Application Note: AN00124

USB CDC Class as Virtual Serial Port

This application note shows how to create a USB device compliant to the standard USB Communications Device Class (CDC) on an XMOS multicore microcontroller.

The code associated with this application note provides an example of using the XMOS USB Device Library (XUD) and associated USB class descriptors to provide a framework for the creation of a USB CDC device that implements Abstract Control Model (ACM).

This example USB CDC ACM implementation provides a Virtual Serial port running over high speed USB. The Virtual Serial port supports the standard requests associated with ACM model of the class.

A serial terminal program from host PC connects to virtual serial port and interacts with the application. This basic application demo implements a loopback of characters from the terminal to the XMOS device and back to the terminal. This application demo code demonstrates a simple way in which USB CDC class devices can easily be deployed using an xCORE-USB device.

Note: This application note provides a standard USB CDC class device and as a result does not require external drivers to run on Windows, Mac or Linux.

Required tools and libraries

- xTIMEcomposer Tools - Version 14.0.0
- XMOS USB library - Version 3.1.0

Required hardware

This application note is designed to run on an XMOS xCORE-USB series device.

The example code provided with the application has been implemented and tested on the xCORE-USB sliceKIT (XK-SK-U16-ST) but there is no dependancy on this board and it can be modified to run on any development board which uses an xCORE-USB series device.

Prerequisites

- This document assumes familiarity with the XMOS xCORE architecture, the Universal Serial Bus 2.0 Specification and related specifications, the XMOS tool chain and the xC language. Documentation related to these aspects which are not specific to this application note are linked to in the references appendix.
- For descriptions of XMOS related terms found in this document please see the XMOS Glossary[1].
- For the full API listing of the XMOS USB Device (XUD) Library please see the document XMOS USB Device (XUD) Library[2].
- For information on designing USB devices using the XUD library please see the XMOS USB Device Design Guide for reference[3].

---

[1]http://www.xmos.com/published/glossary
[2]http://www.xmos.com/published/xuddg
[3]http://www.xmos.com/published/xmos-usb-device-design-guide

 www.xmos.com

XM006845

1 Overview

1.1 Introduction

USB Communication Class is a composite USB device class that enables telecommunication devices like digital telephones, ISDN terminal adapters, etc and networking devices like ADSL modems, Ethernet adapters/hubs, etc to connect to a USB host machine. It specifies multiple models to support different types of communication devices. Abstract Control Model (ACM) is defined to support legacy modem devices and an advantage of ACM is the Serial emulation feature. Serial emulation of a USB device eases the development of host PC application, provides software compatibility with RS-232 based legacy devices, enables USB to RS-232 conversions and gives good abstraction over the USB for application developers.

In this application note, the USB CDC implementation on xCORE-USB device is explained in detail which will help you in two ways. First, it acts as reference for you to build your own USB CDC class device, second, it gives you an idea of how to use this virtual serial port code in your application.

The standard USB CDC class specification can be found in the USB-IF website.

(http://www.usb.org/developers/docs/devclass_docs/CDC1.2_WMC1.1_012011.zip)
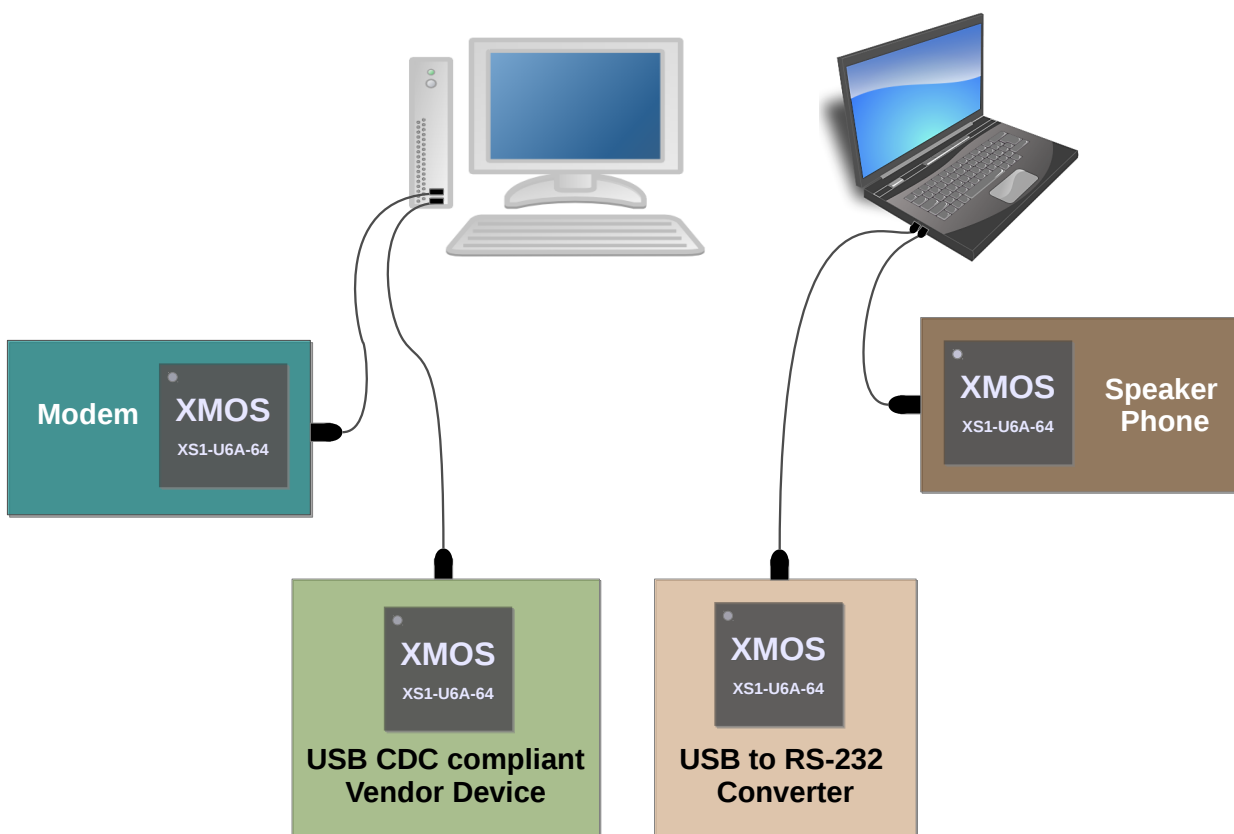
1.2 Block diagram

Figure 1: Block diagram of USB CDC applications

2   USB CDC Class application note

The example in this application note uses the XMOS USB device library and shows a simple program that enumerates a USB CDC Class device as virtual serial port in a host machine and provides a simple character loopback device.

For this USB CDC device application example, the system comprises four tasks running on separate logical cores of an xCORE-USB multicore microcontroller.

The tasks perform the following operations.

- A task containing the USB library functionality to communicate over USB.
- A task implementing Endpoint0 responding to both standard and CDC class-specific USB requests.
- A task implementing the data endpoints and notification endpoint of the CDC ACM class. It handles tx and rx buffers and provides interface for applications.
- A task implementing the application logic to interact with user over the virtual serial port.

These tasks communicate via the use of xCONNECT channels which allow data to be passed between application code running on separate logical cores. In this example, XC interfaces are used, which abstracts out the channel communication details with function level interface.

The following diagram shows the task and communication structure for this USB CDC class application example.
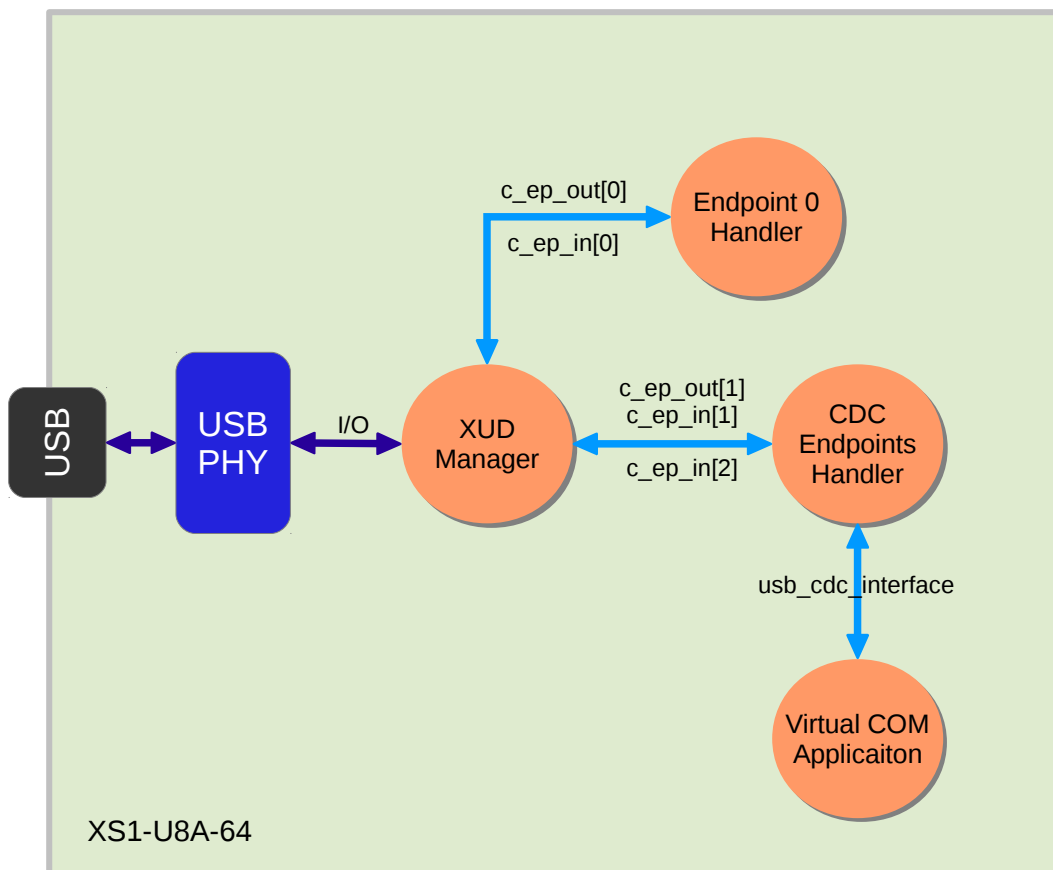

Figure 2: Task diagram of the USB CDC Virtual Serial Port example

2.1 Makefile additions for this example

To start using the USB library, you need to add **lib_usb** to your makefile:

```
USED_MODULES = ... lib_usb ...
```

You can then access the USB functions in your source code via the *usb.h* header file:

```
#include <usb.h>
```

2.2 Source code files

The example application consists of the following files:

```
main.xc
xud_cdc.xc
xud_cdc.h
```

xud_cdc.xc contains the CDC ACM implementation which includes the USB descriptors, endpoints handler functions and the xC interface (APIs) for application programs. The *xud_cdc.h* header is included in the *main.xc* to use the APIs exposed by *xud_cdc.xc*. The *main.xc* implements the application logic that interacts over the USB CDC link with a host terminal application.

2.3 Declaring resource and setting up the USB components

main.xc contains the application implementation for a device based on the USB CDC device class. There are some defines in it that are used to configure the XMOS USB device library. These are displayed below.

```
/* USB Endpoint Defines */
#define XUD_EP_COUNT_OUT   2     //Includes EP0 (1 OUT EP0 + 1 BULK OUT EP)
#define XUD_EP_COUNT_IN    3     //Includes EP0 (1 IN EP0 + 1 INTERRUPT IN EP + 1 BULK IN EP)
```

These defines describe the endpoints configuration for this device. This example has bi-directional communication with the host machine via the standard endpoint0 and three other endpoints for implementing the part of our CDC class device.

These defines are passed to the setup function for the USB library which is called from `main()`.

2.4 The application main() function

Below is the source code for the main function of this application, which is taken from the source file `main.xc`

```
int main() {
    /* Channels to communicate with USB endpoints */
    chan c_ep_out[XUD_EP_COUNT_OUT], c_ep_in[XUD_EP_COUNT_IN];
    /* Interface to communicate with USB CDC (Virtual Serial) */
    interface usb_cdc_interface cdc_data;

    par
    {
        on USB_TILE: xud(c_ep_out, XUD_EP_COUNT_OUT, c_ep_in, XUD_EP_COUNT_IN,
                        null, XUD_SPEED_HS, XUD_PWR_SELF);

        on USB_TILE: Endpoint0(c_ep_out[0], c_ep_in[0]);

        on USB_TILE: CdcEndpointsHandler(c_ep_in[1], c_ep_out[1], c_ep_in[2], cdc_data);

        on tile[1]: app_virtual_com(cdc_data);
    }
    return 0;
}
```

Looking at this in a more detail you can see the following:

- The par statement starts four separate tasks in parallel.
- There is a task to configure and execute the USB library: xud. This library call runs in an infinite loop and handles all the underlying USB communications and provides abstraction at the endpoints level.
- There is a task to startup and run the Endpoint0 code: Endpoint0(). It handles the control endpoint zero and must be run in a separate logical core inorder to promptly respond to the control requests from host.
- There is a task to handle all the other three endpoints required for the CDC class: CdcEnpointsHandler(). This function handles one bulk OUT and one bulk IN endpoints for data transmissions and one interrupt IN endpoint for sending notifications to host.
- There is a task to run the application logic that interacts with user over the virtual serial port: app_virtual_com().
- The define USB_TILE describes the tile on which the individual tasks will run.
- The xCONNECT communication channels and the xC interface *cdc_data* used for inter task communication are setup at the beginning of main() and passed on to respective tasks.
- The USB defines discussed earlier are passed into the function xud().

2.5 Configuring the USB Device ID

The USB ID values used for vendor ID, product ID and device version number are defined in the file xud_cdc.xc. These are used by the host machine to determine the vendor of the device (in this case XMOS) and the product plus the firmware version.

```
/* USB CDC device product defines */
#define BCD_DEVICE  0x0100
#define VENDOR_ID   0x20B1
#define PRODUCT_ID  0x0401
```

2.6 USB Descriptors

USB CDC class device has to support class-specific descriptors apart from the standard descriptors defined in the USB specifications. These class specific descriptors are customized according to the need of the USB CDC device. In the example application code, the descriptors implement the ACM model of the CDC class and are customized to suit a virtual serial port.

The following figure shows the descriptors used in the example code.
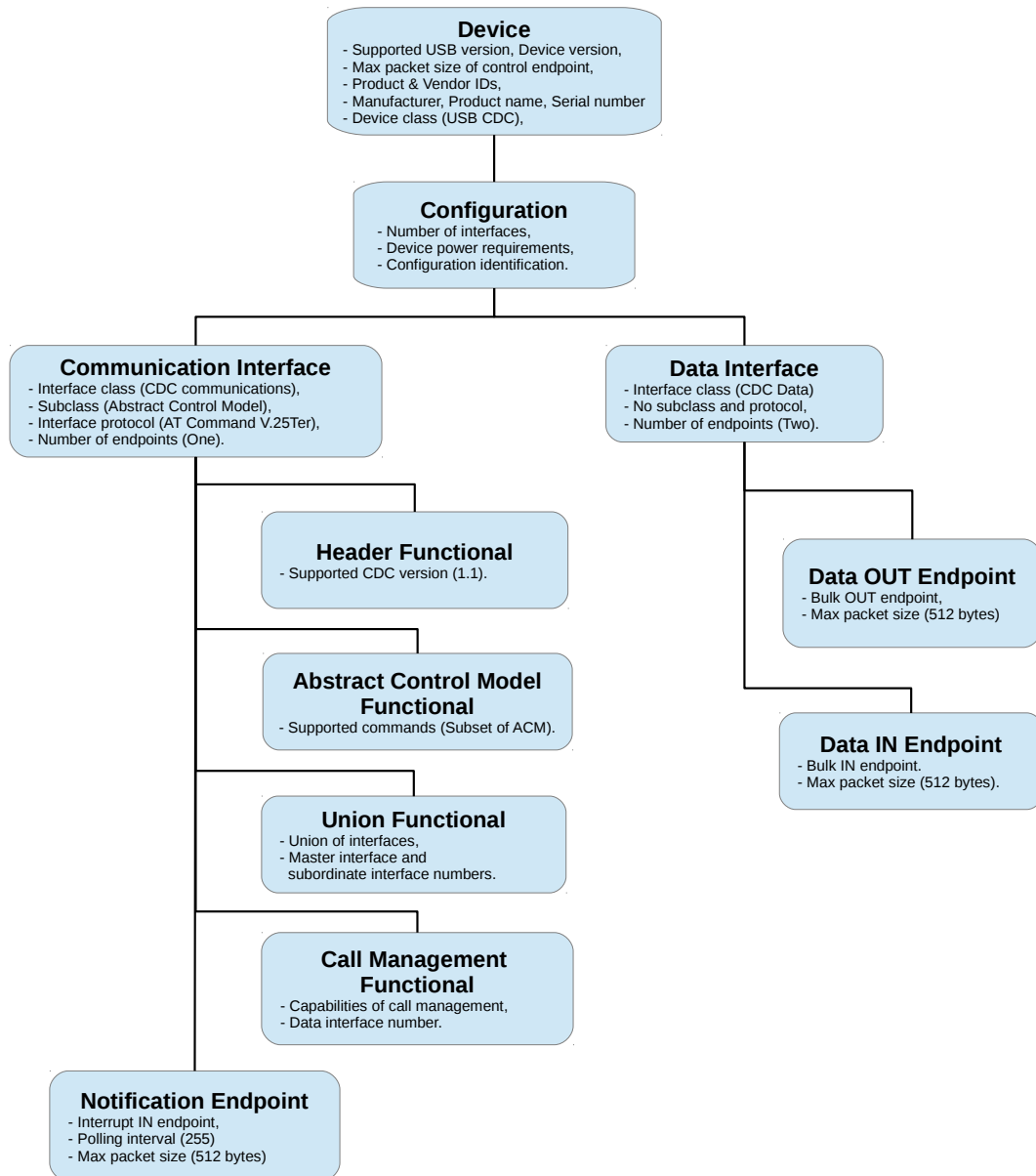

Figure 3: USB descriptors hierarchical structure of CDC example

2.6.1 USB Device Descriptor

`xud_cdc.xc` is where the standard USB device descriptor is declared for the CDC class device. Below is the structure which contains this descriptor. This will be requested by the host when the device is enumerated on the USB bus.

```
/* USB Device Descriptor */
static unsigned char devDesc[] =
{
    0x12,                   /* 0  bLength */
    USB_DESCTYPE_DEVICE,    /* 1  bdescriptorType - Device*/
    0x00,                   /* 2  bcdUSB version */
    0x02,                   /* 3  bcdUSB version */
    USB_CLASS_COMMUNICATIONS,/* 4  bDeviceClass - USB CDC Class */
    0x00,                   /* 5  bDeviceSubClass  - Specified by interface */
    0x00,                   /* 6  bDeviceProtocol  - Specified by interface */
    0x40,                   /* 7  bMaxPacketSize for EP0 - max = 64*/
    (VENDOR_ID & 0xFF),     /* 8  idVendor */
    (VENDOR_ID >> 8),       /* 9  idVendor */
    (PRODUCT_ID & 0xFF),    /* 10 idProduct */
    (PRODUCT_ID >> 8),      /* 11 idProduct */
    (BCD_DEVICE & 0xFF),    /* 12 bcdDevice */
    (BCD_DEVICE >> 8),      /* 13 bcdDevice */
    0x01,                   /* 14 iManufacturer - index of string*/
    0x02,                   /* 15 iProduct  - index of string*/
    0x03,                   /* 16 iSerialNumber  - index of string*/
    0x01                    /* 17 bNumConfigurations */
};
```

From this descriptor you can see that product, vendor and device firmware revision are all coded into this structure. This will allow the host machine to recognise the CDC device when it is connected to the USB bus.

2.6.2 USB Configuration Descriptor

The USB configuration descriptor is used to configure the device in terms of the device class and the endpoints setup. The hierarchy of descriptors under a configuration includes interfaces descriptors, class-specific descriptors and endpoints descriptors.

when a host requests a configuration descriptor, the entire configuration hierarchy including all the related descriptors are returned to the host. The following code shows the configuration hierarchy of the demo application.

```
/* USB Configuration Descriptor */
static unsigned char cfgDesc[] = {

  0x09,                       /* 0  bLength */
  USB_DESCTYPE_CONFIGURATION, /* 1  bDescriptortype - Configuration*/
  0x43, 0x00,                 /* 2  wTotalLength */
  0x02,                       /* 4  bNumInterfaces */
  0x01,                       /* 5  bConfigurationValue */
  0x04,                       /* 6  iConfiguration - index of string */
  0x80,                       /* 7  bmAttributes - Bus powered */
  0xC8,                       /* 8  bMaxPower - 400mA */

  /* CDC Communication interface */
  0x09,                       /* 0  bLength */
  USB_DESCTYPE_INTERFACE,     /* 1  bDescriptorType - Interface */
  0x00,                       /* 2  bInterfaceNumber - Interface 0 */
  0x00,                       /* 3  bAlternateSetting */
  0x01,                       /* 4  bNumEndpoints */
  USB_CLASS_COMMUNICATIONS,   /* 5  bInterfaceClass */
  USB_CDC_ACM_SUBCLASS,       /* 6  bInterfaceSubClass - Abstract Control Model */
  USB_CDC_AT_COMMAND_PROTOCOL,/* 7  bInterfaceProtocol - AT Command V.250 protocol */
  0x00,                       /* 8  iInterface - No string descriptor */

  /* Header Functional descriptor */
  0x05,                       /* 0  bLength */
  USB_DESCTYPE_CS_INTERFACE,  /* 1  bDescriptortype, CS_INTERFACE */
  0x00,                       /* 2  bDescriptorsubtype, HEADER */
  0x10, 0x01,                 /* 3  bcdCDC */

  /* ACM Functional descriptor */
  0x04,                       /* 0  bLength */
  USB_DESCTYPE_CS_INTERFACE,  /* 1  bDescriptortype, CS_INTERFACE */
  0x02,                       /* 2  bDescriptorsubtype, ABSTRACT CONTROL MANAGEMENT */
  0x02,                       /* 3  bmCapabilities: Supports subset of ACM commands */

  /* Union Functional descriptor */
  0x05,                       /* 0  bLength */
  USB_DESCTYPE_CS_INTERFACE,/* 1  bDescriptortype, CS_INTERFACE */
  0x06,                       /* 2  bDescriptorsubtype, UNION */
  0x00,                       /* 3  bControlInterface - Interface 0 */
  0x01,                       /* 4  bSubordinateInterface0 - Interface 1 */

  /* Call Management Functional descriptor */
  0x05,                       /* 0  bLength */
  USB_DESCTYPE_CS_INTERFACE,/* 1  bDescriptortype, CS_INTERFACE */
  0x01,                       /* 2  bDescriptorsubtype, CALL MANAGEMENT */
  0x03,                       /* 3  bmCapabilities, DIY */
  0x01,                       /* 4  bDataInterface */

  /* Notification Endpoint descriptor */
  0x07,                            /* 0  bLength */
  USB_DESCTYPE_ENDPOINT,           /* 1  bDescriptorType */
  (CDC_NOTIFICATION_EP_NUM | 0x80),/* 2  bEndpointAddress */
  0x03,                            /* 3  bmAttributes */
  0x40,                            /* 4  wMaxPacketSize - Low */
  0x00,                            /* 5  wMaxPacketSize - High */
  0xFF,                            /* 6  bInterval */

  /* CDC Data interface */
  0x09,                       /* 0  bLength */
  USB_DESCTYPE_INTERFACE,     /* 1  bDescriptorType */
  0x01,                       /* 2  bInterfacecNumber */
  0x00,                       /* 3  bAlternateSetting */
```

```
0x02,                    /* 4  bNumEndpoints */
USB_CLASS_CDC_DATA,      /* 5  bInterfaceClass */
0x00,                    /* 6  bInterfaceSubClass */
0x00,                    /* 7  bInterfaceProtocol*/
0x00,                    /* 8  iInterface - No string descriptor*/

/* Data OUT Endpoint descriptor */
0x07,                    /* 0  bLength */
USB_DESCTYPE_ENDPOINT,   /* 1  bDescriptorType */
CDC_DATA_RX_EP_NUM,      /* 2  bEndpointAddress */
0x02,                    /* 3  bmAttributes */
0x00,                    /* 4  wMaxPacketSize - Low */
0x02,                    /* 5  wMaxPacketSize - High */
0x00,                    /* 6  bInterval */

/* Data IN Endpoint descriptor */
0x07,                    /* 0  bLength */
USB_DESCTYPE_ENDPOINT,   /* 1  bDescriptorType */
(CDC_DATA_TX_EP_NUM | 0x80),/* 2  bEndpointAddress */
0x02,                    /* 3  bmAttributes */
0x00,                    /* 4  wMaxPacketSize - Low byte */
0x02,                    /* 5  wMaxPacketSize - High byte */
0x01                     /* 6  bInterval */
};
```

The configuration descriptor tells host about the power requirements of the device and the number of interfaces it supports.

The interface descriptors describe on how the host should communicate with the device in the class level. There are two interface descriptors in a USB CDC device.

The **CDC Communication interface** descriptor is for device management. You can see from the code that the device uses Abstract Control Model and supports AT Command V.25ter protocol. Though this example device doesn't support AT V.25ter protocol, it is mentioned to make the device compatible with standard host drivers. This interface has subordinate descriptors like CDC functional descriptors and a notification endpoint descriptor. The class-specific functional descriptors are discussed in detail in the next section. The notification endpoint is an interrupt IN endpoint and is used to report device's serial state to the host. This endpoint is not used in this example application but will be employed when bridging a UART to the USB Virtual COM port.

The **CDC Data interface** descriptor defines the interface for data transmission and reception between host and device. This interface has two endpoints, one bulk OUT endpoint for data transmissions from host to device and one bulk IN endpoint for data transmissions from device to host.

```
unsafe{
```

The above code from the endpoint descriptors shows that the maximum packet size of these endpoints to be 512 bytes (0x200) which is suited for applications requiring high data throughput.

2.6.3   USB CDC Functional Descriptor

Functional descriptors describe the content of class-specific information within the Communication Class interface. The 'USB_DESCTYPE_CS_INTERFACE' define is used in the descriptor structures to identify them. There are four functional descriptors used in this CDC example. They are:

1. Header functional descriptor.
2. ACM functional descriptor.
3. Union functional descriptor.
4. Call management functional descriptor.

Header functional descriptor mentions the version of the CDC specification the interface compiles with and it is shown below as found in the *cfgDesc[]* structure.

```
0x00,                       /* 3  bControlInterface - Interface 0 */
0x01,                       /* 4  bSubordinateInterface0 - Interface 1 */
```

Note: The CDC version number (1.10) is mentioned as BCD in little endian format.

ACM functional descriptor tells the class-specific commands and notifications supported by the CDC device. The application code supports a subset of commands corresponding to ACM subclass and thus the bit D1 is set to 1 in *bmCapabilities* field of the descriptor as shown below

```
0x01,                       /* 2  bDescriptorsubtype, CALL MANAGEMENT */
0x03,                       /* 3  bmCapabilities, DIY */
0x01,                       /* 4  bDataInterface */
```

Union functional descriptor groups the interfaces that forms a CDC functional unit. It specifies one of the interfaces as master to handle control messages of the unit. In the CDC example, the Communication Class interface acts as master and the Data Class interface acts as subordinate and together forming a single functional unit.

Call management functional descriptor decides on how the device manages calls. The bit fields D0 and D1 of *bmCapabilities* are set to one in the descriptor to tell host driver that the device handles call management by itself and it could even use Data class interface for that purpose. The below code shows that configuration.

```
/* CDC Data interface */
0x09,                     /* 0  bLength */
USB_DESCTYPE_INTERFACE,   /* 1  bDescriptorType */
0x01,                     /* 2  bInterfacecNumber */
```

2.6.4   USB String Descriptors

String descriptors provide human readable information for your device and you can configure them with your USB product information. The descriptors are placed in an array as shown in the below code.

```
/* String table - unsafe as accessed via shared memory */
static char * unsafe stringDescriptors[]=
{
  "\x09\x04",             /* Language ID string (US English) */
  "XMOS",                 /* iManufacturer */
  "CDC Virtual COM Port", /* iProduct */
  "0123456789"            /* iSerialNumber */
  "Config",               /* iConfiguration string */
};
```

The XMOS USB library will take care of encoding the strings into Unicode and structures the content into USB string descriptor format.

2.7   USB Standard and Class-Specific requests

In *xud_cdc.xc* there is a function *Endpoint0()* which handles all the USB control requests sent by host to control endpoint 0. USB control requests includes both standard USB requests and the CDC class-specific requests.

In *Endpoint0()* function, a USB request is received as a setup packet by calling *USB_GetSetupPacket()* library function. The setup packet structure is then examined to distinguish between standard and class-specific requests.

The XMOS USB library provides a function *USB_StandardRequests()* to handle the standard USB requests. This function is called with setup packet and descriptors structures as shown below

```
/* Returns  XUD_RES_OKAY if handled okay,
 *          XUD_RES_ERR if request was not handled (STALLed),
 *          XUD_RES_RST for USB Reset */
unsafe{
result = USB_StandardRequests(ep0_out, ep0_in, devDesc,
         sizeof(devDesc), cfgDesc, sizeof(cfgDesc),
         null, 0, null, 0, stringDescriptors, sizeof(stringDescriptors)/sizeof(stringDescriptors[0]),
         sp, usbBusSpeed);
 }
```

The CDC Communication interface uses endpoint 0 as management element and receives class-specific control requests on it. The following code shows how the class-specific requests are filtered and passed to a function *ControlInterfaceClassRequests()* for further handling.

```
switch(bmRequestType)
{
    /* Direction: Device-to-host and Host-to-device
     * Type: Class
     * Recipient: Interface
     */
    case USB_BMREQ_H2D_CLASS_INT:
    case USB_BMREQ_D2H_CLASS_INT:

        /* Inspect for CDC Communications Class interface num */
        if(sp.wIndex == 0)
        {
            /* Returns  XUD_RES_OKAY if handled,
             *          XUD_RES_ERR if not handled,
             *          XUD_RES_RST for bus reset */
            result = ControlInterfaceClassRequests(ep0_out, ep0_in, sp);
        }
        break;
}
```

The *ControlInterfaceClassRequests()* function handles a subset of CDC ACM requests which are defined in the *xud_cdc.xc* as follows:

```
/* CDC Communications Class requests */
#define CDC_SET_LINE_CODING          0x20
#define CDC_GET_LINE_CODING          0x21
#define CDC_SET_CONTROL_LINE_STATE   0x22
#define CDC_SEND_BREAK               0x23
```

In virtual serial port, the above commands are used to set and get serial port parameters like baud rate, parity, stop bits etc and also to emulate hardware flow control using DTR (Data Terminal Ready) and RTS (Request To Send) signals.

You can use the functions *ControlInterfaceClassRequests()* and the *Endpoint0()* as reference to handle more commands of the subclass or you can even implement a different model/subclass for your USB CDC device.

2.8   Data handling

The two bulk data endpoints of the CDC Data interface are handled by the task *CdcEndpointsHandler()* present in *xud_cdc.xc*. As there is no subprotocol used, the bytes received through these endpoints represent the raw data sent from a host terminal software. This data is handled using a double buffer mechanism and hence increases the performance of the device.

To handle asynchronous communication over two endpoints, events are used by means of select statements as shown in the following piece of code from *CdcEndpointsHandler()* task.

```
select
{
  case XUD_GetData_Select(c_epbulk_out, epbulk_out, length, result):

    if(result == XUD_RES_OKAY)
    {
        /* Received some data */
        rxLen[!readBufId] = length;

        /* Check if application has completed reading the read buffer */
        if(rxLen[readBufId] == 0) {
            /* Switch buffers */
            readBufId = !readBufId;
            readIndex = 0;
            /* Make the OUT endpoint ready to receive data */
            XUD_SetReady_Out(epbulk_out, rxBuf[!readBufId]);
        } else {
            /* Application is still reading the read buffer
             * Say that another buffer is also waiting to be read */
            readWaiting = 1;
        }
    } else {
        XUD_SetReady_Out(epbulk_out, rxBuf[!readBufId]);
    }
    break;

  case XUD_SetData_Select(c_epbulk_in, epbulk_in, result):

    /* Packet sent successfully when result in XUD_RES_OKAY */
    if (0 != txLen) {
        /* Data available to send to Host */
        XUD_SetReady_In(epbulk_in, txBuf[writeBufId], txLen);
        /* Switch write buffers */
        writeBufId = !writeBufId;
        txLen = 0;
    } else {
        writeWaiting = 1;
    }

    break;
```

When OUT endpoint receives data, an event is triggered and the *XUD_GetData_Select()* case is executed. Similary, when IN endpoint completes sending data to host the *XUD_SetData_Select()* case is executed. This event driven approach not only handles multiple endpoints but also provides way to include other

events and build more logic into the task.

2.9   Application interface

The application interface is the set of functions defined as xC interface that enables application tasks to send/receive data over the USB CDC endpoints. This API funtions abstract out all the buffering implementation details done at the endpoint level for data communications. This xC interface is declared in *xud_cdc.h* file and it is shown below.

```
interface usb_cdc_interface {

    [[guarded]] void put_char(char byte);

    [[guarded]] char get_char(void);

    [[guarded]] int write(unsigned char data[], REFERENCE_PARAM(unsigned, length));

    [[guarded]] int read(unsigned char data[], REFERENCE_PARAM(unsigned, count));

    int available_bytes(void);

    void flush_buffer(void);
};
```

These interface functions pass arguments and return values over xCONNECT channels and provides well defined inter-task communication. The server side of these functions are defined under select case statements in the *CdcEndpointsHandler()* task.

In the example code, main.xc has the *app_virtual_com()* function which uses this interface to implement a simple loopback to interact with user. The following code is taken from *app_virtual_com()* function

```
void app_virtual_com(client interface usb_cdc_interface cdc) {
  while (1) {
    char cdc_char = cdc.get_char();
    cdc.put_char(cdc_char);
    if (cdc_char == '\r')
      cdc.put_char('\n');
  }
}
```

In the above code you can observe that the interface's functions are accessed via a variable 'cdc'. This variable is the client side of the *usb_cdc_interface.*

APPENDIX A - Demo Hardware Setup

To setup the demo hardware the following boards are required.

- xCORE-USB slicekKIT (XK-SK-U16-ST)
  - xCORE-USB Core board.
  - USB A/B sliceCARD.
  - xTAG-2 debug adaptor
  - Power supply

Connect xCORE-USB board power cable to power on the system

Connect xCORE-USB board to host machine with USB cable

Set xSCOPE switch on xCORE-USB core board to the on position

Connect XTAG to XSYS connector and XTAG USB connection to host PC
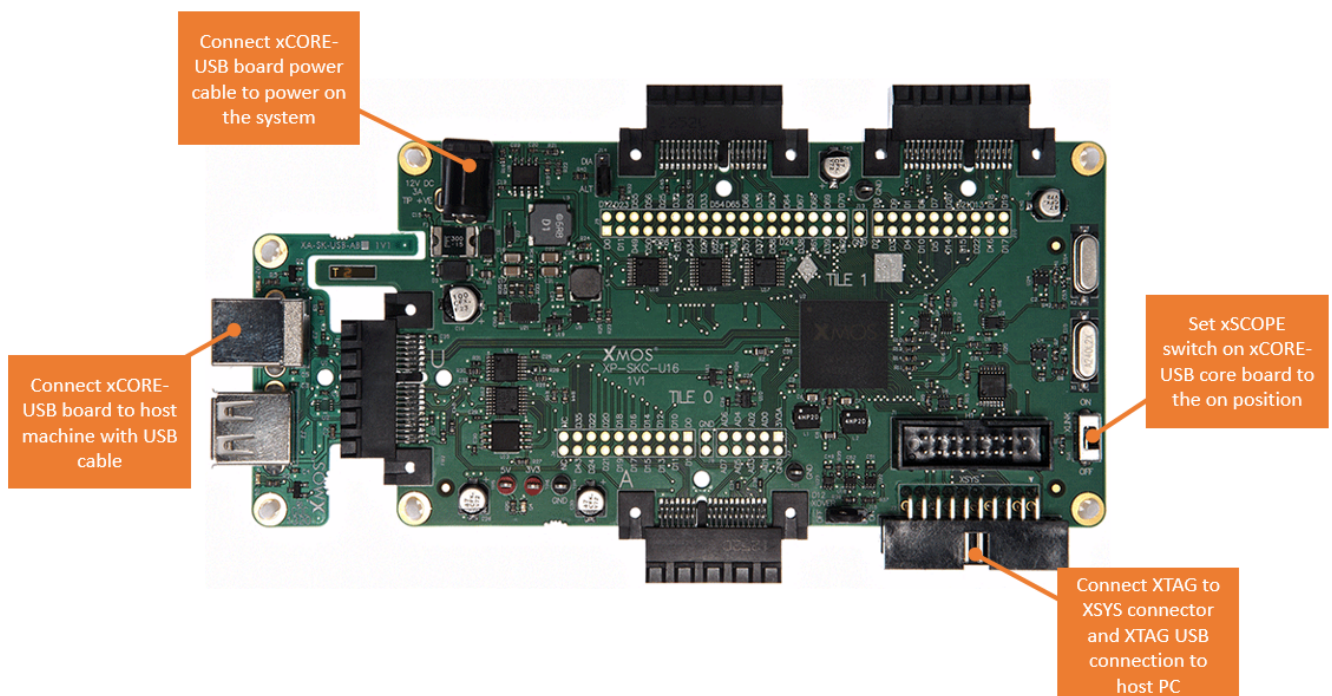
Figure 4: XMOS xCORE-USB sliceKIT

The hardware should be configured as displayed above for this demo:

- The XTAG debug adapter should be connected to the XSYS connector and the XTAG USB should be connected to the host machine.
- The USB sliceCARD should be connected to the U slot (J4 header) of the xCORE-USB Core board and the other end of USB sliceCARD should be connected to host machine using a USB cable.
- The XLINK switch on the core board should be set on ON position.
- The xCORE-USB core board should have the power cable connected.

APPENDIX B - Launching the demo application

Once the demo example has been built either from the command line using xmake or via the build mechanism of xTIMEcomposer studio we can execute the application on the xCORE-USB sliceKIT.

Once built there will be a *bin* directory within the project which contains the binary for the xCORE device. The xCORE binary has a XMOS standard .xe extension.

B.1   Launching from the command line

From the command line we use the xrun tool to download code to both the xCORE devices. If we change into the bin directory of the project we can execute the code on the xCORE microcontroller as follows:

```
> xrun app_usb_cdc_demo.xe          <-- Download and execute the xCORE code
```

Once this command has executed the CDC device will have enumerated as virtual serial port on your host machine.

B.2   Launching from xTIMEcomposer Studio

From xTIMEcomposer Studio we use the run mechanism to download code to xCORE device. From *Project Explorer* select the xCORE binary from the bin directory, right click and then run as xCORE application will execute the code on the xCORE device.

Once this command has executed, the CDC device will have enumerated as virtual serial port on your host machine.

B.3   Running the Virtual COM demo

To run the demo, you need to have a serial terminal software in your host machine.

Following sections describe in detail on how to run the demo on different OS platforms.

B.3.1   Running on Windows

- In Microsoft Windows, when the USB CDC device enumerates for the frist time it will ask for a host driver. Use 'Install driver from specific location' option to point to the 'cdc_demo.inf' supplied along with this application note. This will load the 'usbser.sys' host driver for the virtual COM device.
- Once the driver is installed, the device will be assigned with a COM port number and it will look like the following figure in "Device Manager" (Start->Control Panel->System->Hardware->Device Manager).
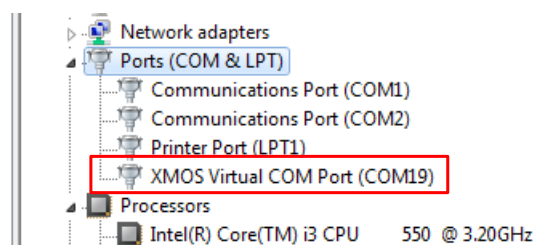

Figure 5: Enumerated Virtual COM Port Device in Windows

- Use any terminal software to open the COM port with default settings. In this demo, we have used *Hercules* as the terminal software.
- Any keys pressed in the terminal will be looped back to the terminal via the xCORE device as you type. Pressing enter will add a new line to the terminal output on return from the xCORE application.

B.3.2   Running on Linux

- Under Linux, when the USB CDC device enumerates the built-in *ACM* driver will be loaded automatically and the device will be mounted as **/dev/ttyACMx** where 'x' is a number.
- You can execute *dmesg* command in a command prompt to determine the name on which the device is mounted.
- Use any serial terminal software to open the virtual serial port with default settings. In this demo, we have used *Putty* software and the serial port is opened as shown below.

Figure 6: Opening Virtual Serial Port in Putty

- Any keys pressed in the terminal will be looped back to the terminal via the xCORE device as you type. Pressing enter will add a new line to the terminal output on return from the xCORE application.

B.3.3   Running on Mac OS

- In Mac OS X, the USB CDC device is supported by a default driver available in the OS and the device will appear as /dev/tty.usbmodem*. You can use *ls /dev/tty.usbmodem** command to determine the exact name of the virtual serial device.
- Use any serial terminal software to open the virtual serial port with default settings. In this demo, we have used *CoolTerm* software and the serial port is opened as shown below.

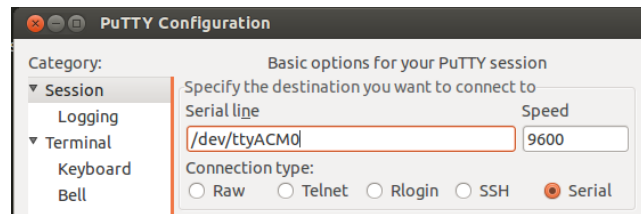- Any keys pressed in the terminal will be looped back to the terminal via the xCORE device as you type. Pressing enter will add a new line to the terminal output on return from the xCORE application.

Figure 7: Opening Virtual Serial Device in CoolTerm

APPENDIX C - References

XMOS Tools User Guide

http://www.xmos.com/published/xtimecomposer-user-guide

XMOS xCORE Programming Guide

http://www.xmos.com/published/xmos-programming-guide

XMOS xCORE-USB Device Library

http://www.xmos.com/published/xuddg

XMOS USB Device Design Guide

http://www.xmos.com/published/xmos-usb-device-design-guide

USB CDC Class Specification, USB.org:

http://www.usb.org/developers/docs/devclass_docs/

USB 2.0 Specification

http://www.usb.org/developers/docs/usb20_docs/usb_20_081114.zip

APPENDIX D - Full source code listing

D.1 Source code for main.xc

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved

#include <platform.h>
#include <xs1.h>
#include "usb.h"
#include "xud_cdc.h"

/* USB Endpoint Defines */
#define XUD_EP_COUNT_OUT   2     //Includes EP0 (1 OUT EP0 + 1 BULK OUT EP)
#define XUD_EP_COUNT_IN    3     //Includes EP0 (1 IN EP0 + 1 INTERRUPT IN EP + 1 BULK IN EP)

/* Application task */
void app_virtual_com(client interface usb_cdc_interface cdc) {
  while (1) {
    char cdc_char = cdc.get_char();
    cdc.put_char(cdc_char);
    if (cdc_char == '\r')
      cdc.put_char('\n');
  }
}

int main() {
    /* Channels to communicate with USB endpoints */
    chan c_ep_out[XUD_EP_COUNT_OUT], c_ep_in[XUD_EP_COUNT_IN];
    /* Interface to communicate with USB CDC (Virtual Serial) */
    interface usb_cdc_interface cdc_data;

    par
    {
        on USB_TILE: xud(c_ep_out, XUD_EP_COUNT_OUT, c_ep_in, XUD_EP_COUNT_IN,
                        null, XUD_SPEED_HS, XUD_PWR_SELF);

        on USB_TILE: Endpoint0(c_ep_out[0], c_ep_in[0]);

        on USB_TILE: CdcEndpointsHandler(c_ep_in[1], c_ep_out[1], c_ep_in[2], cdc_data);

        on tile[1]: app_virtual_com(cdc_data);
    }
    return 0;
}
```

D.2 Source code for xud_cdc.xc

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved

#include <xs1.h>
#include <stdio.h>
#include <string.h>
#include "usb.h"
#include "xud_cdc.h"

/* USB CDC device product defines */
#define BCD_DEVICE  0x0100
#define VENDOR_ID   0x20B1
#define PRODUCT_ID  0x0401

/* USB Sub class and protocol codes */
#define USB_CDC_ACM_SUBCLASS        0x02
#define USB_CDC_AT_COMMAND_PROTOCOL 0x01

/* CDC interface descriptor type */
#define USB_DESCTYPE_CS_INTERFACE   0x24

/* Endpoint Addresses for CDC device */
#define CDC_NOTIFICATION_EP_NUM     1
#define CDC_DATA_RX_EP_NUM          1
#define CDC_DATA_TX_EP_NUM          2
```

```
/* Data endpoint packet size */
#define MAX_EP_SIZE      512

/* CDC Communications Class requests */
#define CDC_SET_LINE_CODING         0x20
#define CDC_GET_LINE_CODING         0x21
#define CDC_SET_CONTROL_LINE_STATE  0x22
#define CDC_SEND_BREAK              0x23

/* Definition of Descriptors */
/* USB Device Descriptor */
static unsigned char devDesc[] =
{
    0x12,                   /* 0  bLength */
    USB_DESCTYPE_DEVICE,    /* 1  bdescriptorType - Device*/
    0x00,                   /* 2  bcdUSB version */
    0x02,                   /* 3  bcdUSB version */
    USB_CLASS_COMMUNICATIONS,/* 4  bDeviceClass - USB CDC Class */
    0x00,                   /* 5  bDeviceSubClass  - Specified by interface */
    0x00,                   /* 6  bDeviceProtocol  - Specified by interface */
    0x40,                   /* 7  bMaxPacketSize for EP0 - max = 64*/
    (VENDOR_ID & 0xFF),     /* 8  idVendor */
    (VENDOR_ID >> 8),       /* 9  idVendor */
    (PRODUCT_ID & 0xFF),    /* 10 idProduct */
    (PRODUCT_ID >> 8),      /* 11 idProduct */
    (BCD_DEVICE & 0xFF),    /* 12 bcdDevice */
    (BCD_DEVICE >> 8),      /* 13 bcdDevice */
    0x01,                   /* 14 iManufacturer - index of string*/
    0x02,                   /* 15 iProduct  - index of string*/
    0x03,                   /* 16 iSerialNumber  - index of string*/
    0x01                    /* 17 bNumConfigurations */
};

/* USB Configuration Descriptor */
static unsigned char cfgDesc[] = {

  0x09,                     /* 0  bLength */
  USB_DESCTYPE_CONFIGURATION, /* 1  bDescriptortype - Configuration*/
  0x43, 0x00,               /* 2  wTotalLength */
  0x02,                     /* 4  bNumInterfaces */
  0x01,                     /* 5  bConfigurationValue */
  0x04,                     /* 6  iConfiguration - index of string */
  0x80,                     /* 7  bmAttributes - Bus powered */
  0xC8,                     /* 8  bMaxPower - 400mA */

  /* CDC Communication interface */
  0x09,                     /* 0  bLength */
  USB_DESCTYPE_INTERFACE,   /* 1  bDescriptorType - Interface */
  0x00,                     /* 2  bInterfaceNumber - Interface 0 */
  0x00,                     /* 3  bAlternateSetting */
  0x01,                     /* 4  bNumEndpoints */
  USB_CLASS_COMMUNICATIONS,   /* 5  bInterfaceClass */
  USB_CDC_ACM_SUBCLASS,       /* 6  bInterfaceSubClass - Abstract Control Model */
  USB_CDC_AT_COMMAND_PROTOCOL,/* 7  bInterfaceProtocol - AT Command V.250 protocol */
  0x00,                     /* 8  iInterface - No string descriptor */

  /* Header Functional descriptor */
  0x05,                     /* 0  bLength */
  USB_DESCTYPE_CS_INTERFACE, /* 1  bDescriptortype, CS_INTERFACE */
  0x00,                     /* 2  bDescriptorsubtype, HEADER */
  0x10, 0x01,               /* 3  bcdCDC */

  /* ACM Functional descriptor */
  0x04,                     /* 0  bLength */
  USB_DESCTYPE_CS_INTERFACE, /* 1  bDescriptortype, CS_INTERFACE */
  0x02,                     /* 2  bDescriptorsubtype, ABSTRACT CONTROL MANAGEMENT */
  0x02,                     /* 3  bmCapabilities: Supports subset of ACM commands */

  /* Union Functional descriptor */
  0x05,                     /* 0  bLength */
  USB_DESCTYPE_CS_INTERFACE,/* 1  bDescriptortype, CS_INTERFACE */
  0x06,                     /* 2  bDescriptorsubtype, UNION */
  0x00,                     /* 3  bControlInterface - Interface 0 */
  0x01,                     /* 4  bSubordinateInterface0 - Interface 1 */
```

```
  /* Call Management Functional descriptor */
  0x05,                        /* 0  bLength */
  USB_DESCTYPE_CS_INTERFACE,/* 1 bDescriptortype, CS_INTERFACE */
  0x01,                        /* 2  bDescriptorsubtype, CALL MANAGEMENT */
  0x03,                        /* 3  bmCapabilities, DIY */
  0x01,                        /* 4  bDataInterface */

  /* Notification Endpoint descriptor */
  0x07,                          /* 0  bLength */
  USB_DESCTYPE_ENDPOINT,         /* 1  bDescriptorType */
  (CDC_NOTIFICATION_EP_NUM | 0x80),/* 2  bEndpointAddress */
  0x03,                          /* 3  bmAttributes */
  0x40,                          /* 4  wMaxPacketSize - Low */
  0x00,                          /* 5  wMaxPacketSize - High */
  0xFF,                          /* 6  bInterval */

  /* CDC Data interface */
  0x09,                   /* 0  bLength */
  USB_DESCTYPE_INTERFACE,    /* 1  bDescriptorType */
  0x01,                   /* 2  bInterfacecNumber */
  0x00,                   /* 3  bAlternateSetting */
  0x02,                   /* 4  bNumEndpoints */
  USB_CLASS_CDC_DATA,       /* 5  bInterfaceClass */
  0x00,                   /* 6  bInterfaceSubClass */
  0x00,                   /* 7  bInterfaceProtocol*/
  0x00,                   /* 8  iInterface - No string descriptor*/

  /* Data OUT Endpoint descriptor */
  0x07,                   /* 0  bLength */
  USB_DESCTYPE_ENDPOINT,    /* 1  bDescriptorType */
  CDC_DATA_RX_EP_NUM,       /* 2  bEndpointAddress */
  0x02,                   /* 3  bmAttributes */
  0x00,                   /* 4  wMaxPacketSize - Low */
  0x02,                   /* 5  wMaxPacketSize - High */
  0x00,                   /* 6  bInterval */

  /* Data IN Endpoint descriptor */
  0x07,                   /* 0  bLength */
  USB_DESCTYPE_ENDPOINT,    /* 1  bDescriptorType */
  (CDC_DATA_TX_EP_NUM | 0x80),/* 2  bEndpointAddress */
  0x02,                   /* 3  bmAttributes */
  0x00,                   /* 4  wMaxPacketSize - Low byte */
  0x02,                   /* 5  wMaxPacketSize - High byte */
  0x01                    /* 6  bInterval */
};

unsafe{
  /* String table - unsafe as accessed via shared memory */
  static char * unsafe stringDescriptors[]=
  {
    "\x09\x04",              /* Language ID string (US English) */
    "XMOS",                  /* iManufacturer */
    "CDC Virtual COM Port", /* iProduct */
    "0123456789"             /* iSerialNumber */
    "Config",                /* iConfiguration string */
  };
}

/* CDC Class-specific requests handler function */
XUD_Result_t ControlInterfaceClassRequests(XUD_ep ep_out, XUD_ep ep_in, USB_SetupPacket_t sp)
{
    /* Word aligned buffer */
    unsigned int buffer[32];
    unsigned length;
    XUD_Result_t result;

    static struct LineCoding {
        unsigned int baudRate;
        unsigned char charFormat;
        unsigned char parityType;
        unsigned char dataBits;
    }lineCoding;

    static struct lineState {
```

```
            unsigned char dtr;
            unsigned char rts;
     } lineState;
#if defined (DEBUG) && (DEBUG == 1)
     printhexln(sp.bRequest);
#endif

     switch(sp.bRequest)
     {
         case CDC_SET_LINE_CODING:

             if((result = XUD_GetBuffer(ep_out, (buffer, unsigned char[]), length)) != XUD_RES_OKAY)
             {
                 return result;
             }

             lineCoding.baudRate = buffer[0];     /* Read 32-bit baud rate value */
             lineCoding.charFormat = (buffer, unsigned char[])[4]; /* Read one byte */
             lineCoding.parityType = (buffer, unsigned char[])[5];
             lineCoding.dataBits = (buffer, unsigned char[])[6];

             result = XUD_DoSetRequestStatus(ep_in);

             #if defined (DEBUG) && (DEBUG == 1)
             printf("Baud rate: %u\n", lineCoding.baudRate);
             printf("Char format: %d\n", lineCoding.charFormat);
             printf("Parity Type: %d\n", lineCoding.parityType);
             printf("Data bits: %d\n", lineCoding.dataBits);
             #endif
             return result;

             break;

         case CDC_GET_LINE_CODING:

             buffer[0] = lineCoding.baudRate;
             (buffer, unsigned char[])[4] = lineCoding.charFormat;
             (buffer, unsigned char[])[5] = lineCoding.parityType;
             (buffer, unsigned char[])[6] = lineCoding.dataBits;

             return XUD_DoGetRequest(ep_out, ep_in, (buffer, unsigned char[]), 7, sp.wLength);

             break;

         case CDC_SET_CONTROL_LINE_STATE:

             /* Data present in wValue */
             lineState.dtr = sp.wValue & 0x01;
             lineState.rts = (sp.wValue >> 1) & 0x01;

             /* Acknowledge */
             result =  XUD_DoSetRequestStatus(ep_in);

             #if defined (DEBUG) && (DEBUG == 1)
             printf("DTR: %d\n", lineState.dtr);
             printf("RTS: %d\n", lineState.rts);
             #endif

             return result;

             break;

         case CDC_SEND_BREAK:
             /* Send break signal on UART (if requried) */
             // sp.wValue says the number of milliseconds to hold in BREAK condition
             return XUD_DoSetRequestStatus(ep_in);

             break;

         default:
             // Error case
             printhexln(sp.bRequest);
             break;
     }
```

```
        return XUD_RES_ERR;
}

/* Endpoint 0 handling both std USB requests and CDC class specific requests */
void Endpoint0(chanend chan_ep0_out, chanend chan_ep0_in)
{
    USB_SetupPacket_t sp;

    unsigned bmRequestType;
    XUD_BusSpeed_t usbBusSpeed;

    XUD_ep ep0_out = XUD_InitEp(chan_ep0_out, XUD_EPTYPE_CTL | XUD_STATUS_ENABLE);
    XUD_ep ep0_in = XUD_InitEp(chan_ep0_in, XUD_EPTYPE_CTL | XUD_STATUS_ENABLE);

    while(1)
    {
        /* Returns XUD_RES_OKAY on success */
        XUD_Result_t result = USB_GetSetupPacket(ep0_out, ep0_in, sp);

        if(result == XUD_RES_OKAY)
        {
            /* Set result to ERR, we expect it to get set to OKAY if a request is handled */
            result = XUD_RES_ERR;

            /* Stick bmRequest type back together for an easier parse... */
            bmRequestType = (sp.bmRequestType.Direction<<7) |
                            (sp.bmRequestType.Type<<5) |
                            (sp.bmRequestType.Recipient);

            if ((bmRequestType == USB_BMREQ_H2D_STANDARD_DEV) &&
               (sp.bRequest == USB_SET_ADDRESS))
            {
              // Host has set device address, value contained in sp.wValue
            }

            switch(bmRequestType)
            {
                /* Direction: Device-to-host and Host-to-device
                 * Type: Class
                 * Recipient: Interface
                 */
                case USB_BMREQ_H2D_CLASS_INT:
                case USB_BMREQ_D2H_CLASS_INT:

                    /* Inspect for CDC Communications Class interface num */
                    if(sp.wIndex == 0)
                    {
                        /* Returns  XUD_RES_OKAY if handled,
                         *          XUD_RES_ERR if not handled,
                         *          XUD_RES_RST for bus reset */
                        result = ControlInterfaceClassRequests(ep0_out, ep0_in, sp);
                    }
                    break;
            }
        } /* if ends */

        /* If we haven't handled the request about then do standard enumeration requests */
        if(result == XUD_RES_ERR )
        {
            /* Returns  XUD_RES_OKAY if handled okay,
             *          XUD_RES_ERR if request was not handled (STALLed),
             *          XUD_RES_RST for USB Reset */
            unsafe{
            result = USB_StandardRequests(ep0_out, ep0_in, devDesc,
                        sizeof(devDesc), cfgDesc, sizeof(cfgDesc),
                        null, 0, null, 0, stringDescriptors, sizeof(stringDescriptors)/sizeof(
                            ↪ stringDescriptors[0]),
                        sp, usbBusSpeed);
            }
        }

        /* USB bus reset detected, reset EP and get new bus speed */
        if(result == XUD_RES_RST)
        {
            usbBusSpeed = XUD_ResetEndpoint(ep0_out, ep0_in);
```

```
            }
        }
}

/* Function to handle all endpoints of the CDC class excluding control endpoint0 */
void CdcEndpointsHandler(chanend c_epint_in, chanend c_epbulk_out, chanend c_epbulk_in,
                         SERVER_INTERFACE(usb_cdc_interface, cdc))
{
    static unsigned char txBuf[2][MAX_EP_SIZE];
    static unsigned char rxBuf[2][MAX_EP_SIZE];
    int readBufId = 0, writeBufId = 0;           // used to identify buffer read/write by device
    int rxLen[2] = {0, 0}, txLen = 0;
    int readIndex = 0;
    int readWaiting = 0, writeWaiting = 1;

    unsigned length;
    XUD_Result_t result;

    /* Initialize all endpoints */
    XUD_ep epint_in = XUD_InitEp(c_epint_in, XUD_EPTYPE_INT);
    XUD_ep epbulk_out = XUD_InitEp(c_epbulk_out, XUD_EPTYPE_BUL);
    XUD_ep epbulk_in = XUD_InitEp(c_epbulk_in, XUD_EPTYPE_BUL);

    /* XUD will NAK if the endpoint is not ready to communicate with XUD */

    /* TODO: Interrupt endpoint to report serial state (if required) */

    /* Just to keep compiler happy */
    epint_in = epint_in;

    XUD_SetReady_Out(epbulk_out, rxBuf[!readBufId]);

    while(1)
    {
      select
      {
        case XUD_GetData_Select(c_epbulk_out, epbulk_out, length, result):

            if(result == XUD_RES_OKAY)
            {
                /* Received some data */
                rxLen[!readBufId] = length;

                /* Check if application has completed reading the read buffer */
                if(rxLen[readBufId] == 0) {
                    /* Switch buffers */
                    readBufId = !readBufId;
                    readIndex = 0;
                    /* Make the OUT endpoint ready to receive data */
                    XUD_SetReady_Out(epbulk_out, rxBuf[!readBufId]);
                } else {
                    /* Application is still reading the read buffer
                     * Say that another buffer is also waiting to be read */
                    readWaiting = 1;
                }
            } else {
                XUD_SetReady_Out(epbulk_out, rxBuf[!readBufId]);
            }
            break;

        case XUD_SetData_Select(c_epbulk_in, epbulk_in, result):

            /* Packet sent successfully when result in XUD_RES_OKAY */
            if (0 != txLen) {
                /* Data available to send to Host */
                XUD_SetReady_In(epbulk_in, txBuf[writeBufId], txLen);
                /* Switch write buffers */
                writeBufId = !writeBufId;
                txLen = 0;
            } else {
                writeWaiting = 1;
            }

            break;
```

```
    /* Case handlers for CDC functions */
    case (0 != rxLen[readBufId]) => cdc.read(unsigned char data[], REFERENCE_PARAM(unsigned, count)) ->
      ↪ int read_count:

        /* Some data available to read */
        if(count <= rxLen[readBufId]) {
            /* Read only 'count' number of bytes */
            memcpy(data, rxBuf[readBufId]+readIndex, count);
            read_count = count;
            readIndex += count;
            rxLen[readBufId] -= count;

        } else if(count > rxLen[readBufId]) {
            /* Read all bytes from buffer */
            memcpy(data, rxBuf[readBufId]+readIndex, rxLen[readBufId]);
            read_count = rxLen[readBufId];
            rxLen[readBufId] = 0;
            readIndex = 0;
        }

        if(readWaiting && (rxLen[readBufId] == 0)) {
            /* Other buffer is waiting to be read; switch it for reading */
            readBufId = !readBufId;
            readIndex = readWaiting = 0;
            XUD_SetReady_Out(epbulk_out, rxBuf[!readBufId]);
        }
        break;

    case (0 != rxLen[readBufId]) => cdc.get_char() -> unsigned char data:
        /* Read one byte of data */
        data = rxBuf[readBufId][readIndex++];
        rxLen[readBufId]--;

        if(readWaiting && (rxLen[readBufId] == 0)) {
            /* Other buffer is waiting to be read; switch it for reading */
            readBufId = !readBufId;
            readIndex = readWaiting = 0;
            XUD_SetReady_Out(epbulk_out, rxBuf[!readBufId]);
        }
        break;

    case (MAX_EP_SIZE != txLen) => cdc.put_char(char byte):
        txBuf[writeBufId][txLen] = byte;
        txLen++;

        /* Check if we can initiate transfer */
        if(writeWaiting) {
            XUD_SetReady_In(epbulk_in, txBuf[writeBufId], txLen);
            writeBufId = !writeBufId;
            txLen = 0;
            writeWaiting = 0;
        }
        break;

    case (MAX_EP_SIZE != txLen) => cdc.write(unsigned char data[], REFERENCE_PARAM(unsigned, length)) ->
      ↪ int write_count:

        if((txLen + length) <= MAX_EP_SIZE) {
            /* Enough space available to hold all data */
            write_count = length;
        } else {
            /* Only partial data can be put into buffer */
            write_count = MAX_EP_SIZE - txLen;
        }
        memcpy(txBuf[writeBufId] + txLen, data, write_count);
        txLen += write_count;

        /* Check if we can initiate transfer */
        if(writeWaiting) {
            XUD_SetReady_In(epbulk_in, txBuf[writeBufId], txLen);
            writeBufId = !writeBufId;
            txLen = 0;
            writeWaiting = 0;
        }
        break;
```

```
        case cdc.available_bytes() -> int count:
            count = rxLen[readBufId];
            break;

        case cdc.flush_buffer():

            /* Flush everything */
            rxLen[readBufId] = 0;
            readIndex = 0;

            if(readWaiting) {
                /* Other buffer is ready to be read, flush that too */
                readBufId = !readBufId;
                rxLen[readBufId] = 0;
                readWaiting = 0;
                XUD_SetReady_Out(epbulk_out, rxBuf[!readBufId]);
            }
            break;
        }
    }

}
```