Application Note: AN00136

USB Vendor Specific Device

This application note shows how to create a vendor specific USB device which is on an XMOS multicore microcontroller.

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

This example uses XMOS libraries to provide a simple USB bulk transfer device running over high speed USB. The code used in the application note creates a device which supports the standard requests associated with this class of USB devices.

The application operates as a simple data transfer device which can transmit and receive buffers between a USB host and XMOS based USB device. This demonstrates the simple way in which custom USB devices can easily be deployed using an xCORE device.

Note: This application note provides a custom USB class device as an example and requires a driver to run on windows. For this example we have used the open source libusb host library and windows driver to allow the demo device to be used from the host machine. On other host platforms supported by this application example a host driver is not required to interact with libusb.

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 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

Copyright 2015 XMOS Ltd.

1 Overview

1.1 Introduction

The USB specification allows the creation of completely custom USB devices which do not conform to any other of the USB device class standards.

These types of devices when enumerating as a USB device declares to the host that it is vendor specific and that the host should not attempt to interface to it in any way other than to enumerate the device on the USB bus.

A vendor specific device can contain a number of endpoints and endpoint types which relate to this vendor specific device and the device class descriptor is used to specify how the device is structured.

Examples of such devices would include

- Adapters which bridge debug interfaces such as JTAG to a host PC
- Devices which control a variety of custom interfaces from a host PC
- Systems which stream large amounts of captured data to a host PC

In most cases these systems implement a custom command set over USB in order to send commands to the USB device to perform operations.

These devices also require a custom driver for the host machine on Windows as there is no OS support for custom vendor specific devices. In most cases the interface provided by the USB device is also vendor specific and requires a vendor specific host application in order to use the device.

There is no USB specification for devices of this type as it is vendor specific, the specification for the USB 2.0 standard in general can be found here,

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

1.2 Block diagram

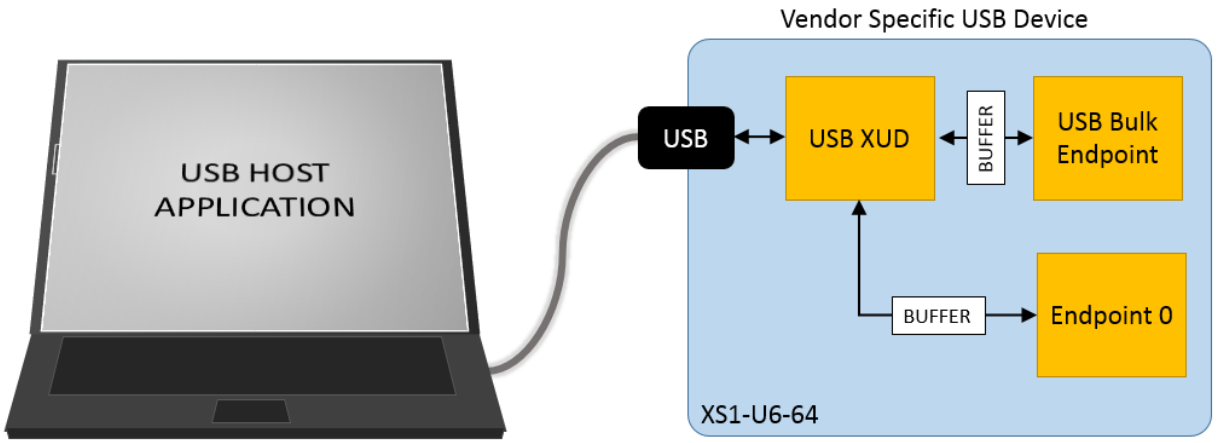

Figure 1: Block diagram of USB vendor specific device application example

2 USB Vendor Specific Device Application Note

The demo in this note uses the XMOS USB device library and shows a simple program that creates a basic vendor specific device which responds to data transfer requests from the host PC.

For the USB HID class application example, the system comprises three tasks running on separate logical cores of a 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 standard USB control requests
- A task implementing the application code for our custom bulk interface

These tasks communicate via the use of xCONNECT channels which allow data to be passed between application code running on the separate logical cores.

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

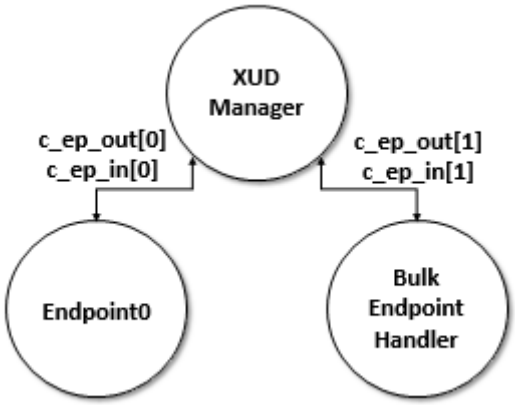

Figure 2: Task diagram of vendor specific bulk endpoint device

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 Declaring resource and setting up the USB components

`main.xc` contains the application implementation for a USB vendor specific device. There are some defines in it that are used to configure the XMOS USB device library. These are displayed below.

The second set of defines describe the endpoint configuration for this device. This example has bi-directional communication with the host machine via the standard endpoint0 and an endpoint for implementing the vendor specific bulk endpoint with is also bi-directional.

Copyright 2015 XMOS Ltd.

```
chan c_ep_out[XUD_EP_COUNT_OUT], c_ep_in[XUD_EP_COUNT_IN];

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: bulk_endpoint(c_ep_out[1], c_ep_in[1]);
}
```

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

2.3 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];

    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: bulk_endpoint(c_ep_out[1], c_ep_in[1]);
    }

    return 0;
}
```

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

- The par functionality describes running three separate tasks in parallel
- There is a function call to configure and execute the USB library: `xud()`
- There is a function call to startup and run the Endpoint0 code: `Endpoint0()`
- There is a function to deal with the custom bulk endpoints `bulk_endpoint()`
- The define USB_TILE describes the tile on which the individual tasks will run
- In this example all tasks run on the same tile as the USB PHY although this is only a requirement of `xud()`
- The xCONNECT communication channels used by the application are set up at the beginning of `main()`
- The USB defines discussed earlier are passed into the function `xud()`

2.4 Configuring the USB Device ID

The USB ID values used for Vendor ID (VID), Product ID (PID) and device version number are defined in the file `endpoint0.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.

Copyright 2015 XMOS Ltd.

```
#define BCD_DEVICE              0x1000
#define VENDOR_ID               0x20B1
#define PRODUCT_ID              0x00B1
#define MANUFACTURER_STR_INDEX  0x0001
#define PRODUCT_STR_INDEX       0x0002
```

2.5 USB Vendor Specific Class specific defines

The USB Vendor Specific Class is configured in the file endpoint0.xc. Below there are a set of standard defines which are used to configure the USB device descriptors to setup a USB vendor specific device running on an xCORE-USB microcontroller.

```
#define VENDOR_SPECIFIC_CLASS    0xff
#define VENDOR_SPECIFIC_SUBCLASS 0xff
#define VENDOR_SPECIFIC_PROTOCOL 0xff
```

These are defined in the USB standard as required in the device description for vendor specific devices and for configuring them as such with the USB host machine.

2.6 USB Device Descriptor

endpoint0.xc is where the standard USB device descriptor is declared for a vendor specific 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. This descriptor contains the vendor specific defines described above.

```
static unsigned char devDesc[] =
{
    0x12,                       /* 0  bLength */
    USB_DESCTYPE_DEVICE,        /* 1  bdescriptorType */
    0x00,                       /* 2  bcdUSB */
    0x02,                       /* 3  bcdUSB */
    VENDOR_SPECIFIC_CLASS,      /* 4  bDeviceClass */
    VENDOR_SPECIFIC_SUBCLASS,   /* 5  bDeviceSubClass */
    VENDOR_SPECIFIC_PROTOCOL,   /* 6  bDeviceProtocol */
    0x40,                       /* 7  bMaxPacketSize */
    (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 */
    MANUFACTURER_STR_INDEX,     /* 14 iManufacturer */
    PRODUCT_STR_INDEX,          /* 15 iProduct */
    0x00,                       /* 16 iSerialNumber */
    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 our vendor specific device when it is connected to the USB bus.

Copyright 2015 XMOS Ltd.

2.7 USB Configuration Descriptor

The USB configuration descriptor is used to configure the device class and the endpoint setup. For the USB vendor specific device provide in this example the configuration descriptor which is read by the host is as follows.

```
static unsigned char cfgDesc[] =
{
    0x09,                       /* 0  bLength */
    0x02,                       /* 1  bDescriptortype */
    0x20, 0x00,                 /* 2  wTotalLength */
    0x01,                       /* 4  bNumInterfaces */
    0x01,                       /* 5  bConfigurationValue */
    0x00,                       /* 6  iConfiguration */
    0x80,                       /* 7  bmAttributes */
    0xFA,                       /* 8  bMaxPower */

    0x09,                       /* 0  bLength */
    0x04,                       /* 1  bDescriptorType */
    0x00,                       /* 2  bInterfacecNumber */
    0x00,                       /* 3  bAlternateSetting */
    0x02,                       /* 4: bNumEndpoints */
    0xFF,                       /* 5: bInterfaceClass */
    0xFF,                       /* 6: bInterfaceSubClass */
    0xFF,                       /* 7: bInterfaceProtocol*/
    0x03,                       /* 8  iInterface */

    0x07,                       /* 0  bLength */
    0x05,                       /* 1  bDescriptorType */
    0x01,                       /* 2  bEndpointAddress */
    0x02,                       /* 3  bmAttributes */
    0x00,                       /* 4  wMaxPacketSize */
    0x02,                       /* 5  wMaxPacketSize */
    0x01,                       /* 6  bInterval */

    0x07,                       /* 0  bLength */
    0x05,                       /* 1  bDescriptorType */
    0x81,                       /* 2  bEndpointAddress */
    0x02,                       /* 3  bmAttributes */
    0x00,                       /* 4  wMaxPacketSize */
    0x02,                       /* 5  wMaxPacketSize */
    0x01                        /* 6  bInterval */
};
```

This decriptor is in the format decribed by the USB 2.0 standard and contains the encoding for the endpoints related to control endpoint 0 and also the descriptors that decribe the 2 bulk endpoints which form our custom device.

2.8 USB string descriptors

The final descriptor for our vendor specific device is the string descriptor which the host machine uses to report to the user when the device is enumerated and when the user queries the device on the host system. This is setup as follows.

```
/* String table */
unsafe
{
static char * unsafe stringDescriptors[] =
{
    "\x09\x04",                             // Language ID string (US English)
    "XMOS",                                 // iManufacturer
    "XMOS Custom Bulk Transfer Device",     // iProduct
    "Custom Interface",                     // iInterface
    "Config",                               // iConfiguration
};
```

2.9 USB Vendor Specific Class Endpoint0

The function Endpoint0() contains the code for dealing with device requests made from the host to the standard endpoint0 which is present in all USB devices.

There are no additional requests which need to be handled for a vendor specific device.

```
/* Endpoint 0 Task */
void Endpoint0(chanend chan_ep0_out, chanend chan_ep0_in)
{
    USB_SetupPacket_t sp;
    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)
        {
            /* Returns  XUD_RES_OKAY if handled okay,
             *          XUD_RES_ERR if request was not handled (i.e. STALLed),
             *          XUD_RES_RST if USB Reset */
            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);
        }
    }
}
```

2.10 Handling requests to the custom bulk endpoints

The application endpoints for receiving and transmitting to the host machine are implemented in the file `main.xc`. This is contained within the function `bulk_endpoint()` which is shown below:

```
void bulk_endpoint(chanend chan_ep_from_host, chanend chan_ep_to_host)
{
    int host_transfer_buf[BUFFER_SIZE];
    unsigned host_transfer_length = 0;
    XUD_Result_t result;

    XUD_ep ep_from_host = XUD_InitEp(chan_ep_from_host, XUD_EPTYPE_BUL | XUD_STATUS_ENABLE);
    XUD_ep ep_to_host = XUD_InitEp(chan_ep_to_host, XUD_EPTYPE_BUL | XUD_STATUS_ENABLE);

    while(1)
    {
        /* Receive a buffer (512-bytes) of data from the host */
        if((result = XUD_GetBuffer(ep_from_host, (host_transfer_buf, char[BUFFER_SIZE * 4]),
          ↪ host_transfer_length)) == XUD_RES_RST)
        {
            XUD_ResetEndpoint(ep_from_host, ep_to_host);
            continue;
        }

        /* Perform basic processing (increment data) */
        for (int i = 0; i < host_transfer_length/4; i++)
            host_transfer_buf[i]++;

        /* Send the modified buffer back to the host */
        if((result = XUD_SetBuffer(ep_to_host, (host_transfer_buf, char[BUFFER_SIZE * 4]),
          ↪ host_transfer_length)) == XUD_RES_RST)
        {
            XUD_ResetEndpoint(ep_from_host, ep_to_host);
        }
    }
}
```

From this you can see the following.

- A buffer is declared to communicate and transfer data with the host `host_transfer_buf` of size BUFFER_SIZE.
- This task operates inside a `while (1)` loop which repeatedly deals with a sequence of requests from the host to send data to the device and then host to then read data from the device.
- A blocking call is made to the XMOS USB device library to receive (using `XUD_GetBuffer`) and send data (using `XUD_SetBuffer`) to the host machine at every loop iteration.
- The function performs some basic processing on the recieved host buffer and simply increments the values in the buffer received from the host and then sends it back.
- This simple processing could easily be replaced with access to a piece of hardware connected to the xCORE GPIO or communication with another parallel task.

Copyright 2015 XMOS Ltd.

APPENDIX A - Example Hardware Setup

To run the example, connect the xCORE-USB sliceKIT USB-B and xTAG-2 USB-A connectors to separate USB connectors on your development PC.

On the xCORE-USB sliceKIT ensure that the xCONNECT LINK switch is set to ON, as per the image, to allow xSCOPE to function. 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.

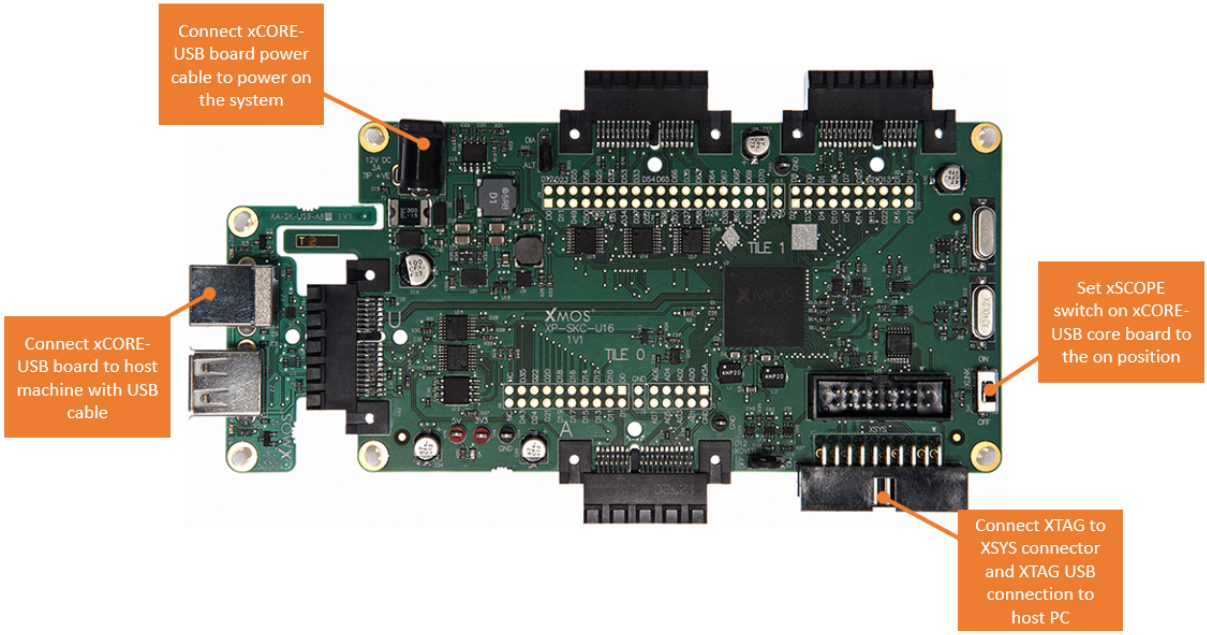

Figure 3: XMOS xCORE-USB sliceKIT

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 cable should be connected to the host machine
- The xCORE-USB core board should have a USB cable connecting the device to the host machine
- The xSCOPE switch on the board should be set to the on position
- The xCORE-USB core board should have the power cable connected

Copyright 2015 XMOS Ltd.

APPENDIX B - Host Application Setup

B.1 Test application

This simple host example demonstrates simple bulk transfer requests between the host processor and the XMOS device.

The application simply transfers a data buffer to the device and back. The device increments the data values before returning the new values to the host. The host then increments the values and sends them again a number of times.

The binaries and (where required) setup scripts are provided for each sample platform in the named directory.

B.2 Windows driver

On Windows you require a custom driver to support the vendor specific USB device. This is provided in the driver directory within the Win32 directory. When starting the device for the first time you will need to point Windows at this directory when it requests a driver to install for the device.

B.3 Licensing

libusb is written in C and licensed under the LGPL-2.1.

B.4 Compilation instructions

If you require to recompile the binary test program then the instructions to do so are below for each platform,

Win32:

```
cl -o bulktest ..\bulktest.cpp -I ..\libusb\Win32 ..\libusb\Win32\libusb.lib
```

OSX:

```
g++ -o bulktest ../bulktest.cpp -I ../libusb/OSX ../libusb/OSX/libusb-1.0.0.dylib -m32
```

Linux32:

```
g++ -o bulktest ../bulktest.cpp -I ../libusb/Linux32 ../libusb/Linux32/libusb-1.0.a -lpthread -lrt
```

Linux64:

```
g++ -o bulktest ../bulktest.cpp -I ../libusb/Linux64 ../libusb/Linux64/libusb-1.0.a -lpthread -lrt
```

Copyright 2015 XMOS Ltd.

APPENDIX C - 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 the applcation can be executed 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.

C.1 Launching from the command line

From the command line the `xrun` tool is used to download code to the xCORE-USB device. Changing into the bin directory of the project the code can be executed on the xCORE microcontroller as follows:

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

Once this command has executed the vendor specific USB device should have enumerated on the host machine

C.2 Launching from xTIMEcomposer Studio

From xTIMEcomposer Studio the run mechanism is used to download code to the xCORE device. 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 vendor specific USB device should have enumerated on your machine

C.3 Running the vendor specific host demo

To run the example, source the appropriate setup script and then execute the 'bulktest' application from the command line.

This will connect to the USB device running on the xCORE microcontroller and transfer data buffers back and forth.

The output should be similar to below:

```
XMOS Bulk USB device opened .....
Timing write/read of 1000 512-byte buffers.....
125 ms (7.81 MB/s)
XMOS Bulk USB device data processed correctly .....
XMOS Bulk USB device closed .....
```

This application is intended as a simple demonstration application and has not been programmed for efficient data transfer. The performance reported for this simple application will vary depending on the capabilities of your USB host and host operating system.

Copyright 2015 XMOS Ltd.

APPENDIX D - Bulk read benchmark example

Currently the optimized bulk read benchmark is only supported on OSX and Linux, Windows is not supported at this time.

Included with the example host code is a bulk read benchmark demo. This demonstrates high performance data throughput from the device to the host. The main difference is in the host code which uses asynchronous non blocking calls to utilize the USB bus more effectively. In order to run this benchmark you need to do the following.

D.1 Device code changes

There is an replacement function that needs to be used in the xCORE device code, this can be swapped in by editing 'src/main.xc' and changing the call in main() from 'bulk_endpoint' to 'bulk_endpoint_read_benchmark'. This function is a simplified version of the one used in the full example but only deals with read requests from the USB host.

The code can be seen below,

```
void bulk_endpoint_read_benchmark(chanend chan_ep_from_host, chanend chan_ep_to_host)
{
    char host_transfer_buf[BUFFER_SIZE*4];
    unsigned host_transfer_length = 512;

    XUD_ep ep_to_host = XUD_InitEp(chan_ep_to_host, XUD_EPTYPE_BUL);
    XUD_ep ep_from_host = XUD_InitEp(chan_ep_from_host, XUD_EPTYPE_BUL);

    while(1)
    {
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
    }
}
```

From this the following can be seen,

- The host transfer length is set to 512 bytes to match the host application
- The while loop has been unrolled to contain 8 calls to 'XUD_SetBuffer'
- The from host endpoint is not used but it still needs to be initialized

Once you have made the changes to the application mentioned above it needs to be rebuilt and executed using the instructions in APPENDIX C

D.2 Host code changes

There is a new host application provided to work with this example, it is contained in the file 'bulk_read_benchmark.cpp' which is in the host directory. In order to use this you will need to use the instructions for building the host application in APPENDIX B for your required platform. The only change to the command line it to replace the file 'bulktest.cpp' with 'bulk_read_benchmark.cpp'.

Once you have built this it can be executed as described in APPENDIX C.3

This example runs forever and will need to be terminated with a ctrl-C when required.

The output should look as follows, with the performance depending on host platform and USB hardware:

Copyright 2015 XMOS Ltd.

```
XMOS Bulk USB device opened .....
Read transfer rate 32.19 MB/s
Read transfer rate 34.94 MB/s
Read transfer rate 39.56 MB/s
Read transfer rate 39.62 MB/s
Read transfer rate 39.56 MB/s
Read transfer rate 39.56 MB/s
Read transfer rate 39.56 MB/s
Read transfer rate 39.56 MB/s
Read transfer rate 39.56 MB/s
```

Copyright 2015 XMOS Ltd.

APPENDIX E - 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 2.0 Specification

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

APPENDIX F - Full source code listing

F.1 Source code for endpoint0.xc

```
// Copyright (c) 2015, XMOS Ltd, All rights reserved

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

#define BCD_DEVICE              0x1000
#define VENDOR_ID               0x20B1
#define PRODUCT_ID              0x00B1
#define MANUFACTURER_STR_INDEX  0x0001
#define PRODUCT_STR_INDEX       0x0002

/* Vendor specific class defines */
#define VENDOR_SPECIFIC_CLASS    0xff
#define VENDOR_SPECIFIC_SUBCLASS 0xff
#define VENDOR_SPECIFIC_PROTOCOL 0xff

/* Device Descriptor */
static unsigned char devDesc[] =
{
    0x12,                     /* 0  bLength */
    USB_DESCTYPE_DEVICE,      /* 1  bdescriptorType */
    0x00,                     /* 2  bcdUSB */
    0x02,                     /* 3  bcdUSB */
    VENDOR_SPECIFIC_CLASS,    /* 4  bDeviceClass */
    VENDOR_SPECIFIC_SUBCLASS, /* 5  bDeviceSubClass */
    VENDOR_SPECIFIC_PROTOCOL, /* 6  bDeviceProtocol */
    0x40,                     /* 7  bMaxPacketSize */
    (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 */
    MANUFACTURER_STR_INDEX,   /* 14 iManufacturer */
    PRODUCT_STR_INDEX,        /* 15 iProduct */
    0x00,                     /* 16 iSerialNumber */
    0x01                      /* 17 bNumConfigurations */
};


/* Configuration Descriptor */
static unsigned char cfgDesc[] =
{
    0x09,                     /* 0  bLength */
    0x02,                     /* 1  bDescriptortype */
    0x20, 0x00,               /* 2  wTotalLength */
    0x01,                     /* 4  bNumInterfaces */
    0x01,                     /* 5  bConfigurationValue */
    0x00,                     /* 6  iConfiguration */
    0x80,                     /* 7  bmAttributes */
    0xFA,                     /* 8  bMaxPower */

    0x09,                     /* 0  bLength */
    0x04,                     /* 1  bDescriptorType */
    0x00,                     /* 2  bInterfacecNumber */
    0x00,                     /* 3  bAlternateSetting */
    0x02,                     /* 4: bNumEndpoints */
    0xFF,                     /* 5: bInterfaceClass */
    0xFF,                     /* 6: bInterfaceSubClass */
    0xFF,                     /* 7: bInterfaceProtocol*/
    0x03,                     /* 8  iInterface */

    0x07,                     /* 0  bLength */
    0x05,                     /* 1  bDescriptorType */
    0x01,                     /* 2  bEndpointAddress */
    0x02,                     /* 3  bmAttributes */
    0x00,                     /* 4  wMaxPacketSize */
    0x02,                     /* 5  wMaxPacketSize */
    0x01,                     /* 6  bInterval */
```

```
    0x07,                       /* 0  bLength */
    0x05,                       /* 1  bDescriptorType */
    0x81,                       /* 2  bEndpointAddress */
    0x02,                       /* 3  bmAttributes */
    0x00,                       /* 4  wMaxPacketSize */
    0x02,                       /* 5  wMaxPacketSize */
    0x01                        /* 6  bInterval */
};

/* Set language string to US English */
#define STR_USENG 0x0409

/* String table */
unsafe
{
static char * unsafe stringDescriptors[] =
{
    "\x09\x04",                             // Language ID string (US English)
    "XMOS",                                 // iManufacturer
    "XMOS Custom Bulk Transfer Device",     // iProduct
    "Custom Interface",                     // iInterface
    "Config",                               // iConfiguration
};
}

/* Endpoint 0 Task */
void Endpoint0(chanend chan_ep0_out, chanend chan_ep0_in)
{
    USB_SetupPacket_t sp;
    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)
        {
            /* Returns  XUD_RES_OKAY if handled okay,
             *          XUD_RES_ERR if request was not handled (i.e. STALLed),
             *          XUD_RES_RST if USB Reset */
            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);
        }
    }
}
```

F.2 Source code for main.xc

```
// Copyright (c) 2015, XMOS Ltd, All rights reserved

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

#define XUD_EP_COUNT_OUT   2
#define XUD_EP_COUNT_IN    2

/* Prototype for Endpoint0 function in endpoint0.xc */
void Endpoint0(chanend c_ep0_out, chanend c_ep0_in);
```

```
#define BUFFER_SIZE 128

/* A optimized endpoint for the read benchmark test
 * both host and device stay in sync.
 */
void bulk_endpoint_read_benchmark(chanend chan_ep_from_host, chanend chan_ep_to_host)
{
    char host_transfer_buf[BUFFER_SIZE*4];
    unsigned host_transfer_length = 512;

    XUD_ep ep_to_host = XUD_InitEp(chan_ep_to_host, XUD_EPTYPE_BUL);
    XUD_ep ep_from_host = XUD_InitEp(chan_ep_from_host, XUD_EPTYPE_BUL);

    while(1)
    {
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
    }
}


/* A basic endpoint function that receives 512-byte packets of data, processes
 * them and sends them back to the host. If at any point an error is detected
 * (return value < 0) then the process needs to be started again so that
 * both host and device stay in sync.
 */
void bulk_endpoint(chanend chan_ep_from_host, chanend chan_ep_to_host)
{
    int host_transfer_buf[BUFFER_SIZE];
    unsigned host_transfer_length = 0;
    XUD_Result_t result;

    XUD_ep ep_from_host = XUD_InitEp(chan_ep_from_host, XUD_EPTYPE_BUL | XUD_STATUS_ENABLE);
    XUD_ep ep_to_host = XUD_InitEp(chan_ep_to_host, XUD_EPTYPE_BUL | XUD_STATUS_ENABLE);

    while(1)
    {
        /* Receive a buffer (512-bytes) of data from the host */
        if((result = XUD_GetBuffer(ep_from_host, (host_transfer_buf, char[BUFFER_SIZE * 4]),
          ↪ host_transfer_length)) == XUD_RES_RST)
        {
            XUD_ResetEndpoint(ep_from_host, ep_to_host);
            continue;
        }

        /* Perform basic processing (increment data) */
        for (int i = 0; i < host_transfer_length/4; i++)
            host_transfer_buf[i]++;

        /* Send the modified buffer back to the host */
        if((result = XUD_SetBuffer(ep_to_host, (host_transfer_buf, char[BUFFER_SIZE * 4]),
          ↪ host_transfer_length)) == XUD_RES_RST)
        {
            XUD_ResetEndpoint(ep_from_host, ep_to_host);
        }
    }
}

/* The main function runs three tasks: the XUD manager, Endpoint 0, and bulk
 * endpoint. An array of channels is used for both IN and OUT endpoints,
 * endpoint zero requires both, bulk endpoint requires an IN and an OUT endpoint
 * to receive and send a data buffer to the host.
 */
int main()
{
    chan c_ep_out[XUD_EP_COUNT_OUT], c_ep_in[XUD_EP_COUNT_IN];

    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: bulk_endpoint(c_ep_out[1], c_ep_in[1]);
    }

    return 0;
}
```

Copyright © 2015, All Rights Reserved.

Xmos Ltd. is the owner or licensee of this design, code, or Information (collectively, the “Information”) and is providing it to you “AS IS” with no warranty of any kind, express or implied and shall have no liability in relation to its use. Xmos Ltd. makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.