable or disable interrupt on IN0 endpoint
USBOUTIEN	2Dh	[3] OUT3IEN	Enable or disable interrupt on OUT3 endpoint
		[2] OUT2IEN	Enable or disable interrupt on OUT2 endpoint
		[1] OUT1IEN	Enable or disable interrupt on OU1 endpoint
		[0] OUT0IEN	Enable or disable interrupt on OUT0 endpoint
USBIID	28h	[6:2] IID	Interrupt identification source
USBIRQ	2Bh	[4] URESIRQ	USB reset bus state detected
		[3] SUSPIRQ	USB suspend detected
		[2] SUTOKIRQ	USB setup token received
		[1] SOFIRQ	USB start of frame packet received
		[0] SUDAVIRQ	USB error-free setup stage data packet received
USBINIRQ	29h	[3] IN3IRQ	Interrupt request from IN3 endpoint
		[2] IN2IRQ	Interrupt request from IN2 endpoint
		[1] IN1IRQ	Interrupt request from IN1 endpoint
		[0] IN0IRQ	Interrupt request from IN0 endpoint
USBOUTIRQ	2Ah	[3] OUT3IRQ	Interrupt request from OUT3 endpoint
		[2] OUT2IRQ	Interrupt request from OUT2 endpoint
		[1] OUT1IRQ	Interrupt request from OUT1 endpoint
		[0] OUT0IRQ	Interrupt request from OUT0 endpoint

Endpoints

The Z8F6482 MCU's USB module includes four endpoint pairs, including endpoint 0. Endpoints act as a buffer on USB devices that are used for sending or receiving data. Data received by the USB module can be read from the OUT endpoint buffer, while data transmitted by the USB module must be written to the IN endpoint buffer. Endpoint 0 is the default endpoint for all USB devices and is used by the host to control USB devices. Therefore, Endpoint 0 must always be enabled. Table 3 lists the registers that affect endpoint access.

Table 3. Endpoint-Related Registers

Register	Sub-Register	Bit	Description
USBOUTVAL	5Fh	[3] OUT3VAL	Defines if OUT3 endpoint is valid or not
		[2] OUT2VAL	Defines if OUT2 endpoint is valid or not
		[1] OUT1VAL	Defines if OUT1 endpoint is valid or not
		[0] OUT0VAL	Defines if OUT0 endpoint is valid or not
USBINVAL	5Eh	[3] IN3VAL	Defines if IN3 endpoint is valid or not
		[2] IN2VAL	Defines if IN2 endpoint is valid or not
		[1] IN1VAL	Defines if IN1 endpoint is valid or not
		[0] IN0VAL	Defines if IN0 endpoint is valid or not

Table 3. Endpoint-Related Registers (Continued)

Register	Sub-Register	Bit	Description
USBEP0CS	34h	[5] CHGSET	Setup Buffer Contents Changed 0: No change to the setup buffer contents. 1: Setup buffer contents changed. Software must clear this bit after reading by writing a 1 to this bit. Automatically set to 1 when the USB module receives a setup data packet.
		[4] DSTALL	Endpoint 0 Data Stall 0: Do not send a STALL handshake for any IN or OUT token in the Data stage. 1: Send a STALL handshake for any IN (IN endpoint) or OUT (OUT endpoint) token in the Data stage.
		[3] OUTBUSY	Endpoint 0 OUT Busy Status 0: Buffer access is enabled for software. 1: Buffer access is exclusive to the USB module. Automatically set when software writes to BC.
		[2] INBUSY	Endpoint 0 IN Busy Status 0: Buffer access is enabled for software. Automatically cleared when a setup token is received. 1: Buffer access is exclusive to the USB module. Automatically set when software writes a value to BC.
		[1] HSNAK	Endpoint 0 Handshake 0: Do not send a NAK handshake 1: Send a NAK handshake
		[0] STALL	Endpoint 0 Stall 0: Do not send a STALL handshake 1: Send a STALL handshake
USBOxCS	46h, 48h, 4Ah	[1] OUTBUSY	OUT 1-3 Endpoint Busy Status (Read-Only) 0: Buffer access is enabled for software. Automatically cleared when the USB module receives an error-free data packet.Buffer ready to read by software. 1: Buffer access is exclusive to the USB module. Automatically set when software writes to BC. Buffer is empty. USB Interrupt is generated.
		[0] STALL	OUT 1-3 Stall 0: Do not send a STALL handshake 1: Send a STALL handshake
USBOxBC	45h, 47h, 49h, 4Bh	[6:0] BC	OUT 0-3 Byte Count Contains the number of bytes to read from the OUTx buffer.

Table 3. Endpoint-Related Registers (Continued)

Register	Sub-Register	Bit	Description
USBIxCS	36h, 38h, 3Ah	[1] INBUSY	IN 1-3 Endpoint Busy Status (Read-Only) 0: Buffer access is enabled for software. Automatically cleared when the USB module finishes transmitting the BC length data packet. 1: Buffer access is exclusive to the USB module. Buffer is empty and automatically set when software writes a value to BC.
		[0] STALL	IN 1–3 Stall 0: Do not send a STALL handshake 1: Send a STALL handshake
USBIxBC	35h, 37h, 39h, 3Bh	[6:0] BC	IN 0–3 Byte Count Contains the number of bytes loaded onto the INx buffer.

Endpoint Buffer Memory

The USB module provides a total of 512 bytes of endpoint buffer space, giving up to 64 bytes of memory for each endpoint. Endpoint allocation is arranged sequentially in the memory starting from OUT0 endpoint, followed by endpoints OUT1, OUT2, and OUT3. IN endpoints are placed after OUT3 endpoint in a similar order – IN0, IN1, IN2, and IN3. OUT0 endpoint starts at a fixed location at address 000h. Table 4 lists the registers that define the start addresses of each endpoint.

Table 4. Endpoint Buffer Memory Allocation Registers

Register	Buffer Address	Description
USBOxADDR	USBOxADDR << 1	starting address for OUTx endpoint
USBISTADDR	USBISTADDR << 2	starting address for IN0 endpoint
USBIxADDR	(USBISTADDR << 2) + (USBIxADDR << 1)	starting address for INx endpoint
USBISPADDR	(USBISPADDR << 4) - 1	ending address for the last numbered IN endpoint (IN3, if enabled)

The firmware for this application note uses software DAC Triggering, which indicates that DAC conversion is started when data is written to the DACD_H Register.

Figure 1 shows the endpoint buffer memory allocation, depending on the assigned values for the registers.

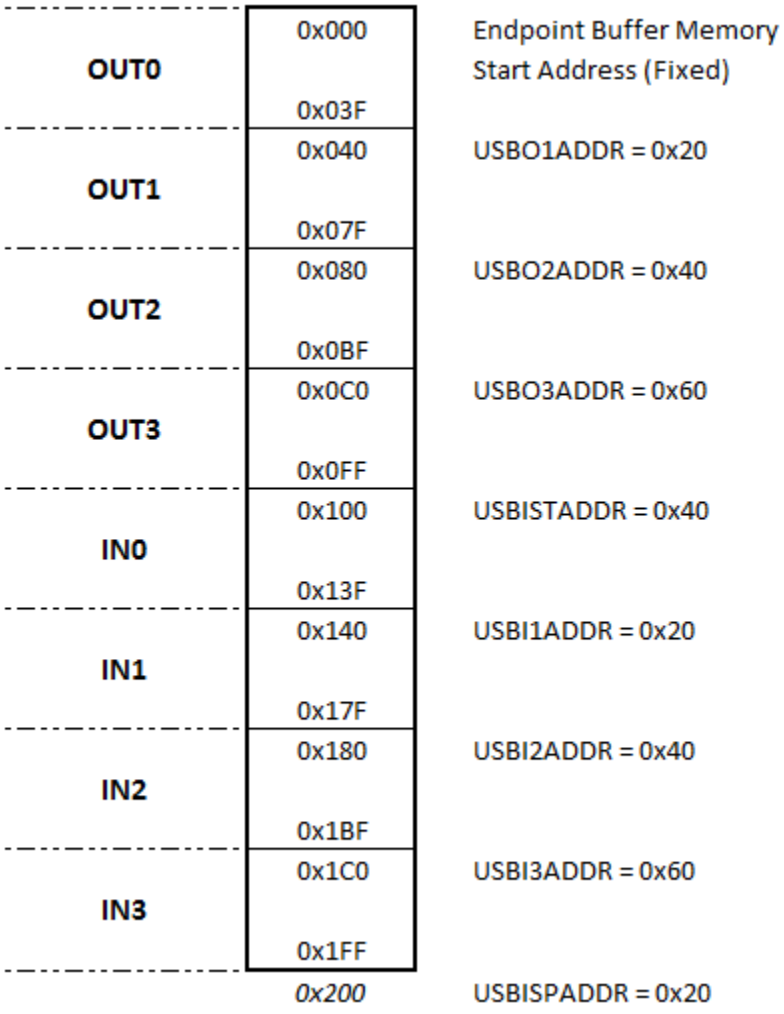

Figure 1. Example Endpoint Buffer Memory Allocation

Data Transfer via Endpoint 1–3

Endpoints 1–3 are capable of bulk and interrupt transfers. Both transfer types use IN and OUT transactions in a similar fashion. The difference lies in the manner in which the host carries out the transactions ─ in an interrupt transfer, the host issues an IN or OUT transaction at regular intervals, whereas in a bulk transfer, the host issues an IN or OUT transaction only as required.

The IN and OUT transactions consist of the following three packets:

- Token packet
- Data packet
- Handshake packet

A token packet acts as the header of a transaction which defines the target endpoint and details of what follows next. A token packet can either be a SETUP, OUT, or IN token to indicate the type of transaction being performed. A data packet contains the payload, and a handshake packet is used to ACK, NAK, or STALL transactions. The token and handshake packets are handled by the USB module. Therefore, there is no need to process these packets via software. The USB module sends interrupts and sets the necessary flags in the USB registers for processing by the software.

IN Transaction

When the host requests data from the device, it issues an IN token. Figure 2 shows the communication flow during an IN transaction.

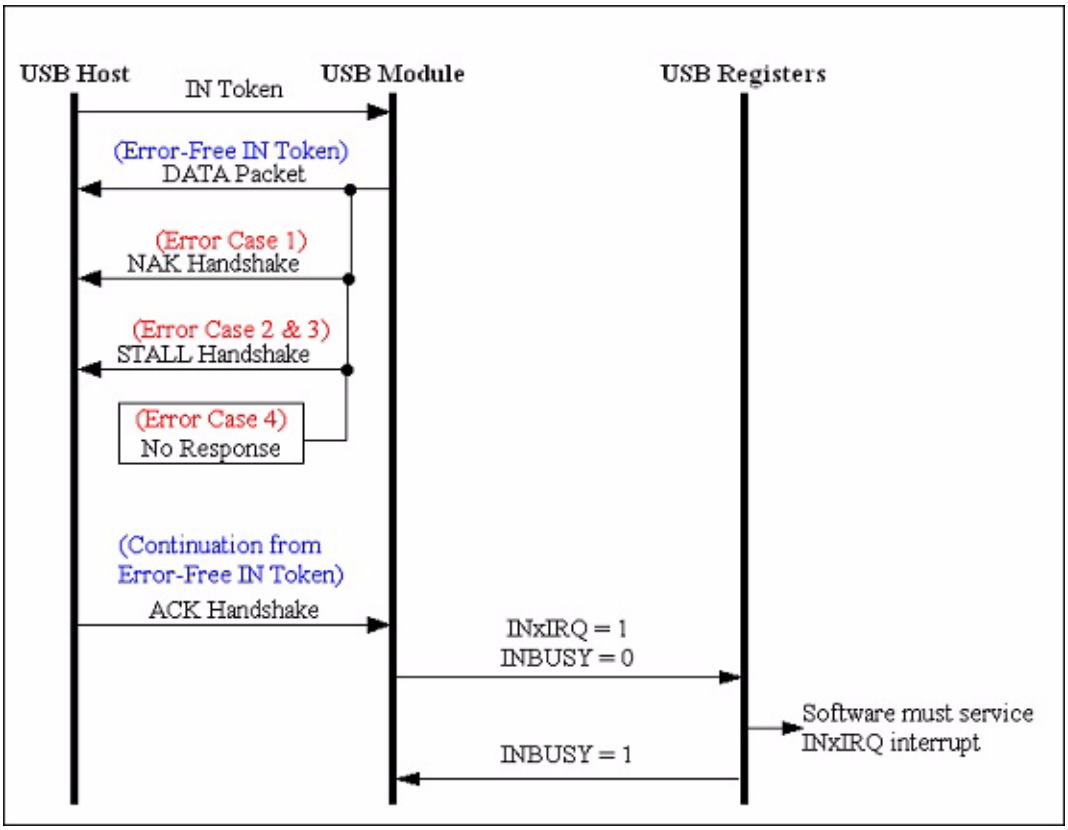

Figure 2. IN Transaction Communication Flow

Notice that communication ends with a NAK/STALL/No Response when the module detects an error. Table 5 shows the conditions to determine an error case.

Table 5. Error Conditions — IN Transaction

		USBIxCS		
Error Case	**Error in Token**	**INBUSY**	**STALL**	**Module Response**
1	NO	0	0	NAK
2	NO	0	1	STALL
Error-free	NO	1	0	BC bytes data packet
3	NO	1	1	STALL
4	YES	x	x	No response

If the host receives an error-free data packet, it sends an ACK handshake to the module. Upon receiving an ACK handshake, the USB module clears INBUSY to indicate that the software is allowed access to the IN endpoint buffer memory. The module also sets USBINIRQ.INxIRQ, in which software must respond with the following steps:

1. Write new data into the INx endpoint buffer memory
2. Write USBIxBC.BC with the number of bytes written in the INx endpoint buffer memory (The USB module automatically sets USBIxCS.INBUSY to 1 when USBIxBC.BC is written with a new value).

OUT Transaction

When the host sends data to the device, it issues an OUT token. Figure 3 shows the communication flow during a bulk OUT transfer.

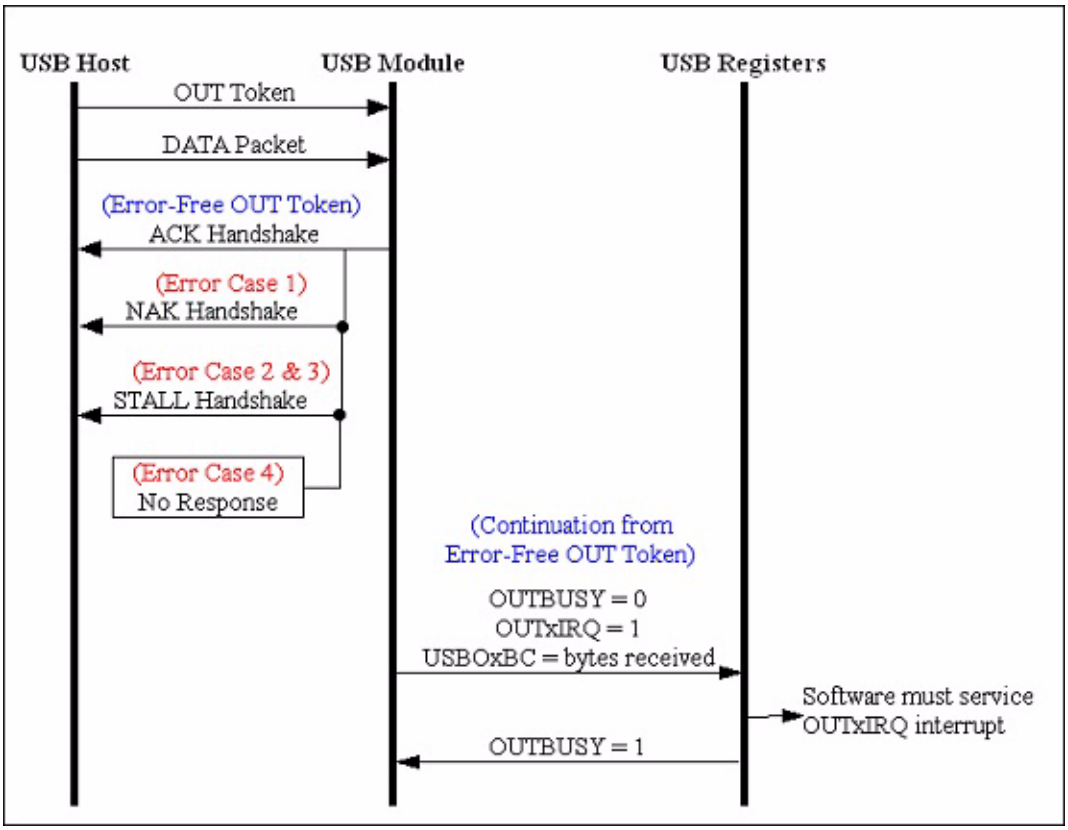

Figure 3. OUT Transaction Communication Flow

Note that communication ends with a NAK/STALL/No Response when the module detects an error. Table 6 shows the conditions to determine an error case.

Table 6. Error Conditions – OUT Transaction

		USBOxCS		
Error Case	**Error in Token**	**OUTBUSY**	**STALL**	**Module Response**
1	NO	0	0	NAK
2	NO	0	1	STALL
Error-free	NO	1	0	ACK
3	NO	1	1	STALL
4	YES	x	x	No response

If no error is detected, the USB module sends an ACK handshake to the host. Additionally, the module clears USBOxCS.OUTBUSY to indicate that the software can now

access the OUT endpoint buffer memory. The module also sets OUTxIRQ in which the software must respond with the following steps:

1. Read the received data packet from the OUTx endpoint buffer memory
2. Write a dummy value to USBOxBC.BC (The USB module automatically sets USBOxCS.OUTBUSY to 1 when USBOxBC.BC is written with a new value).

Control Transfer via Endpoint 0

A control transfer is used during USB enumeration so that the host can identify the type of device attached to it, and the type of transfer to use for succeeding communications. Therefore, it is mandatory that a device implements a control transfer. Control transfers always occur on Endpoint 0 and are divided into the following three stages:

- Setup stage
- Data stage
- Status stage

Setup Stage

During the setup stage, the host issues a SETUP token. Upon receiving a SETUP token, the module sets HSNAK and SUTOKIRQ. At this point, no processing is required for software; therefore, SUTOKIEN can be disabled. The host then sends an 8-byte DATA packet, SETUP DATA PACKET, which defines the type of request issued by the host. Upon receiving this packet, the USB module stores it in the USBSUx subregister and then sets CHGSET to indicate that it already contains the SETUP DATA PACKET. Figure 4 displays the communication flow during the setup stage of a control transfer.

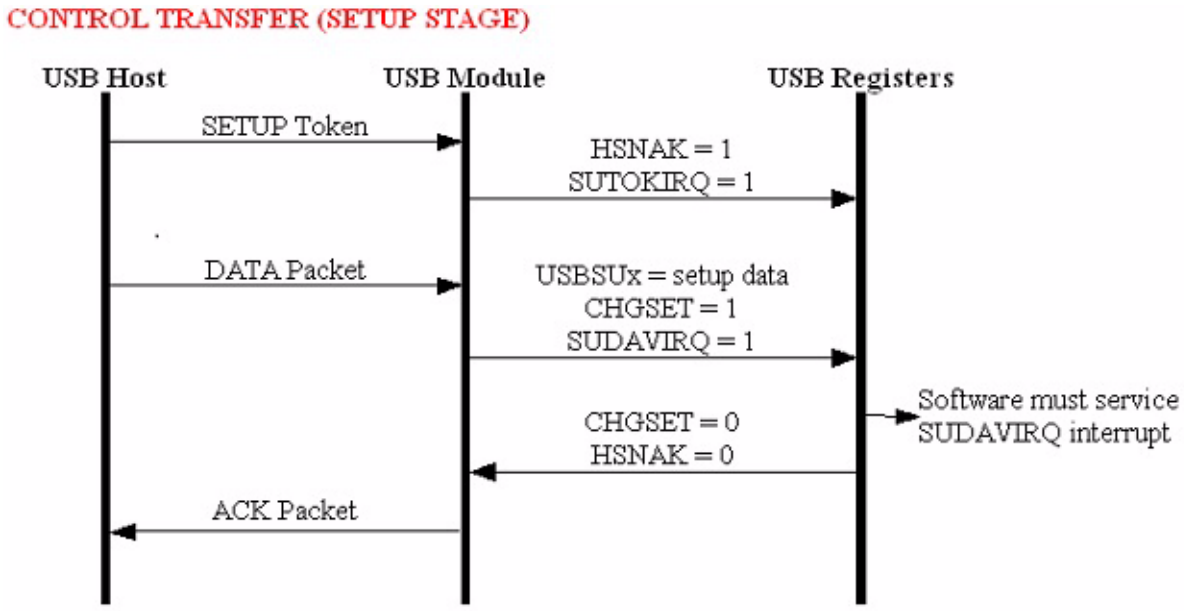

Figure 4. Control Write Transfer Communication Flow

If the SETUP DATA PACKET is received error-free, the module will set SUDAVIRQ, in which software must respond with the following steps:

1. Read data packet in USBSUx
2. Clear CHGSET (write 1) to indicate that the setup data packet is read
3. Clear HSNAK (write 1) to ACK the status stage
4. Clear the interrupt (SUDAVIRQ, SUTOKIRQ, SOFIRQ)
5. Process the setup data packet received. Depending on request type (GET or SET), this may involve writing to the IN0 endpoint buffer memory (GET) or reading from the OUT0 endpoint buffer memory (SET)

Data Stage

The data stage can either be an IN or an OUT transaction, depending on request type. A GET request indicates that the host has requested reading data from the device. In this situation, the data stage contains an IN transaction. The software must process this request in a manner similar to an IN transaction.

A SET request indicates that the host wishes to write data to the device. In this situation, the data stage contains an OUT transaction. Software must process this request in a manner similar to an OUT transaction.

Status Stage

The status stage can also be an IN or an OUT transaction. It is always in the opposite direction of the data stage. Thus, a GET request will have an OUT transaction performed in the status stage, and a SET request will have an IN transaction as the status stage.

Initializing the USB Module

The software must perform the following basic steps to initialize the USB module:

1. Select port alternate function for USB pins
2. Configure the endpoint buffers' memory
3. Enable the IN endpoints to be used and enable corresponding endpoint interrupt
4. Enable the OUT endpoints to be used and enable corresponding endpoint interrupt
5. Write a dummy value to each enabled OUT endpoints byte count register, USBOxBC, to enable data reception
6. Enable interrupts from the USB module
7. Enable interrupts from the MCU's interrupt controller
8. Clear DISCON to connect the USB module to the USB bus

USB Suspend/Resume

Suspend is a mechanism for reducing power consumption from the USB device when there is no traffic. The host initiates SUSPEND by idling the bus for at least 3msec. When

the USB device is in SUSPEND state, either the USB device itself or the USB host can cause a Resume state by changing the bus state to non-idle.

Suspend

When the module detects an idle condition for at least 3 msec, SUSPIRQ is generated. The software reacts to this interrupt by going into a reduced power suspend state with the following steps:

1. Read USBIRQ to determine if the interrupt is caused by a Suspend request.
2. Write any value to USBCLKGATE to gate off the USB clock and power down the USB PHY.
3. If using PLL and the USB clock source is dedicated to USB (not selected as system clock source), disable the PLL.

Host-Initiated Resume

The USB host initiates a Resume by driving the Data K bus state for 20 msec. When the USB module detects this action, it generates a USB Resume Interrupt. The software responds to this interrupt with the following steps:

1. If not running, configure USB PLL for 48 MHz, then enable PLL and wait until it becomes stable.
2. Clear the RIRQE bit.

Device-Initiated Resume (Remote Wake-Up)

To perform a device-initiated Resume, the software must perform the following steps:

1. Initiate a *wakeup*
    a. If not running, configure USB PLL for 48 MHz, then enable PLL and wait until it becomes stable.
    b. Set the USBIRQCTL.RIRQE bit in the USBIRQCTL register to enable device-initiated Resume.
    c. Set the WAKEUP bit to initiate the Resume.
    d. The USB module will count 0–5 msec and a USB Resume interrupt will be generated after 5 msec of continuous USB bus idle state, clearing the WAKEUP bit.
2. Attend to the Resume interrupt:
    a. Read the DEVRESUME bit in USBCS to determine if the Resume is device-initiated.
    b. Set and then clear the FORCEJ bit in USBCS.
    c. Set the SIGRESUME bit in USBCS to initiate remote wake-up signaling to the host.
    d. Wait for 1–15 msec before clearing SIGRESUME.

Firmware Design

The Z8F6482 Development Board includes a USB interface that provides power to the board and a connection to a USB host. This application note uses this USB interface with no additional components. The main application initializes the clock for the USB module (48 MHz) and the system clock (16MHz).

The USB firmware is divided into the following two modules:

- USB driver
- Device Class Driver.

The USB driver refers to the set of routines necessary to control the Z8F6482 MCU's USB peripheral. The device class driver provides a framework for implementing a virtual serial port that is able to communicate with a host PC using a serial terminal program.

Note: The endian converter, endpoint data handler, and USB device request functions can be found in the `usb.h` file. The device information block, CDC device class driver, device descriptor, and configuration descriptor functions can be found in the `cdc_conf.c` file.

USB Driver

The USB driver module includes basic routines to drive the USB peripheral to communicate with a host USB. Table 7 lists routines included in this module.

Table 7. USB Driver Module Routines

Function Name	Returns	Description
USB_Init	void	Initializes the USB module
USB_EpInit	void	Initializes an endpoint buffer
USB_ResetAllEp	void	Resets all endpoint buffers to their invalid state, leaving endpoint 0 which should always be valid.
USB_Receive	UINT8	Enables and endpoint for data reception
USB_Transmit	UINT8	Initiates data transmission on an endpoint

Endian Converter

The USB protocol is defined in little endian format. Therefore, all hosts are implemented in little endian format. The Z8F6482 is in big endian format. To maintain compatibility, the USB driver module defines the following macros for easier byte order swapping.

```
// Changes byte order from Big Endian (Zilog MCU) to Little Endian
// (USB)
#define SWAP(w)   ((((w) & 0xFF) << 8) | (((w) >> 8) & 0xFF))
// High and Low byte is according to Little Endian (USB) format
#define GET_LOWBYTE(w)   ((UINT8)((w) & 0xFF00) >> 8)
```

```
#define GET_HIGHBYTE(w)   ((UINT8)((w) & 0x00FF))
```

Endpoint Data Handler

An endpoint data handler is used to temporarily store OUT/IN transaction data for software usage. In an OUT transaction, data read from the module's endpoint buffer memory gets stored in the endpoint data handler by the interrupt routine for use by software. In an IN transaction, this handler is used by the software to temporarily store the data to be transmitted before it writes to the module's endpoint buffer memory when it becomes available. The endpoint data handler is defined as part of the device information block.

```
typedef struct USBEP_DATA
{
    UINT8 ucEpStatus;               // EP Status
    UINT8 ucEpBufferSize;           // EP Buffer Size
    UINT8 *aucBuffer;               // EP Buffer Data
    UINT16 ucEpLen;                 // EP Buffer Data Length
    // Function to call when EP is done processing Rx/Tx
    void (*afTxDone)(struct USBEP_DATA *pEpInfo);
}USBEP_DATA;
```

USB Device Request

During a control transfer, the host sends a SETUP DATA PACKET via Endpoint 0. This 8-byte data packet is stored in the USBSU0-7 registers of the USB module. The software reads these registers and stores the data in the USB device request data structure for easier access.

```
typedef struct
{
   UINT8 bmRequestType;           // Characteristics of request
   UINT8 bRequest;                // Specific request
   UINT16 wValue;                 // Varies depending on request
   UINT16 wIndex;                 // Varies depending on request
   UINT16 wLength;                // Number of bytes to requested
                                  //from device
}USBDVC_RQST;
```

This application note supports the following USB requests:

- GET_STATUS
- SET_ADDRESS
- GET_DESCRIPTOR
- SET_CONFIGURATION

Device Information Block

The device information block contains the device address (given by the host during enumeration), the device state, endpoint buffer table, and a series of pointers to the descriptor tables as specified in USB 2.0 specifications. The device class driver (Communications Device Class, CDC driver, in this application note) is responsible for declaring and initializing the device information block.

The CDC device class modules defines the data for the device information block, leaving ucDeviceAddress and tEpBuffer[] as NULL. These parameters will be initialized during the USB enumeration process.

```
USBDVC_BLOCK tUSB_Device = {
    0,                  // ucDeviceAddress
    DVCSTATE_IDLE,      // ucDeviceState
    {                   // tEpBuffer
        { USBEP_INVALID, EPIDX_IN0, NULLPTR, 0, NULLPTR },
        { USBEP_INVALID, EPIDX_IN1, NULLPTR, 0, NULLPTR },
        { USBEP_INVALID, EPIDX_IN2, NULLPTR, 0, NULLPTR },
        { USBEP_INVALID, EPIDX_IN3, NULLPTR, 0, NULLPTR },
        { USBEP_INVALID, EPIDX_OUT0, NULLPTR, 0, NULLPTR },
        { USBEP_INVALID, EPIDX_OUT1, NULLPTR, 0, NULLPTR },
        { USBEP_INVALID, EPIDX_OUT2, NULLPTR, 0, NULLPTR },
        { USBEP_INVALID, EPIDX_OUT3, NULLPTR, 0, NULLPTR },
    },
    // USB Configuration
   &DeviceDescr,
   &ConfigDescr,
   &PackedStrDescr,
   1,

   // Function to call when a class-specific device request is
   // received
   CDC_ClassRequest,
  // Function to call when a SET_CONFIGURATION device request is
   // received
   CDC_SetConfig
};
```

CDC Device Class Driver

The CDC device class driver module includes basic endpoint handler routines to implement a virtual serial port. It also includes wrapper routines for getch() and putch() to enable use of printf() and scanf() with the CDC device class. Table 8 lists the routines included in this module.

Table 8. CDC Device Class Driver Routines

Function Name	Returns	Description
CDC_ClassRequest	void	Handles class-specific USB device request/s
CDC_SetConfig	void	Handles class-specific endpoint configuration
CDC_Out1Hdlr	void	Processes received data
ucCDC_IsConnected	UINT8	Returns CDC status (connected/ disconnected)
putch	UINT8	Wrapper routine to use CDC for printf()
getch	UINT8	Wrapper routine to use CDC for scanf()

Device Descriptor

The device descriptor describes general information about the USB device. This includes the product ID, vendor ID, USB specification release number, device class code, and other general information. The host requests this information during the enumeration process when the host sends a GET_DESCRIPTOR request with a DEVICE descriptor type.

The device descriptor used in this application note specifies CDC as the device class.

```
const USBDVC_DESCR DeviceDescr =
{
    sizeof(USBDVC_DESCR),               // bLength
    USBDESCR_DEVICE,                    // bDescriptorType = 0x01
    SWAP(USB_SPECIFICATION),            // bcdUSB = 0x0200
    CDC_DEVICE_CLASS,                   // bDeviceClass = 0x02
    CDC_SUBCLASS_ACM,                   // bDeviceSubClass = 0x02
    CDC_PROTOCOL_NONE,                  // bDeviceProtocol = 0x00
    EP0_PACKETSIZE,                     // bMaxPacketSize0 = 0x40
    SWAP(USB_VENDOR_ID),                // idVendor = 0xFFFF
    SWAP(USB_PRODUCT_ID),               // idProduct = 0xFFFF
    SWAP(USB_RELEASE_NUMBER),           // bcdDevice = 0x0001
    USB_MANUFACTURER_IDX,               // iManufacturer
    USB_PRODUCT_IDX,                    // iProduct
    USB_SERIALNUMBER_IDX,               // iSerialNumber
    1                                   // bNumConfigurations
};
```

Configuration Descriptor

The configuration descriptor describes the device's configuration and interface settings, alternate settings, and their endpoints. The host requests this information during the enumeration process when the host sends a GET_DESCRIPTOR request with a CONFIGURATION descriptor type.

The USB specification defines a set of standard configuration descriptors; while the CDC specification defines additional class-specific descriptors. Table 5 shows the sequence of a configuration descriptor for a generic CDC device that functions as a virtual serial port. This configuration descriptor is a combination of the USB and CDC specifications.

The configuration descriptor used in this application note shows a typical CDC device that can be used as a virtual serial port.

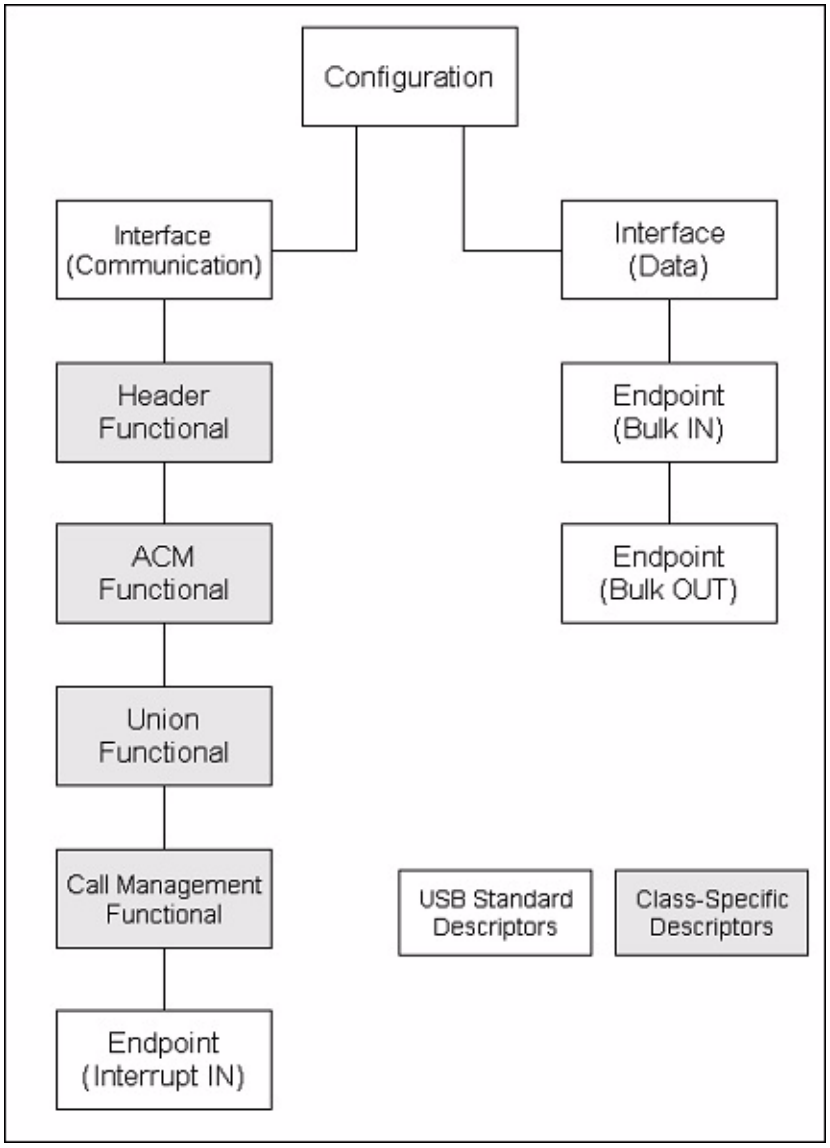

Figure 5. Communications Device Class Configuration Descriptor

```
// Configuration Descriptor
const USBDVC_CONFDESCR ConfigDescr =
{
    sizeof(USBDVC_CONFDESCR),       // bLength
    USBDESCR_CONFIG,                // bDescriptorType
    SWAP(CONFIG_TOTAL_LEN),         // wTotalLength
    2,                              // bNumInterfaces
    1,                              // bConfigurationValue
    0,                              // iConfiguration
    0x80 | (USB_SELF_POWER << 6) | (USB_REMOTE_WU << 5),
                                    // bmAttributes
    250                             // bMaxPower
};


// Interface Descriptor (Communication)
const USBDVC_IFDESCR InterfaceDescr =
{
    sizeof(USBDVC_IFDESCR),         // bLength
    USBDESCR_INTERFACE,             // bDescriptorType
    0,                              // bInterfaceNumber
    0,                              // bAlternateSetting
    1,                              // bNumEndpoints

    CDC_DEVICE_CLASS,               // bInterfaceClass
    CDC_SUBCLASS_ACM,               // bInterfaceSubClass
    CDC_PROTOCOL_NONE,              // bInterfaceProtocol
    0                               // iInterface
};
```

```
// Functional Descriptors
UINT8 COM_FuncDesc[] =
{
   // Header Functional
   0x05,                              // bFunctionLength
   CDC_DESCR_INTERFACE,               // bDescriptorType
   CDC_DESCRFUNC_HEADER,              // bDescriptorSubtype
   GET_HIGHBYTE(CDC_VERSION),         // bcdCDC
   GET_LOWBYTE(CDC_VERSION,           // ACM Functional
   0x04,                              // bFunctionLength
   CDC_DESCR_INTERFACE,               // bDescriptorType
   CDC_DESCRFUNC_ACM,                 // bDescriptorSubtype
   0x02,                              // bmCapabilities
                          // D7:D4 - Reserved
                          // D3 - 1: Supports NETWORK_CONNECTION
                          // notification
                          // D2 - 1: Supports SEND_BREAK
                          // D1 - 1: Supports GET/SET_LINE_CODING,
                          // SET_CONTROL_LINE_STATE, and
                         // SERIAL_STATE notification
                         // D0 - 1: Supports GET/SET_COMM_FEATURE,
                         // CLEAR_COMM_FEATURE
// Union Functional
   0x05,                              // bFunctionLength
   CDC_DESCR_INTERFACE,               // bDescriptorType
   CDC_DESCRFUNC_UNION,               // bDescriptorSubtype
   0x00,                              // bControlInterface
                                      //(Communication Class)
  0x01,                              // bSubordinateInterface0
                                     //(Data Class)
// Call Management Functional
   0x05,                              // bFunctionLength
   CDC_DESCR_INTERFACE,               // bDescriptorType
   CDC_DESCRFUNC_CALLMGMT,            // bDescriptorSubtype
   0x01,                              // bmCapabilities
                           // D7:D2 - Reserved
                           // D1 - 1: Device sends/receives call
                           // mgmt info via Data Class Interface
                           // - 0: Device sends/receives call
                           // mgmt info via Communications Class
                           // Interface
                          // D0 - 1: Device handles call mgmt
                          // itself
                          // - 0: Device does not handle call
                          // mgmt itself
   0x01                               // bDataInterface
};
```

```
// Endpoint(Interrupt IN) Descriptor
USBDVC_EPDESCR EpIn2Descr =
{
   sizeof(USBDVC_EPDESCR),          // bLength
   USBDESCR_ENDPOINT,               // bDescriptorType
   0x82,                            // bEndpointAddress: IN2
   3,                               // bmAttributes: Interrupt
   SWAP(EP0_PACKETSIZE),            // wMaxPacketSize
   100                              // bInterval
};

// Interface Descriptor (Data)
USBDVC_IFDESCR DataClassInterfaceDescr =
{
   sizeof(USBDVC_IFDESCR),         // bLength
   USBDESCR_INTERFACE,             // bDescriptorType
   1,                              // bInterfaceNumber
   0,                              // bAlternateSetting
   2,                              // bNumEndpoints
   CDC_DATACLASS_INTERFACE,        // bInterfaceClass
   CDC_SUBCLASS_NONE,              // bInterfaceSubClass
   CDC_PROTOCOL_NONE,              // bInterfaceProtocol
   0                               // iInterface
};


// Endpoint (Bulk IN) Descriptor
USBDVC_EPDESCR EpIn1Descr =
{
   sizeof(USBDVC_EPDESCR),          // bLength
   USBDESCR_ENDPOINT,               // bDescriptorType
   0x81,                            // bEndpointAddress: IN1
   2,                               // bmAttributes: Bulk
   SWAP(EP0_PACKETSIZE),            // wMaxPacketSize
   0                                // bInterval
};


// Endpoint (Bulk OUT) Descriptor
USBDVC_EPDESCR EpOut1Descr =
{
   sizeof(USBDVC_EPDESCR),          // bLength
   USBDESCR_ENDPOINT,               // bDescriptorType
   0x01,                            // bEndpointAddress: OUT1
   2,                               // bmAttributes: Bulk
   SWAP(EP0_PACKETSIZE),            // wMaxPacketSize
   0                                // bInterval
};
```

Equipment Used

The following equipment is used to build and test this application:

- Z8 Encore! XP Z8F6482 Development Board
- Zilog USB (or Ethernet) SmartCable
- USB A (Male) to Mini-B USB Cable
- RealTerm terminal simulator (can be downloaded for free at https://sourceforge.net/projects/realterm)

Testing and Demonstrating the Application

This section contains instructions for downloading, installing, and running the application.

Downloading Code to the Z8F6482 Development Board

1. Connect the USB SmartCable to the DBG terminal of the development board and connect the USB side to the development PC's USB port.
2. Connect the USB A (male) to Mini-B cable to P1 on the development board and connect the cable's other end to a PC's USB port to apply power to the development board.
3. Download and extract AN0411-SC01.zip from zilog.com.
4. Open ZDS II – Z8 Encore v5.2.2 (or later) IDE.
5. From the **File** menu, select **Open Project…**
6. Navigate to the extracted folder, select the `AN0411-DevKit.zdsproj` project and click **Open** to open the project.
7. From the **Build** menu, select **Rebuild All** to rebuild the project.
8. From the **Debug** menu, select **Download Code** to load the program to the MCU.
9. After programming is complete, from the **Debug** menu, select **Stop Debugging**.
10. Remove the USB A (male) to Mini-B cable to remove power from the board.
11. Disconnect the USB SmartCable from the DBG terminal of the development board.

Installing the Driver Software

1. Upon detecting the USB device, Windows attempts to install drivers for this device but fails. Open the **Device Manager** and search for **Unknown device**.

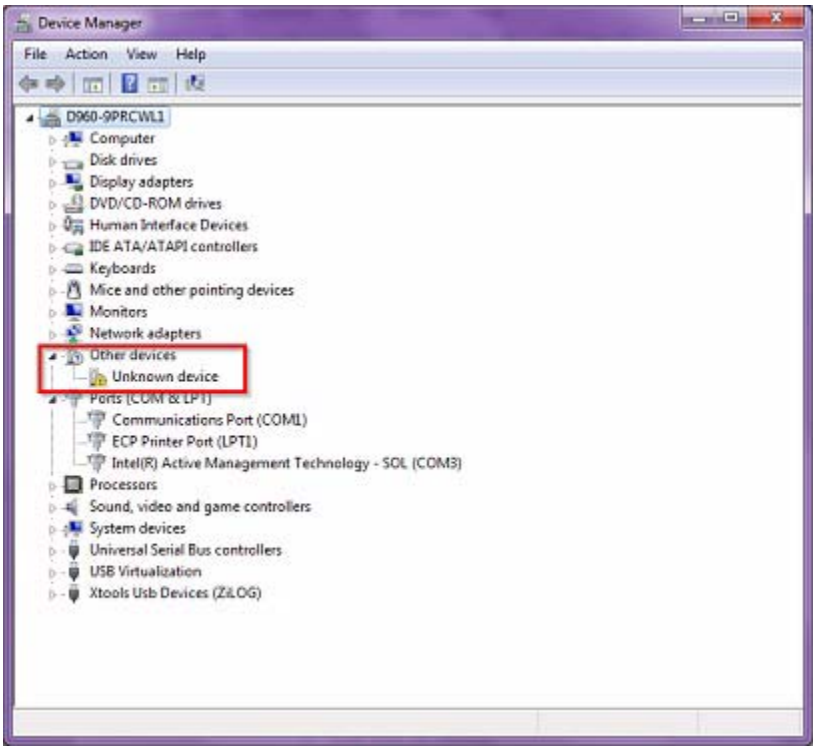

Figure 6. Device Manager

2. Verify that the Unknown device refers to the USB device in this application note.
   - Select and right click the Unknown device, then choose **Properties.**

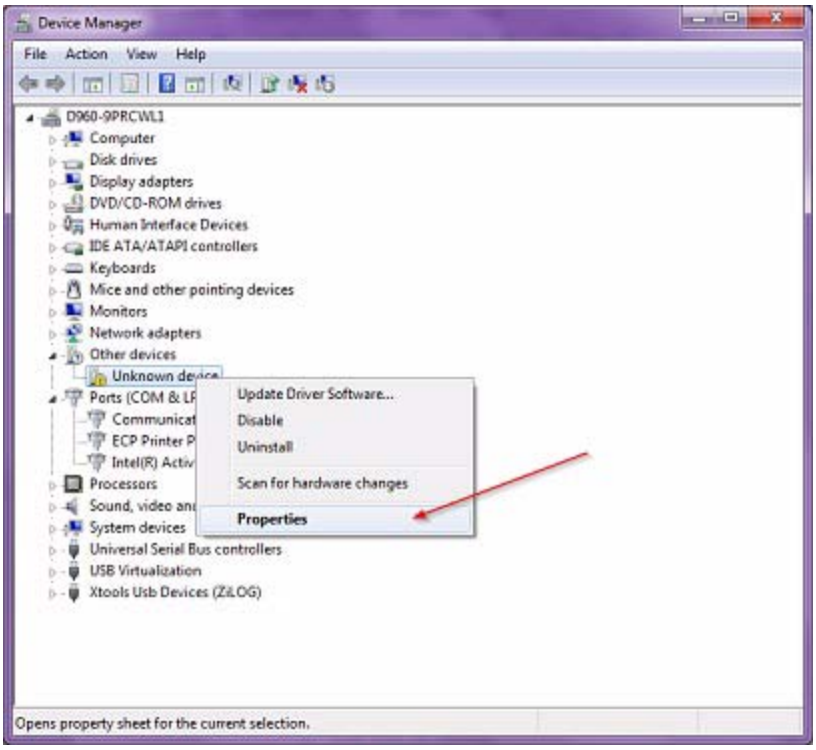

Figure 7. Displaying Properties of a Device

– A Device Properties dialog appears. Select the **Details** tab. Under **Property**, choose **Hardware Ids**. If the values list `USB\VID_FFFF&PID_FFFF&REV_0001` and `USB\VID_FFFF&PID_FFFF`, this is the correct device. Proceed to Step 3. Otherwise, click **Cancel** and go back to Step 1 to search for other Unknown devices.

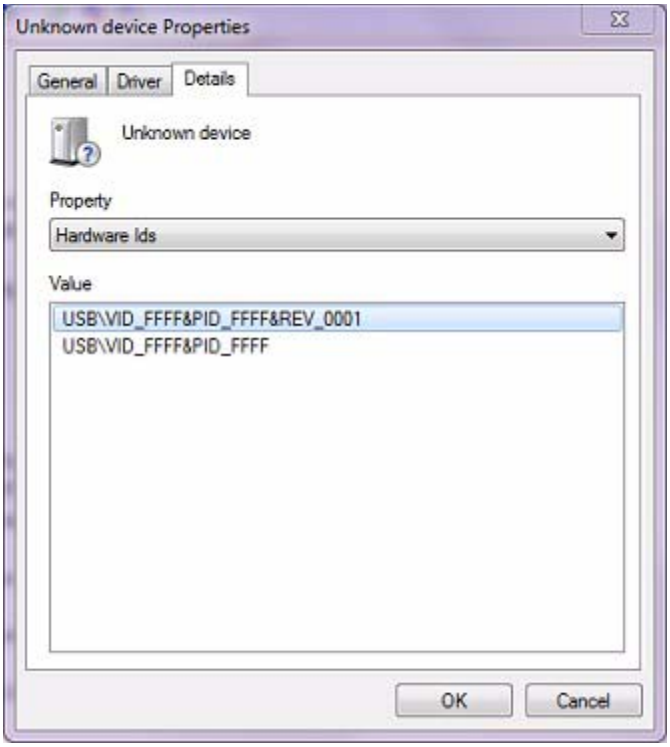

Figure 8. Displaying VID and PID of the Device

3. On the **Driver** tab, click **Update Driver…**

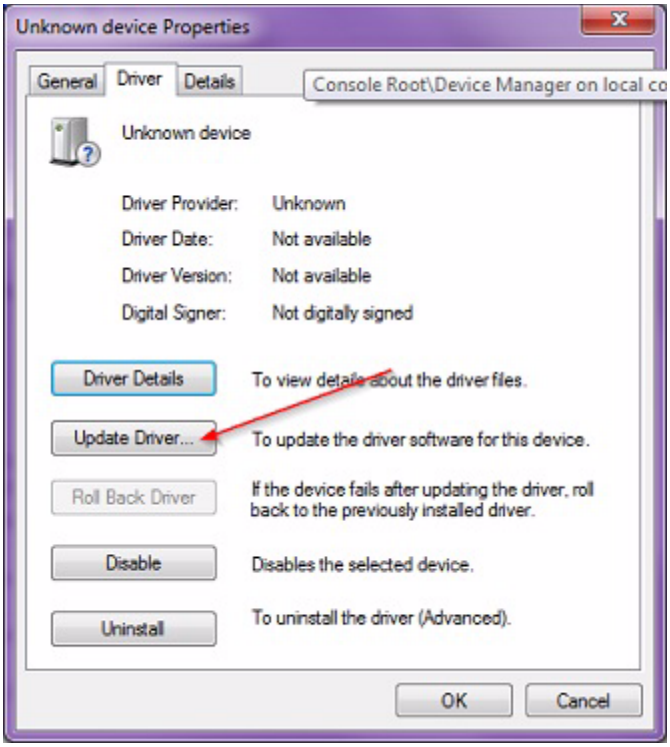

Figure 9. Update Driver Software

4. The **Update Driver Software** wizard appears. In response to the *How do you want to search for driver software?* question, choose **Browse my computer for driver software**.
5. On the **Browse for driver software on your computer** screen, click the **Browse** button.
6. A **Locate File** dialog appears. Locate the folder where the `AN0411-SC01.zip` file is extracted and click **OK**.
7. The window returns to the **Browse for driver software on your computer** screen with the directory path populated with the selected software. Click **Next**.

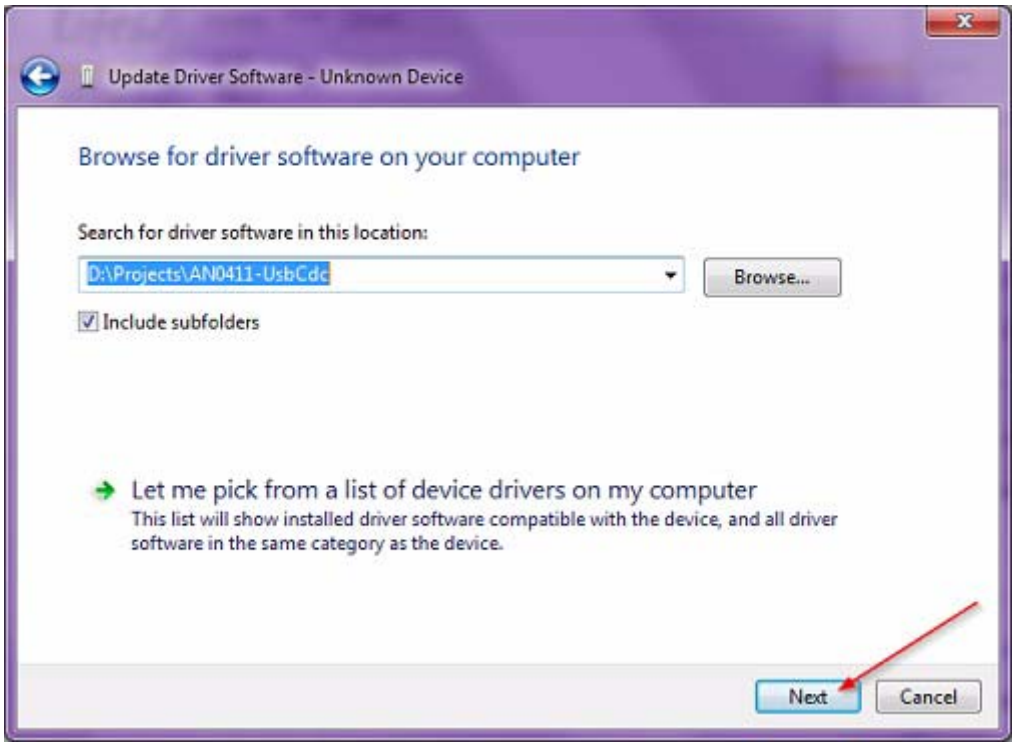

Figure 10. Browse for Driver Software on Your Computer

8. Windows Security displays a warning: *Windows can't verify the publisher of this driver software.* Click **Install this driver software anyway.**

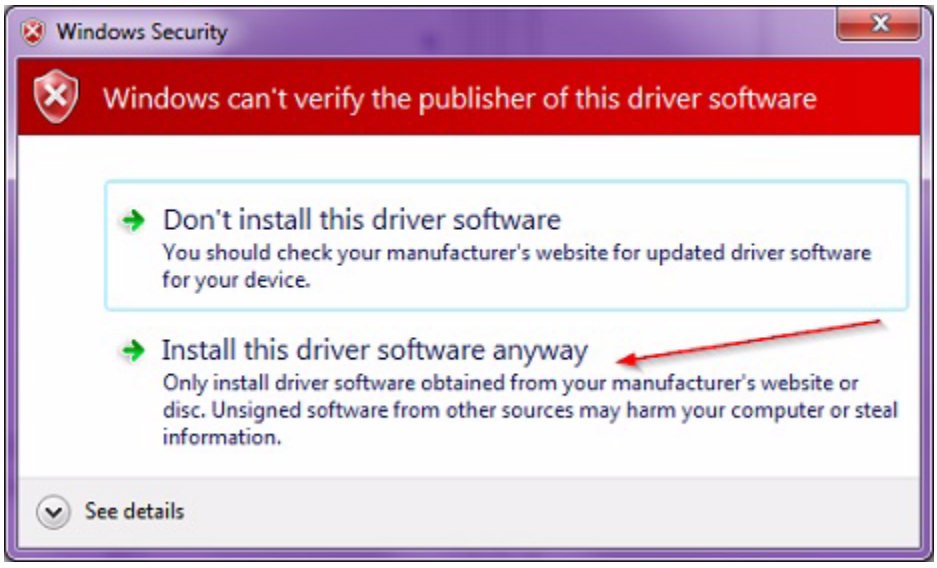

Figure 11. Windows Security Warning

9. The window returns to the Update Driver Software window, which displays the message, *Installing driver software….* Wait for the installation to complete.
10. When Windows finishes installing the driver, it displays the message, *Windows has successfully updated your driver software.*
11. Click **Close**. The Device Manager updates its display, showing *AN0411 USB Serial Port* under **Ports (COM & LPT)**, as shown in Figure 12. Note the COM number assigned to this device.

Note: The name displayed on the Device Manager depends on the `.inf` file used. If the accompanying `.inf` file from the AN0411-SC01 source code is used, the Device Manager displays **AN0411 USB Serial Port.** However, if Windows locates a previously-installed `.inf` file that works with this Application Note or includes a built-in USB serial port driver (for example, Windows 10) which automatically installs when a USB serial port device is detected, a different name will appear.

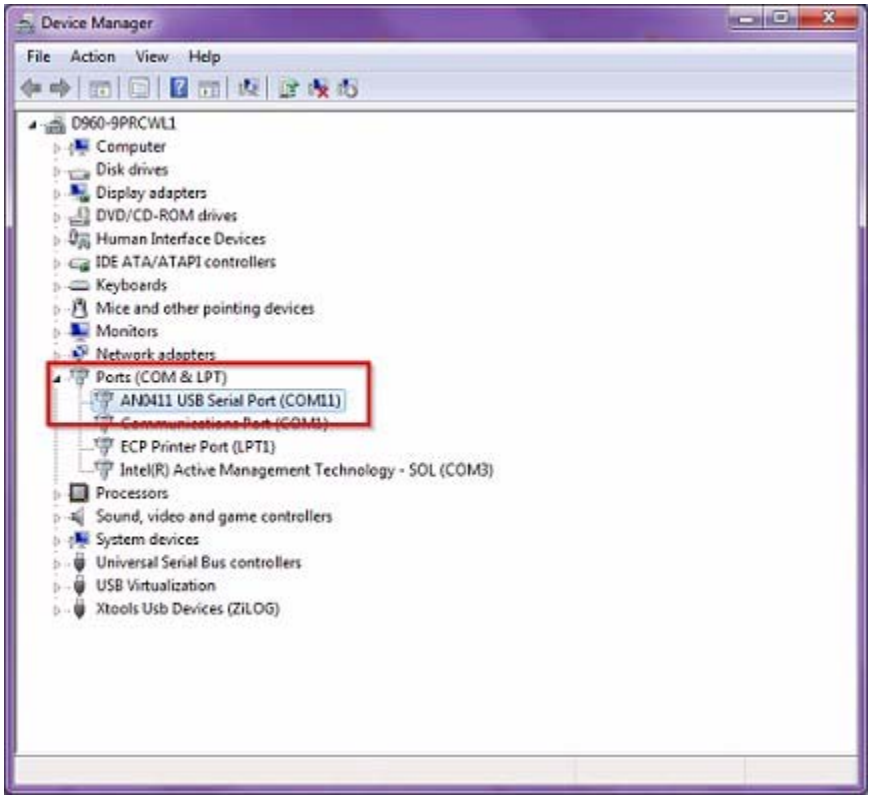

Figure 12. USB Serial Port

Running the Application

1. Ensure that the software code is programmed onto the development board by following the steps listed in the the Downloading Code to the Z8F6482 Development Board section on page 22.
2. Connect the USB A (male) to Mini-B cable to P1 on the development board. Connect the other end of the cable to a PC's USB port to apply power to the development board and provide a USB serial connection to the PC.
3. For Windows 10, the driver is automatically installed when the USB device is detected. Proceed to Step 4. For Windows 7 and Windows XP, follow the steps listed in the Installing the Driver Software section on page 23 prior to proceeding.

Note: After the Development Board with the USB cable is connected to the PC, wait for about 6-10 seconds before opening up the port in the RealTerm terminal.

4. Open RealTerm. On the **Display** tab, select the **Half Duplex** and **newLine mode** settings.

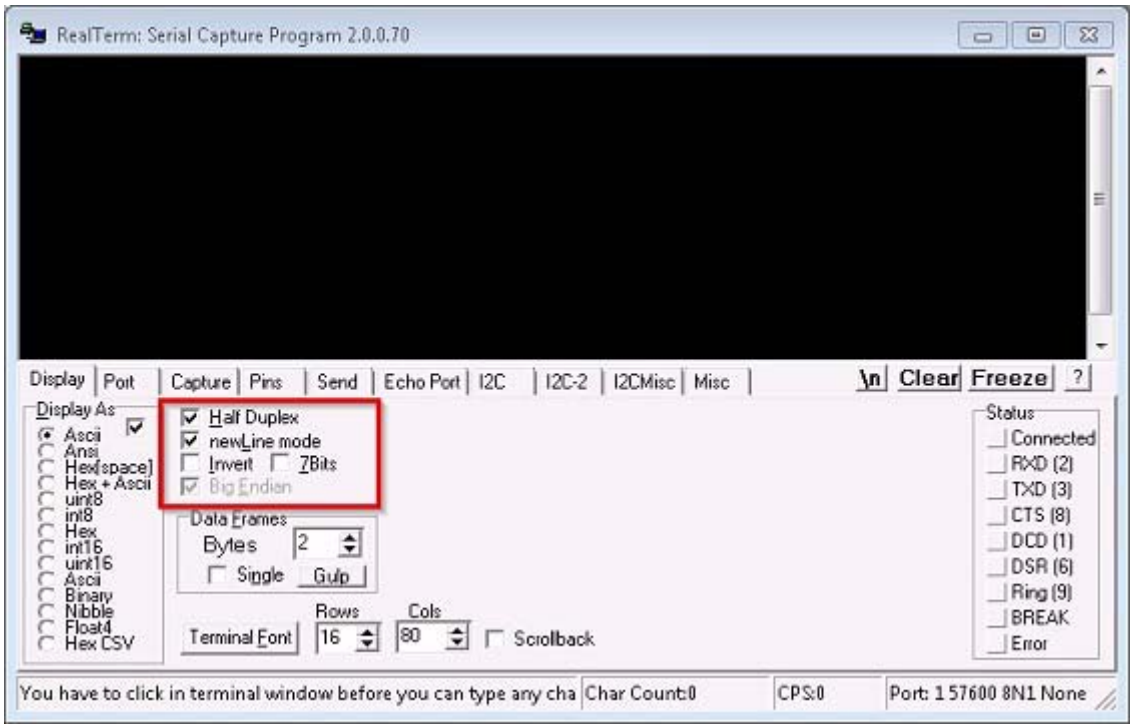

Figure 13. Configuring RealTerm – Display Tab

5. On the **Port** tab, select the Port number assigned to the USB serial port, then click **Open**.

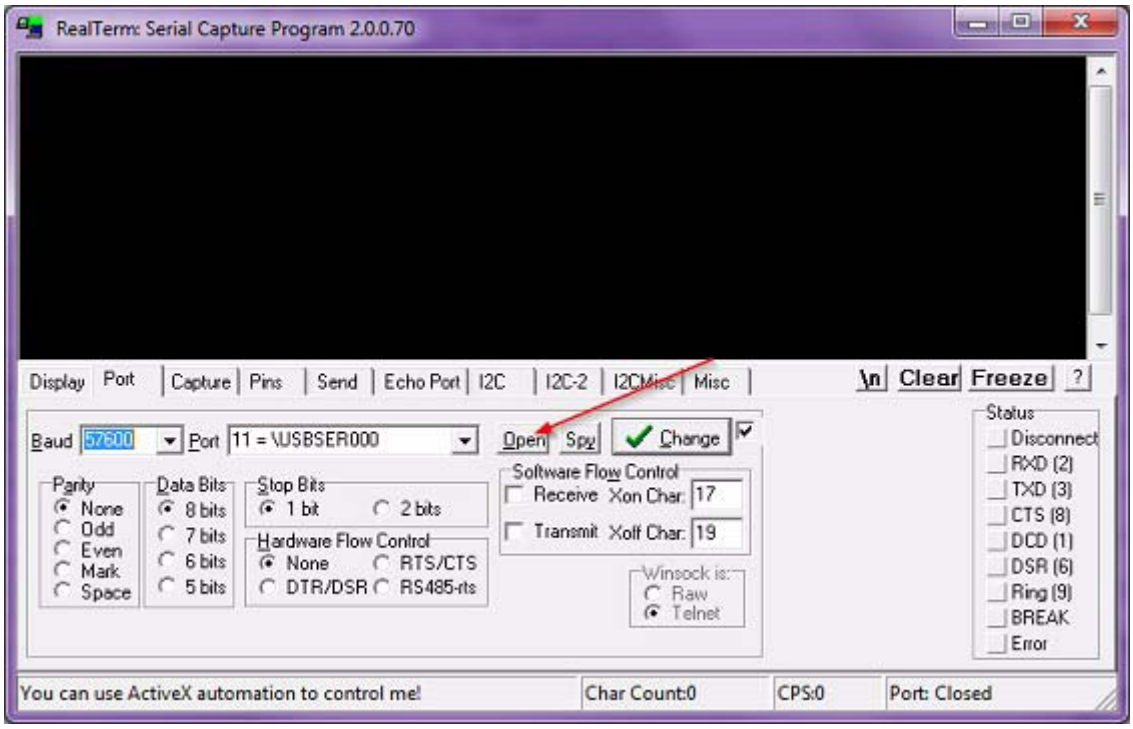

Figure 14. Configuring RealTerm – Port Tab

6. RealTerm displays a welcome message from the USB device.

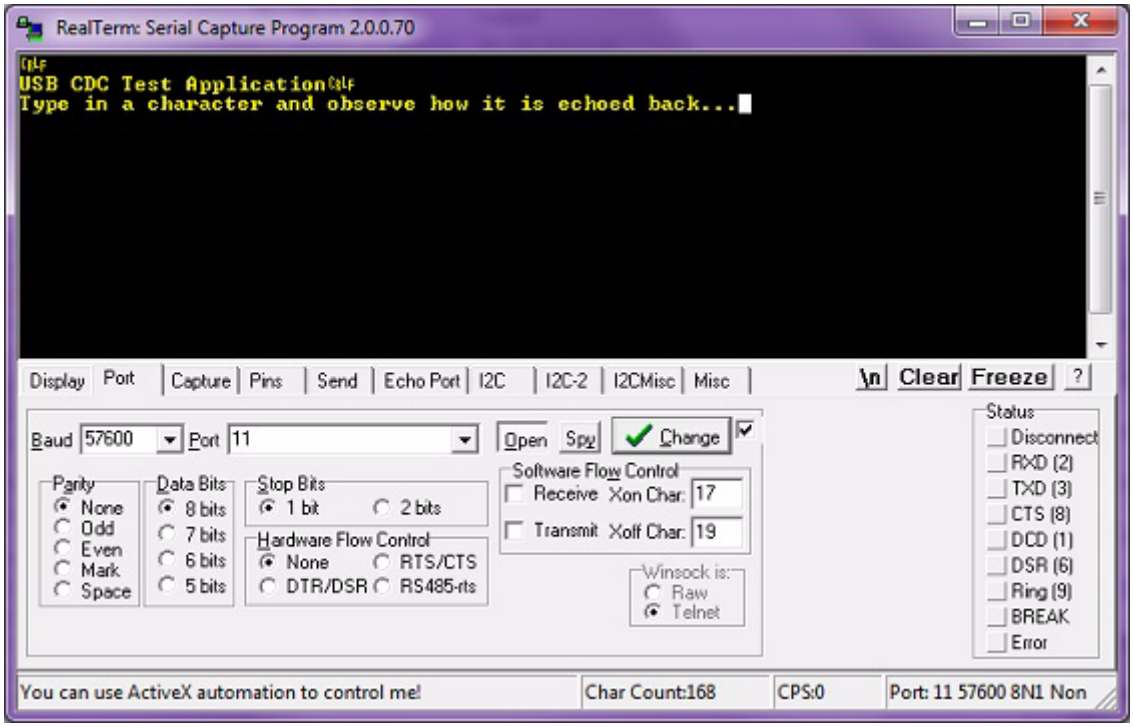

Figure 15. Welcome Message

7. Type in a character and observe how it is echoed back to the terminal. In this example, the red characters are from keyboard input and the yellow characters are data coming from the USB device.

Note: To reset the application, prior to pressing the **SW2** reset button on the Development Board, ensure that you close the RealTerm terminal port first by clicking the **Open** button on the **Port** tab and clearing the display. Then, press the **SW2** reset button and wait 6-10 seconds before opening the RealTerm port again.

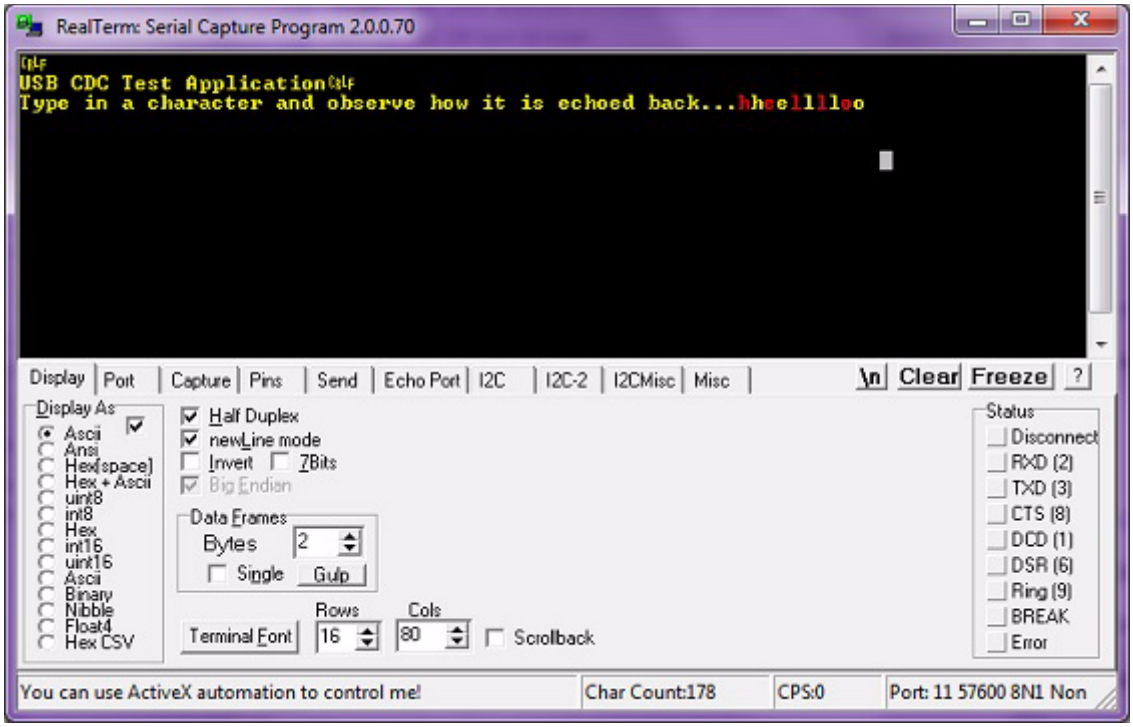

Figure 16. USB CDC Application

Summary

This application note provides sample software for using the Z8F6482 MCU's USB peripheral as a virtual serial port. This software is modular and easy to customize for any USB-based application.

References

The following documents/resources are associated with this Application Note:

- Z8 Encore XP F6482 Series Product Specification (PS0294)
- Z8 Encore XP F6482 Series Development Kit User Manual (UM0263)
- www.usb.org

Customer Support

To share comments, get your technical questions answered, or report issues you may be experiencing with our products, please visit Zilog's Technical Support page at http://support.zilog.com.

To learn more about this product, find additional documentation, or to discover other facets about Zilog product offerings, please visit the Zilog Knowledge Base at http://zilog.com/kb or consider participating in the Zilog Forum at http://zilog.com/forum.

This publication is subject to replacement by a later edition. To determine whether a later edition exists, please visit the Zilog website at http://www.zilog.com.

Warning: DO NOT USE THIS PRODUCT IN LIFE SUPPORT SYSTEMS.

LIFE SUPPORT POLICY

ZILOG'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS PRIOR WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF ZILOG CORPORATION.

As used herein

Life support devices or systems are devices which (a) are intended for surgical implant into the body, or (b) support or sustain life and whose failure to perform when properly used in accordance with instructions for use provided in the labeling can be reasonably expected to result in a significant injury to the user. A critical component is any component in a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system or to affect its safety or effectiveness.

Document Disclaimer

©2018 Zilog, Inc. All rights reserved. Information in this publication concerning the devices, applications, or technology described is intended to suggest possible uses and may be superseded. ZILOG, INC. DOES NOT ASSUME LIABILITY FOR OR PROVIDE A REPRESENTATION OF ACCURACY OF THE INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED IN THIS DOCUMENT. ZILOG ALSO DOES NOT ASSUME LIABILITY FOR INTELLECTUAL PROPERTY INFRINGEMENT RELATED IN ANY MANNER TO USE OF INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED HEREIN OR OTHERWISE. The information contained within this document has been verified according to the general principles of electrical and mechanical engineering.

Z8 Encore! XP is a trademark or registered trademark of Zilog, Inc. All other product or service names are the property of their respective owners.