XMOS

AN00131: USB CDC-ECM Class for Ethernet over USB

Publication Date: 2025/7/1
Document Number: XM-007103-AN v3.1.0

IN THIS DOCUMENT

1 Introduction

This application note shows how to create a USB device compliant to the standard USB Communications Device Class (CDC) and the Ethernet Control Model (ECM) Subclass on an *XMOS* device.

The code associated with this application note provides an example of using the XMOS USB Device Library (lib_xud) and associated USB class descriptors to provide a framework for the creation of a USB device emulating Ethernet.

This example USB CDC-ECM implementation provides an emulated Ethernet interface running over high speed USB. It supports the standard requests associated with ECM model of the USB CDC specification.

The demo application handles the Ethernet frames received from the USB endpoitns and hosts a HTTP web server acting as another virtual network device. A standard web browser from host PC can open the web page served from the USB device. The web page provides a statistics of different packets like ICMP, TCP, UDP etc received through the Ethernet frames from the host PC. This demonstrates a simple way in which Ethernet over USB applications can easily be deployed using an *xcore* device.

The demo application code can be extended to bridge an actual Ethernet interface by adding MAC and MII software layers. This enables you to create USB to Ethernet Adaptors using *xcore* device.

Note: This application note provides a standard USB CDC-ECM class device and as a result does not require external drivers to run on Linux and macOS. Windows doesn't support USB ECM model natively and thus requires third party drivers.

This application note is designed to run on *XMOS xcore-200* or *xcore.ai* series devices.

The example code provided with the application has been implemented and tested on the *XK-EVK-XU316* board but there is no dependency on this board and it can be modified to run on any development board which uses an *xcore-200* or *xcore.ai* series device.

- 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 the full API listing of the XMOS USB Device (XUD) Library please see the document XMOS USB Device (XUD) Library[1].

2 Overview

USB Communications Class is a USB standard that defines mechanism for connecting Networking devices to host machines. USB CDC has specified multiple Subclass standards to support different networking devices. For Ethernet-style networking over USB, one of the following Subclass specifications is used:

1. Ethernet Control Model (ECM).
2. Ethernet Emulation Model (EEM).
3. Network Control Model (NCM).

Microsoft's RNDIS is also a famous Ethernet over USB standard that is not only supported in Windows but also in other OS platorms through thrid party drivers.

Ethernet over USB provides the following advantages:

- Application independent exchange of data over USB.
- Leverage the networking protocol stack present in operating systems.
- Abstraction for USB application developers at network level like sockets, HTTP (web pages) etc.
- USB-to-Ethernet Adaptors etc.

In this application note, the USB CDC-ECM class is chosen for the implementation of Ethernet emulation of USB device and it is discussed in detail which will help you to develop your own USB CDC-ECM product using an *xcore* device or act as a reference to implement other Ethernet supporting subclass on *xcore* devices.

The standard USB CDC-ECM class is specified in a document **ECM 120.pdf** which can be found in the USB-IF website.

(https://www.usb.org/sites/default/files/CDC1.2_WMC1.1_012011.zip)

2.1 Block diagram

Fig. 1 shows a block diagram of example USB CDC-ECM applications.

3 USB CDC-ECM Class application

The example in this application note uses the XMOS USB device library (`lib_xud`) and shows a simple program that enumerates a USB CDC-ECM Class device. The host side driver of this device will emulate an Ethernet interface that seamlessly connects with the network stack of host's operating system. The USB device runs a very simple web server and acts like another device on Ethernet network i.e. the USB device imitates a network interface card of the host machine and a server connected to the host machine through Ethernet network.

For this USB CDC-ECM device application example, the system comprises four tasks running on separate logical threads of an *xcore* device.

The tasks perform the following operations.

- A task containing the USB library functionality to communicate over USB.
- A task implementing Endpoint 0 responding to both standard and CDC-ECM class-specific USB requests.

[1] https://www.xmos.com/file/lib_xud

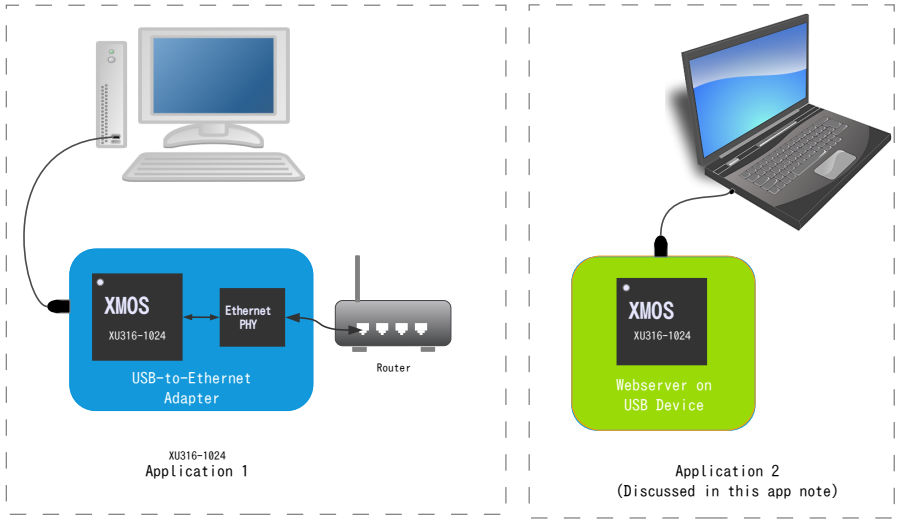

Fig. 1: Block diagram of USB CDC-ECM applications

- A task implementing the data endpoints and notification endpoint of the CDC-ECM class. It handles tx and rx buffers and provides APIs for applications to receive and transmit Ethernet frames.
- A task containing Ethernet frame handler and simple HTTP server.

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.

Fig. 2 shows the task and communication structure for this USB CDC-ECM class application example.

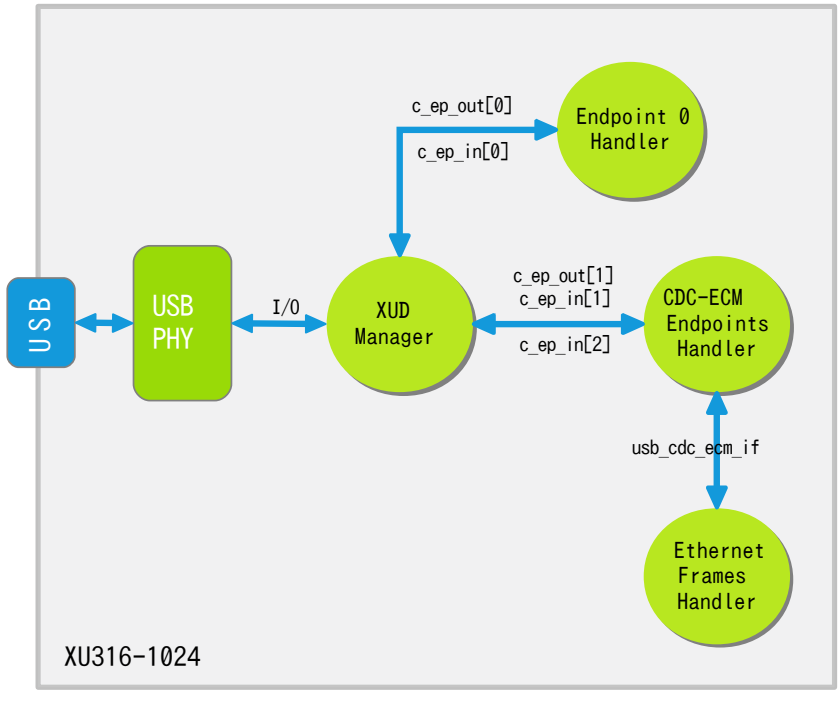

Fig. 2: Task diagram of the USB CDC-ECM example

3.1 CMakeLists.txt additions for this application

To start using the USB library, you need to add `lib_xud` to your *xcommon* compatible *CMakeLists.txt*:

```
set (APP_DEPENDENT_MODULES "lib_xud")
```

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

```
#include "xud_device.h"
```

3.2 Source code files

The example application consists of several source code files and the following list provides an overview of how the source code is organized.

- *xud_ecm.xc*, *xud_ecm.h* - Contains the USB CDC-ECM implementation which includes the USB descriptors, endpoints handler tasks (functions), class-specific defines, xC interface to read/write Ethernet frames.
- *main.xc* - Contains `main()` function and some USB defines.
- *packet_buffer.xc*, *queue.xc* - Buffer implementation to hold Ethernet frames (Max of 1514 bytes) and to allocate and deallocate dynamically from buffer pool. Queue implementation to queue up the tx and rx frame buffers in a FIFO fashion.

- *ethernet.xc* - Contains implementation of application handling the Ethernet frames and upper layer protocols like ICMP, DNS etc
- *eth_tcp* - This folder has TCP/IP and HTTP server implementation source files. Note: This is not a complete TCP/IP stack.

3.3 Setting up the USB library

`main.xc` has some arrays in it that are used to configure the endpoints for the 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)

XUD_EpType epTypeTableOut[XUD_EP_COUNT_OUT] = {XUD_EPTYPE_CTL | XUD_STATUS_ENABLE, XUD_EPTYPE_BUL};
XUD_EpType epTypeTableIn[XUD_EP_COUNT_IN] =   {XUD_EPTYPE_CTL | XUD_STATUS_ENABLE, XUD_EPTYPE_INT, XUD_EPTYPE_
↪BUL};
```

The tables above describe the endpoint configurations for this device. This example has bi-directional communication with the host machine via the standard endpoint 0 and three other endpoints for implementing the CDC-ECM class.

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

3.4 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() {

    chan c_ep_out[XUD_EP_COUNT_OUT], c_ep_in[XUD_EP_COUNT_IN];
    interface usb_cdc_ecm_if cdc_ecm;

    /* 'Par' statement to run the following tasks in parallel */
    par
    {
        on USB_TILE: XUD_Main(c_ep_out, XUD_EP_COUNT_OUT, c_ep_in, XUD_EP_COUNT_IN,
                      null, epTypeTableOut, epTypeTableIn,
                      XUD_SPEED_HS, XUD_PWR_BUS);

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

        on USB_TILE: CdcEcmEndpointsHandler(c_ep_in[1], c_ep_out[1], c_ep_in[2], cdc_ecm);

        on USB_TILE: EthernetFrameHandler(cdc_ecm);
    }
    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_Main()`. 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-ECM class: `CdcEcmEnpointsHandler()`. This function handles one bulk OUT and one bulk IN endpoints for Ethernet frame transmissions and one interrupt IN endpoint for sending notifications to host.
- There is a task to handle the Ethernet frames received from the host: `EthernetFrameHandler()`. This task implements network stack and hosts the HTTP server. It also handles DNS queries, DHCP requests and ICMP pings.
- The define USB_TILE describes the tile on which the individual tasks will run.

- In this example all tasks run on the same tile, this is a requirement of `XUD_Main()`.
- The *xCONNECT* communication channels and the xC interface *cdc_ecm* used for inter-task communication are setup at the beginning of `main()` and passed on to respective tasks.
- The endpoint type tables discussed earlier are passed into the function `XUD_Main()`.

3.5 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_ecm.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  0x0402
```

3.6 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 ECM model of the CDC class to support Ethernet emulation at the host machine.

Fig. 3 shows the descriptors used in the example code.

USB Device Descriptor

`xud_ecm.xc` is where the standard USB device descriptor is declared for the CDC-ECM 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*/
    0x00,                   /* 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.

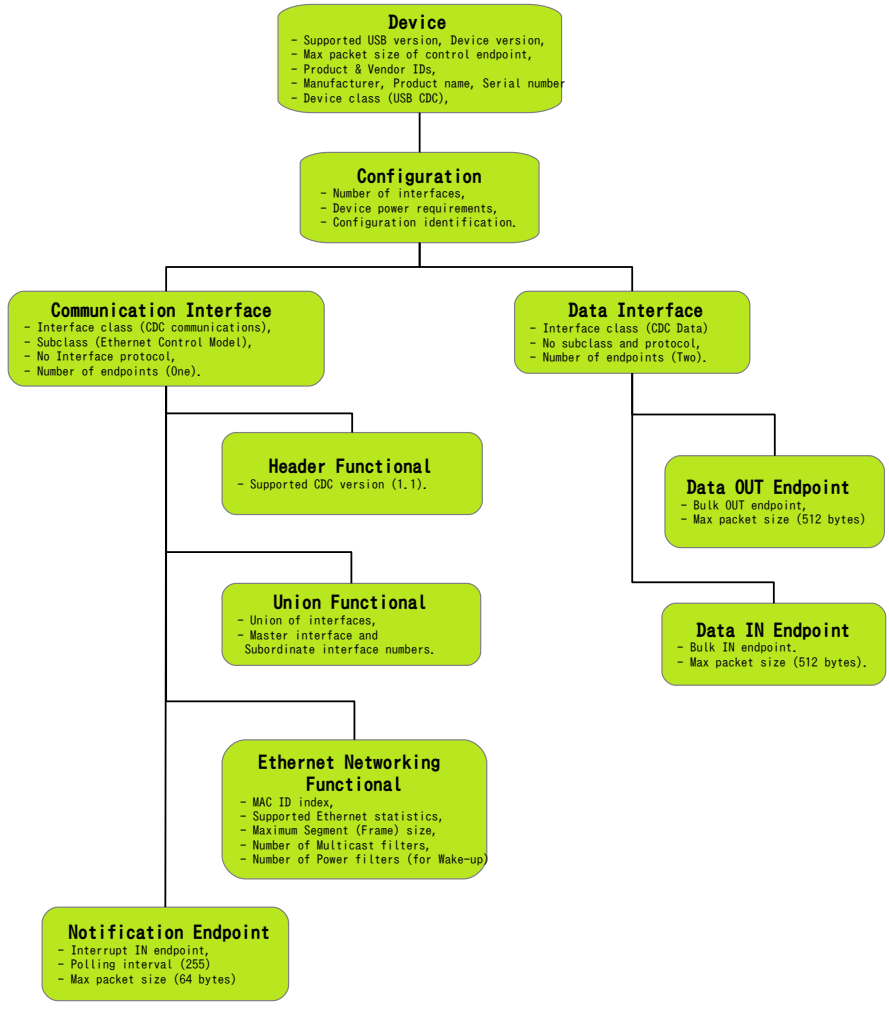

Fig. 3: USB descriptors hierarchical structure of CDC-ECM example

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*/
  0x47,00,                    /* 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_ECM_SUBCLASS,       /* 6  bInterfaceSubClass - Ethernet Control Model */
  0x00,                       /* 7  bInterfaceProtocol - No specific 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 */

  /* 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 */

  /* Ethernet Networking Functional descriptor */
  0x0D,                      /* 0 bLength - 13 bytes */
  USB_DESCTYPE_CS_INTERFACE,/* 1 bDescriptortype, CS_INTERFACE */
  0x0F,                      /* 2 bDescriptorsubtype, ETHERNET NETWORKING */
  0x03,                      /* 3 iMACAddress, Index of MAC address string */
  0x00,0x00,0x00,0x00,0x00,  /* 4 bmEthernetStatistics - Handles None */
  0xEA,05,                   /* 8 wMaxSegmentSize - 1514 bytes */
  0x00,0x00,                 /* 10 wNumberMCFilters - No multicast filters */
  0x00,                      /* 12 bNumberPowerFilters - No wake-up feature */

  /* Notification Endpoint descriptor */
  0x07,                          /* 0  bLength */
  USB_DESCTYPE_ENDPOINT,         /* 1  bDescriptorType */
  (CDC_NOTIFICATION_EP_NUM | 0x80),/* 2  bEndpointAddress - IN endpoint*/
  0x03,                          /* 3  bmAttributes - Interrupt type */
  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 - OUT endpoint */
  0x02,                      /* 3  bmAttributes - Bulk type */
  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 - IN endpoint */
  0x02,                      /* 3  bmAttributes - Bulk type */
```

(continues on next page)

(continued from previous page)

```
  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-ECM device.

The **CDC Communication interface** descriptor is for device management. You can see from the code that the device uses Etherner Control Model as the interface subclass, this will make hosts to load default driver for CDC Ethernet. 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 network connection state to the host. This endpoint is not used in this example application but will be employed when bridging an Ethernet interface to the USB-ECM device.

The **CDC Data interface** descriptor defines the interface for Ethernet frames 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{
  /* String table - unsafe as accessed via shared memory */
```

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 handling Ethernet frames in less number of transactions.

USB CDC-ECM Functional Descriptors

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 three functional descriptors used in this CDC-ECM example. They are:

1. Header Functional Descriptor.
2. Union Functional Descriptor.
3. Ethernet Networking Functional Descriptor.

The Header and Union functional descriptors are generally used in all CDC subclasses and the Ethernet Networking functional descriptor is meant for the ECM subclass.

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

```
  0x03,                   /* 3 iMACAddress, Index of MAC address string */
  0x00,0x00,0x00,0x00,    /* 4 bmEthernetStatistics - Handles None */
  0xEA,05,                /* 8 wMaxSegmentSize - 1514 bytes */
```

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

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. Following code from the CDC-ECM example shows that the the Communication Class interface 0 acts as master and the Data Class interface 1 acts as subordinate and together forming a single functional unit.

```
  /* Notification Endpoint descriptor */
  0x07,                       /* 0  bLength */
  USB_DESCTYPE_ENDPOINT,      /* 1  bDescriptorType */
  (CDC_NOTIFICATION_EP_NUM | 0x80),/* 2  bEndpointAddress - IN endpoint*/
```

Ethernet networking functional descriptor provides the Ethernet related capabilities and parameters to the host. Following code shows the fields of the descriptor.

```
  0xFF,                         /* 6  bInterval */

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

The above code shows the followings:

- 48-bit MAC address is provided through the string descriptor index, the string descriptor will carry the MAC address in a Unicode string format (example: 00229708A003).
- Ethernet statistics which the device collects. All bits are set to '0' in this example denoting that the device will not collect any Ethernet statistics.
- Maximum segment size of 1514 bytes for handling 802.3 Ethernet frames.
- Ethernet multicast filters and power filters are not supported in this USB-ECM example.

USB String Descriptors

String descriptors provide human readable information for your device and you can configure them with your USB product information. In CDC-ECM class it also provides the 48-bit MAC address of the device. 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 */
    "XMOS USB CDC Ethernet Device",/* iProduct */
    "002297000000",         /* iMACAddress */
    "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.

3.7 USB Standard and Class-Specific requests

In *xud_ecm.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-ECM 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 all 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 ECM 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 is the place to handle all CDC-ECM requests, These requests are defined in the *xud_ecm.xc* as follows:

```
/* CDC ECM Class requests Section 6.2 in CDC ECM spec */
#define SET_ETHERNET_MULTICAST_FILTERS                0x40
#define SET_ETHERNET_POWER_MANAGEMENT_PATTERN_FILTER  0x41
#define GET_ETHERNET_POWER_MANAGEMENT_PATTERN_FILTER  0x42
#define SET_ETHERNET_PACKET_FILTER                    0x43
#define GET_ETHERNET_STATISTIC                        0x44
/* 45h-4Fh RESERVED (future use) */
```

According to ECM specification, except *SET_ETHERNET_PACKET_FILTER* all the other requests are optional. These requests are not used in this example application but will be used when bridging an Ethernet interface to the USB device. In case of bridging an Ethernet interface, the Ethernet MAC software layer and PHY software layer will be employed and the MAC API functions can be directly used to perform the actions requested by the class-specific requests.

3.8 Ethernet frame handling

The Data Class interface containing the bulk endpoints is used for exchanging Ethernet frames between the device and host. The Ethernet frame includes everything starting from the destination MAC address to the end of the data field, but excludes the CRC. This frame structure is shown in Fig. 4

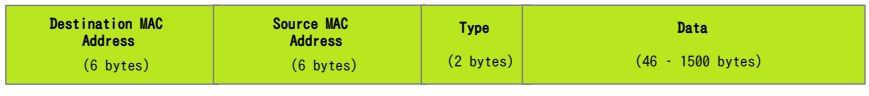

Fig. 4: Ethernet frame exchanged over the USB endpoints

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

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

        if(result == XUD_RES_OKAY)
        {
            /* Received some data */
            if(length < MAX_EP_SIZE) {
                /* USB Short packet or Zero length packet is received */
                outLen += length;
                /* Ethernet packet is received completely */
                qPut(toDevQ, outBufId, outLen);

                if(qIsFull(toDevQ)) {
                    devWaiting = 1;
                } else {
                    outBufId = packetBufferAlloc();
```

(continues on next page)

(continued from previous page)

```
                outLen = 0;
                XUD_SetReady_Out(epbulk_out, (packetBuffer[outBufId], unsigned char[]));
            }

        } else {
            /* USB Full packet */
            outLen += MAX_EP_SIZE;
            XUD_SetReady_Out(epbulk_out, (packetBuffer[outBufId], unsigned char[])+outLen);
        }

    } else {
        XUD_SetReady_Out(epbulk_out, (packetBuffer[outBufId], unsigned char[])+outLen);
    }
    break;

case XUD_SetData_Select(c_epbulk_in, epbulk_in, result):

    /* USB Packet sent successfully when result is XUD_RES_OKAY */
    int index = qPeek(toHostQ);
    int bytesSent = toHostQ.data[index].from - inIndex;
    inIndex = toHostQ.data[index].from;
    int bytesToSend = toHostQ.data[index].len - toHostQ.data[index].from;

    if(bytesToSend) {
        if (bytesToSend > MAX_EP_SIZE) {
            /* Still Large packet, split the transfer */
            bytesToSend = MAX_EP_SIZE;
        }
        XUD_SetReady_In(epbulk_in, (packetBuffer[inBufId], unsigned char[])+inIndex, bytesToSend);
        toHostQ.data[index].from += bytesToSend;

    } else if (bytesSent == MAX_EP_SIZE){
        /* Send a Zero Length Packet to indicate completion of transfer */
        XUD_SetReady_In(epbulk_in, (packetBuffer[inBufId], unsigned char[]), 0);
        inIndex += bytesSent;
    } else {
        /* Ethernet frame transfer is over */
        packetBufferFree(toHostQ.data[index].packet);
        /* Remove packet out of queue */
        qGet(toHostQ);

        /* Check if other packet is waiting to be sent to host */
        if(!qIsEmpty(toHostQ)) {
            index = qPeek(toHostQ);
            inBufId = toHostQ.data[index].packet;
            inIndex = toHostQ.data[index].from;
            bytesToSend = toHostQ.data[index].len;

            if(bytesToSend > MAX_EP_SIZE) {
                bytesToSend = MAX_EP_SIZE;
            }
            XUD_SetReady_In(epbulk_in, (packetBuffer[inBufId], unsigned char[])+inIndex, bytesToSend);
            toHostQ.data[index].from += bytesToSend;
        } else {
            /* No packets are available to send */
            hostWaiting = 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.

The maximum size of an Ethernet frame (1514 bytes) is greater than the maximum packet size (512 bytes) of the USB endpoints, therefore an Ethernet frame may be split into multiple USB packet transfers. A USB short packet notifies the end of an Ethernet frame, if the frame size is exactly a multiple of the maximum packet size of USB then a zero length packet is used to notify the end of frame.

You can see from the above code that a free buffer is allocated using *packetBufferAlloc()*, which is used to receive an Ethernet frame. Once the frame is completely received the *qPut()* function is used to add the frame into a queue. The *QUEUE_LENGTH* (four) defined in *queue.h* determines the maximum number of ethernet frames buffered up in the queue.

There is also a separate queue to handle the Ethernet frames that are transmitted to the host from the device.

3.9 Application interface

The application interface is the set of functions defined as xC interface that enables application tasks to send/receive Ethernet frames over the ECM data endpoints. The API functions abstracts out all the buffering implementation details done at the endpoint level. This xC interface is declared in *xud_ecm.h* file as shown below.

```
interface usb_cdc_ecm_if {

    int is_frame_available();

    [[guarded]] void read_frame(unsigned char buf[], REFERENCE_PARAM(unsigned, length));

    [[guarded]] void send_frame(unsigned char buf[], REFERENCE_PARAM(unsigned, length));

};
```

In the above code, the *read_frame()* function gets an Ethernet frame which is waiting in the reception queue, Similarly, the *write_frame()* function adds the Ethernet frame to the transmission queue.

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

3.10 Demo application

In this USB CDC-ECM example, the Ethernet frames received from the host are handled by the *EthernetFrameHandler()* task. This application task runs a simple HTTP server acting as a virtual network device. This task performs the following:

- Handles DHCP requests from the host PC to provide an IP address to it. The IP addresses are defined in the *ethernet.xc* file as shown below

```
/* IP Addresses of the USB device (Server) and the Host PC (Client) */
int ipAddressServer    = 0xA9FE5555;
int ipAddressClient = 0xA9FEAAAA;
unsigned char ipAddressServerArray[4] = {169, 254, 85, 85};
unsigned char ipAddressClientArray[4] = {169, 254, 170, 170};
```

The client IP address corresponds to the host PC and the server IP address belongs to the USB device.

- Handles DNS queries containing the server name defined in the *ethernet.xc* file. The server name is as shown below.

```
/* Server name as found in the DNS query */
unsigned char localName[] = "\x08xmos-cdc\x05local";
```

The *localname[]* is initialized in DNS name format and it corresponds to “xmos-cdc.local”

- Handles HTTP webpage requests and ICMP control ping requests.
- Collects a statistics of the different packets received from the host PC and embeds the information in the web page.

3.11 Demo hardware setup

To setup the demo hardware the following boards are required.

- *XK-EVK-XU316* board shown in Fig. 5
- 2 x Micro-B USB cable

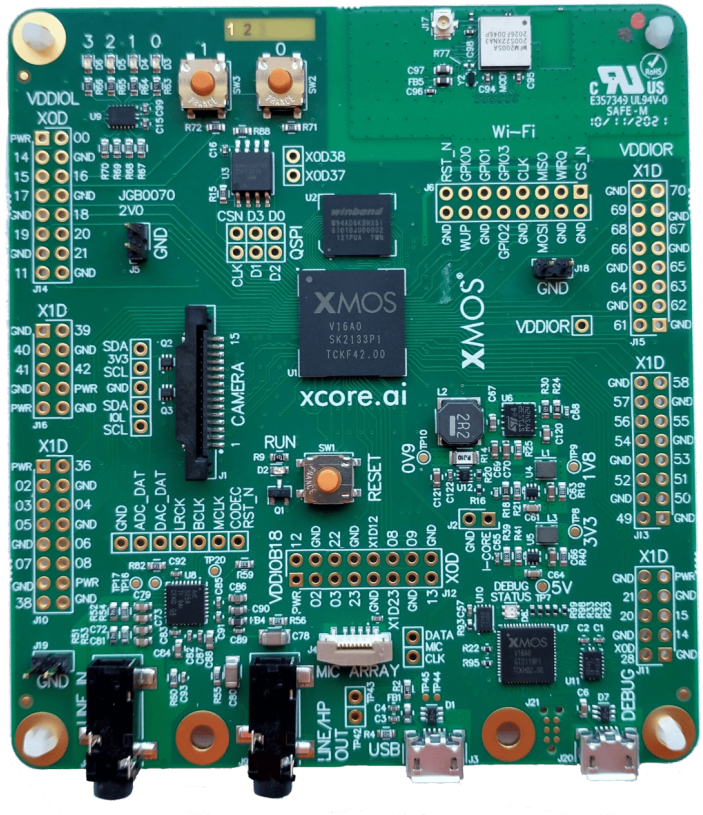

Fig. 5: XMOS *XK-EVK-316* Board

The hardware should be configured as follows:

- Connect the **USB** receptacle of the *XK-EVK-XU316* to the host machine using a USB cable
- Connect the **DEBUG** receptacle *XK-EVK-XU316* to the host machine using a USB cable

The use of *xSCOPE* is required in this application so that the print messages that are generated on the device as part of the demo do not interfere with the real-time behavior of the USB device.

3.12 Building the application

The application uses the xcommon-cmake build system as bundled with the XTC tools.

To configure the build run the following from an XTC command prompt:

```
cd app_an00131
cmake -G "Unix Makefiles" -B build
```

If any dependencies are missing it is at this configure step that they will be downloaded by the build system.

Finally, the application binary can be built using `xmake`:

```
xmake -C build
```

This command will cause a binary (app_usb_cdc_edc.xe file) to be generated in the *app_an00131/bin* directory,

3.13 Launching the demo application

Once the demo example has been built the application can be executed on the *XK-EVK-XU316*.

Once built there will be a `bin` directory within `app_an00131` which contains the binary for the *xcore* device. The *xcore* binary has a *XMOS* standard .xe extension.

Launching from the command line

From the command line the `xrun` tool is used to download and run the code on the *xcore* device. In a terminal with XTC tools sourced, from the `app_an00131/bin` directory, run:

```
xrun --xscope app_usb_cdc_edc.xe
```

Once this command has executed the application will be running on the *xcore* device.

The CDC Ethernet device should have enumerated on the host machine and the following text should be in the console window:

```
--XMOS USB CDC-ECM Class demo--

Server IP Address: 169.254.85.85
Server URL: http://xmos-cdc.local
```

3.14 Running the demo

This demo works right away on platforms that have native support for USB CDC-ECM class. Most versons of Linux have native support for this CDC Ethernet. The demo application is tested on Ubuntu 20.04.6 LTS version.

Running on Linux

Once the USB device is enumerated in the host machine, the default CDC Ethernet driver will be loaded. This host driver emulates the virtual Ethernet interface.

Run the command `ip a` in a terminal to view the emulated Ethernet interface among the list of available network interfaces. To know which is the XMOS emulated interface, run this command before and after connecting the device and see the extra interface that appears. The emulated interface showing up on a Linux host machine in the `ip a` output should look similar to the following:

```
8: enx00229708a003: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN group default qlen
↪1000
   link/ether 00:22:97:08:a0:03 brd ff:ff:ff:ff:ff:ff
```

(continues on next page)

(continued from previous page)

```
  inet 169.254.170.170/16 brd 169.254.255.255 scope global dynamic enx00229708a003
     valid_lft 230sec preferred_lft 230sec
  inet6 fe80::222:97ff:fe08:a003/64 scope link
     valid_lft forever preferred_lft forever
```

Run `ping 169.254.85.85` command to ping the server running in the USB device. This ping command sends ICMP request packets to the server and the output should look as follows:

```
xmos@sw-hw-eth-ubu0:~$ ping 169.254.85.85
PING 169.254.85.85 (169.254.85.85) 56(84) bytes of data.
64 bytes from 169.254.85.85: icmp_seq=1 ttl=64 time=0.211 ms
64 bytes from 169.254.85.85: icmp_seq=2 ttl=64 time=0.153 ms
64 bytes from 169.254.85.85: icmp_seq=3 ttl=64 time=0.152 ms
64 bytes from 169.254.85.85: icmp_seq=4 ttl=64 time=0.165 ms
64 bytes from 169.254.85.85: icmp_seq=5 ttl=64 time=0.146 ms
64 bytes from 169.254.85.85: icmp_seq=6 ttl=64 time=0.188 ms
64 bytes from 169.254.85.85: icmp_seq=7 ttl=64 time=0.089 ms
64 bytes from 169.254.85.85: icmp_seq=8 ttl=64 time=0.185 ms
64 bytes from 169.254.85.85: icmp_seq=9 ttl=64 time=0.330 ms
^C
--- 169.254.85.85 ping statistics ---
9 packets transmitted, 9 received, 0% packet loss, time 8185ms
```

Open the URL *http://xmos-cdc.local* in a standard web browser like Mozilla or Chrome to see the web page hosted by the USB device. The web page provides a statistics of the packets handled by the USB device and it is shown in Fig. 6.

Refresh the opened webpage to see the packets count updated. Also note the number of ICMP packets is same as the number of ping requests sent out previously.

3.15 Troubleshooting

If the host cannot ping the *xcore* device, potential problems could be:

- Link is down for the XMOS network. Run `ip a` and ensure that the link is up for the XMOS emulated network. `ip a` output when the link state is down looks similar to:

  ```
  8: enx00229708a003: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group default qlen 1000
  link/ether 00:22:97:08:a0:03 brd ff:ff:ff:ff:ff:ff
  ```

  If the link is down, run `ip link set <interface_name> up` command to change this.

- The XMOS network shows up with an IPV6 address. The `ip a` output might look similar to:

  ```
  8: enx00229708a003: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN group default
  ↪qlen 1000
  link/ether 00:22:97:08:a0:03 brd ff:ff:ff:ff:ff:ff
  inet6 fe80::222:97ff:fe08:a003/64 scope link
     valid_lft forever preferred_lft forever
  ```

  Restart DHCP to fix this:

  ```
  sudo dhclient -r
  sudo dhclient <interface_name>
  ```

  Run `ip a` to ensure that the XMOS Ethernet interface has an IPV4 address assigned to it, similar to:

  ```
  8: enx00229708a003: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN group default
  ↪qlen 1000
    link/ether 00:22:97:08:a0:03 brd ff:ff:ff:ff:ff:ff
    inet 169.254.170.170/16 brd 169.254.255.255 scope global dynamic enx00229708a003
      valid_lft 230sec preferred_lft 230sec
    inet6 fe80::222:97ff:fe08:a003/64 scope link
      valid_lft forever preferred_lft forever
  ```

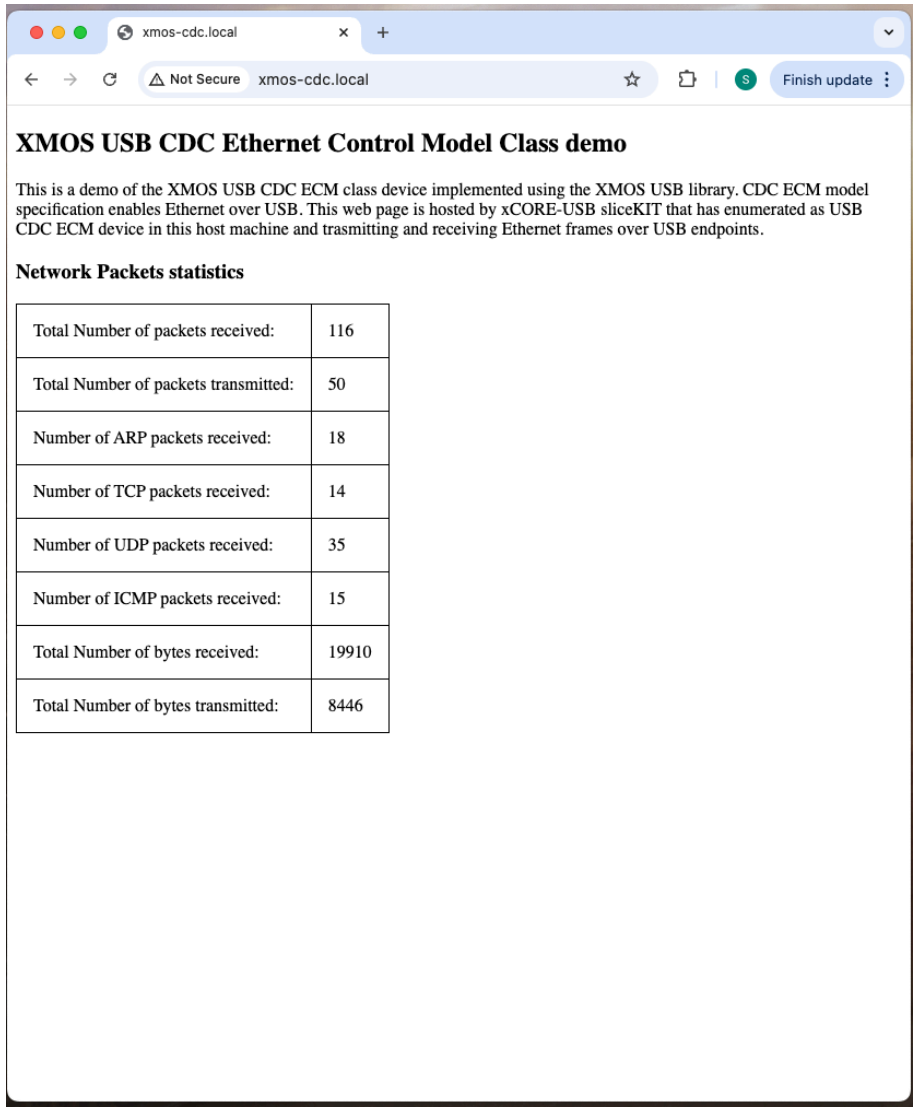

Fig. 6: Webpage hosted by USB CDC-ECM device

4 Further reading

- XMOS XTC Tools Installation Guide
- XMOS XTC Tools User Guide
- USB 2.0 Specification
- XMOS application build and dependency management system; xcommon-cmake
- USB CDC Class Specification, USB.org:
- Ethernet Frame Format