Application Note: AN00132

USB Image Device Class

This application note shows how to create a USB device compliant to the standard USB still image capture device class 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 image acquisition over high speed USB using an XMOS device. The code used in the application note creates a still image capture device and supports the transactions between the USB host and the device compliant to PIMA 15740 Picture Transfer Protocol.

Commands for image capture are sent from a host application to the device. The example running on the xCORE in turn responds to these commands. It also generates the appropriate image and transfers to the host. The host application stores the received data in an image file format.

Note: For the example in this application note, we have used the open source `libusb` and `ImageMagick` host libraries.

Required tools and libraries

- xTIMEcomposer Tools - Version 14.0.0
- XMOS USB Device Library - Version 2.0.0

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 (XP-SKC-U16 1V2) with USB AB slice (XA-SK-USB-AB 1V2) but there is no dependency 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, the XMOS tool chain and the xC language. Please see the references in the 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[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

1 Overview

1.1 Introduction

The Universal Serial Bus (USB) is a communications architecture that gives a PC the ability to interconnect a variety of devices via a simple four-wire cable. One such device is the digital still camera.

This application note gives a simple example for interfacing a USB still image capture device with a USB host. In the example, the USB transactions implemented for image capture are compliant to the Picture Transfer Protocol (PTP) of Photographic and Imaging Manufacturers Association (PIMA) 15470 standard[4]. An image simulated at xCORE is transferred to the host.

The USB specification provides a standard device class for the implementation of USB still image capture device[5].

1.2 Block diagram

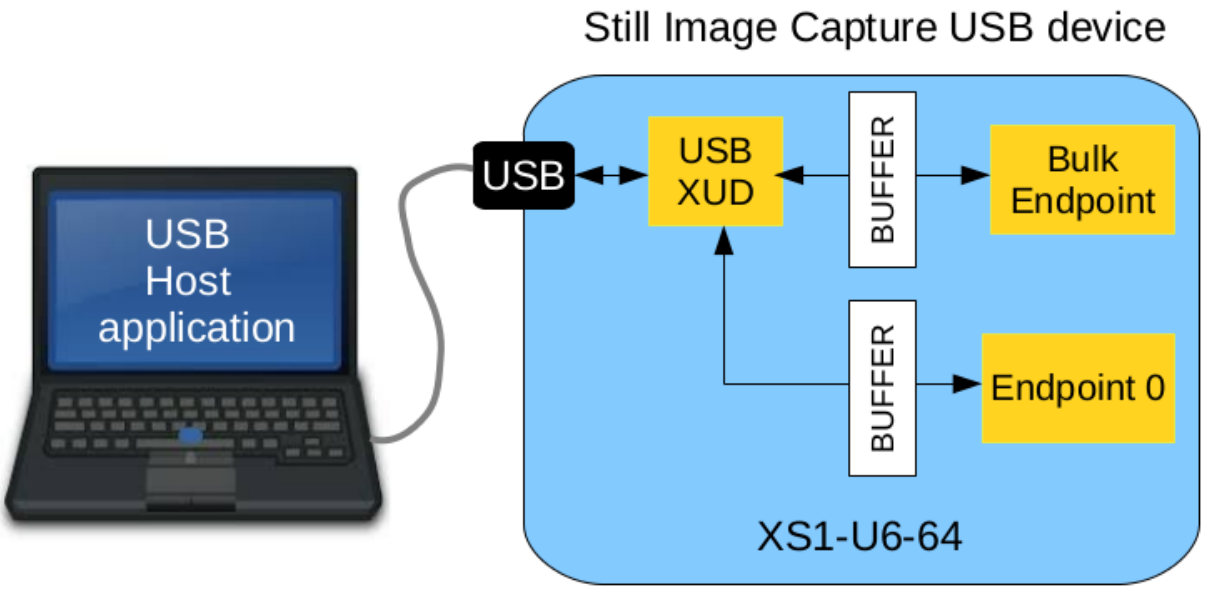

Figure 1: Block diagram of USB still image capture device application example

[4] http://www.pima.net/standards/it10/IT10_POW.htm
[5] http://www.usb.org/developers/docs/devclass_docs/usb_still_img10.zip

2 USB Still Image Capture Device Application Note

The demo in this note uses the XMOS USB Device (XUD) library and shows a simple program for interfacing a USB still image capture device. It responds to device control and image data transfer requests from the host PC.

For the USB still image capture 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 device interface

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

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

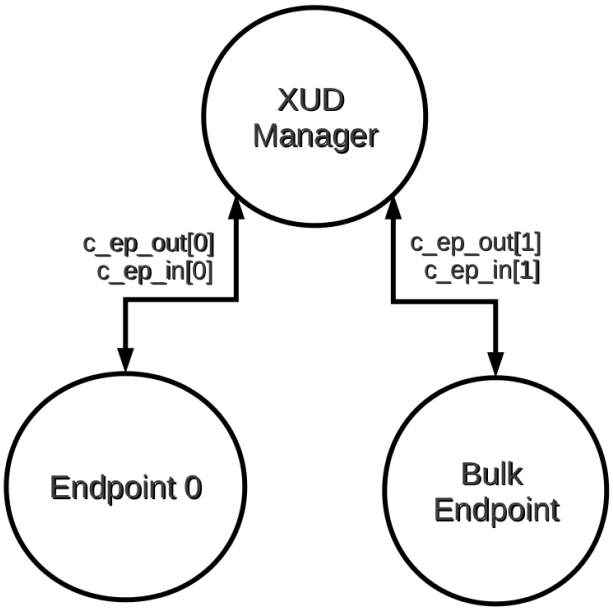

Figure 2: Task diagram of USB still image capture device interface

The still image capture class has an interrupt endpoint as well to report the asynchronous events occuring at the device to the host. The demo in this note however emulates the image capture device. The interrupt endpoint is therefore not implemented although it is instantiated in Endpoint0. Application developers can implement this endpoint when a real image capture device is connected.

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 xud.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 still image capture device. 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
#define XUD_EP_COUNT_IN    2
```

These 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 still image capture class bulk data endpoint which is also bi-directional. The endpoint type tables inform XUD what the transfer types for each endpoint in use and also if the endpoint wishes to be informed of USB bus resets.

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 bulk endpoints for command and data transfer: 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 in the beginning of main()
- The USB defines discussed earlier are passed to the function xud()

2.4 Configuring the USB Device ID

The USB ID values used for Vendor ID, Product ID 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.

```
/* USB Device ID defines */
#define BCD_DEVICE    0x1000
#define VENDOR_ID     0x20B1
#define PRODUCT_ID    0x00C1
```

2.5 USB Still Image Capture Class specific defines

The USB Still Image Capture Class is configured in the file endpoint0.xc. Below there are a set of class-specific defines and requests which are used to configure the USB device descriptors to setup a USB still image capture device running on an xCORE-USB microcontroller.

```
/* USB Still Image Capture Class defines */
#define STILL_IMAGE_SUBCLASS 0x01
#define STILL_IMAGE_PROTOCOL 0x01

/* Class-Specific Requests - bRequest values */
#define STILL_IMAGE_CANCEL_REQUEST      0x64
#define STILL_IMAGE_GET_EXT_EVENT_DATA  0x65
#define STILL_IMAGE_DEV_RESET_REQ       0x66
#define STILL_IMAGE_GET_DEV_STATUS      0x67
```

These are defined in the USB standard as required in the device definition for still image capture 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 still image capture 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.

```
static unsigned char devDesc[] =
{
    0x12,                   /* 0  bLength */
    USB_DESCTYPE_DEVICE,    /* 1  bdescriptorType */
    0x10,                   /* 2  bcdUSB */
    0x01,                   /* 3  bcdUSB */
    0x00,                   /* 4  bDeviceClass */
    0x00,                   /* 5  bDeviceSubClass */
    0x00,                   /* 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 */
    0x01,                   /* 14 iManufacturer */
    0x02,                   /* 15 iProduct */
    0x03,                   /* 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 still image device when it is connected to the USB bus.

2.7 USB Configuration Descriptor

The USB configuration descriptor is used to configure the device class and the endpoint setup. For the USB still image capture device provided in this example the configuration descriptor which is read by the host is as follows.

```
static unsigned char cfgDesc[] = {
    /* Configuration Descriptor */
    0x09,                       /* 0  bLength */
    USB_DESCTYPE_CONFIGURATION, /* 1  bDescriptortype */
    0x27, 0x00,                 /* 2  wTotalLength = 39*/
    0x01,                       /* 4  bNumInterfaces */
    0x01,                       /* 5  bConfigurationValue */
    0x04,                       /* 6  iConfiguration */
    0x80,                       /* 7  bmAttributes - Bus powered */
    0xFA,                       /* 8  bMaxPower - 500 mA*/

    /* Interface Descriptor */
    0x09,                       /* 0  bLength */
    USB_DESCTYPE_INTERFACE,     /* 1  bDescriptorType */
    0x00,                       /* 2  bInterfacecNumber */
    0x00,                       /* 3  bAlternateSetting */
    0x03,                       /* 4: bNumEndpoints */
    USB_CLASS_IMAGE,            /* 5: bInterfaceClass */
    STILL_IMAGE_SUBCLASS,       /* 6: bInterfaceSubClass */
    STILL_IMAGE_PROTOCOL,       /* 7: bInterfaceProtocol*/
    0x00,                       /* 8  iInterface */

    /* Data-in Bulk Endpoint Descriptor */
    0x07,                     /* 0  bLength */
    USB_DESCTYPE_ENDPOINT,    /* 1  bDescriptorType */
    0x01,                     /* 2  bEndpointAddress - Data EP1 OUT */
    0x02,                     /* 3  bmAttributes - Bulk transfer*/
    0x40,                     /* 4  wMaxPacketSize - set to 64 */
    0x00,                     /* 5  wMaxPacketSize */
    0x00,                     /* 6  bInterval */

    /* Data-out Bulk Endpoint Descriptor */
    0x07,                     /* 0  bLength */
    USB_DESCTYPE_ENDPOINT,    /* 1  bDescriptorType */
    0x81,                     /* 2  bEndpointAddress - Data EP1 IN */
    0x02,                     /* 3  bmAttributes - Bulk transfer*/
    0x40,                     /* 4  wMaxPacketSize */
    0x00,                     /* 5  wMaxPacketSize */
    0x00,                     /* 6  bInterval */

    /* Interrupt Endpoint Descriptor */
    0x07,                     /* 0  bLength */
    USB_DESCTYPE_ENDPOINT,    /* 1  bDescriptorType */
    0x82,                     /* 2  bEndpointAddress - Data EP1 IN */
    0x03,                     /* 3  bmAttributes - Interrupt transfer*/
    0x40,                     /* 4  wMaxPacketSize */
    0x00,                     /* 5  wMaxPacketSize */
    0x01                      /* 6  bInterval */

};
```

This decriptor is in the format decribed by the USB 2.0 standard. It contains the encoding for the end-

points related to control endpoint 0 and the descriptors that describe the two bulk endpoints for data in and out. It also contains the descriptors for the interrupt endpoint. The bulk endpoints are used for transferring image data and non-image data to adjust device controls. The interrupt endpoint is used to send asynchronous event data such as battery low indication or the removal of the memory card to the host from the device.

2.8 USB string descriptors

The final descriptor for our still image capture 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 */
static char * unsafe stringDescriptors[]=
{
    "\x09\x04",                  // Language ID string (US English)
    "XMOS",                      // iManufacturer
    "USB Still Image Capture",   // iProduct
    "0123456789",                // iSerialNumber
    "config",                    // iConfiguration
};}
```

2.9 USB Still Image Capture Class requests

Inside endpoint0.xc there is some code for handling the USB image capture device class-specific requests. These are shown in the following code:

```
/* Still Image Class-Specific Requests */
XUD_Result_t StillImageClassRequests(XUD_ep c_ep0_out, XUD_ep c_ep0_in, USB_SetupPacket_t sp)
{
    unsigned buffer[64];
    unsigned TransactionID, length;
    XUD_Result_t result;

    switch(sp.bRequest)
    {
        case STILL_IMAGE_CANCEL_REQUEST:
            /* Receives the transaction ID that was cancelled by the host */
            if((result = XUD_GetBuffer(c_ep0_out, (buffer, unsigned char[]), length)) != XUD_RES_OKAY)
            {
                return result;
            }
            memcpy (&TransactionID, &(buffer, unsigned char[])[2], 4);
            result = XUD_DoSetRequestStatus(c_ep0_in);
            return result;
            break;

        case STILL_IMAGE_GET_EXT_EVENT_DATA:
            /* Transfers the extended information on an asynchronous event stored in the buffer to the host */
            return XUD_DoGetRequest(c_ep0_out, c_ep0_in, (buffer, unsigned char[]), length, sp.wLength);
            break;

        case STILL_IMAGE_DEV_RESET_REQ:
            /* Put the device in the idle state */
            return XUD_RES_RST;
            break;

         case STILL_IMAGE_GET_DEV_STATUS:
            /* Transfers information regarding the status of the device */
            return XUD_DoGetRequest(c_ep0_out, c_ep0_in, (buffer, unsigned char[]), length, sp.wLength);
            break;

         default:
             /* Error */
             return XUD_RES_ERR;
             break;

    }

}
```

These class-specific requests are implemented by the application as they do not form part of the standard requests which have to be accepted by all device classes via endpoint0.

2.10 USB Still Image Capture Class Endpoint0

The function `Endpoint0()` contains the code for dealing with device requests made from the host to the standard endpoint 0 which is present in all USB devices. In addition to requests required for all devices, the code handles the requests specific to the still image capture device class.

```
/* Endpoint 0 Task */
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);

             /* Handle specific requests first */
            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:

                    if(sp.wIndex == 0)
                        /* Returns  XUD_RES_OKAY if handled, XUD_RES_ERR if not handled,
                         *          XUD_RES_RST for bus reset */
                        result = StillImageClassRequests(ep0_out, ep0_in, sp);
                    break;
            }
        }

        /* If we haven't handled the request above then do standard enumeration requests */
        if(result == XUD_RES_ERR)
            /* Returns  XUD_RES_OKAY if handled okay, XUD_RES_RST if USB Reset,
             *          XUD_RES_ERR if request was not handled (i.e. STALLed) */
            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);
    }
}
```

2.11 Handling requests to the bulk endpoints

The application endpoints for receiving commands and transmitting data and response 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)
{
    PTPContainer operation_response;
    PTPObjectInfo image_info;
    unsigned char cmd_buf[sizeof(PTPContainer)];
    unsigned char data_buf[USB_DATA_PKT_SIZE];
    unsigned char info_buf[sizeof(PTPObjectInfo)];
    unsigned cmd_length;
    XUD_Result_t result;

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


    while(1)
    {
        /* Receive the operation request from the host */
        if((result = XUD_GetBuffer(ep_from_host, cmd_buf, cmd_length)) == XUD_RES_RST)
        {
            XUD_ResetEndpoint(ep_from_host, ep_to_host);
            continue;
        }
        memcpy (&operation_response, cmd_buf, sizeof(PTPContainer));

        /* Process the command of custom simple protocol similar to PTP */
        uint16_t opcode = operation_response.Code;
        switch (opcode){

        case PTP_OC_OpenSession:
        case PTP_OC_InitiateCapture:
        case PTP_OC_CloseSession:
            operation_response.Code = PTP_RC_OK;
            break;

        case PTP_OC_GetObjectInfo:
            image_info.ObjectFormat = PTP_OFC_Undefined_0x3806;
            image_info.ImagePixHeight = IMG_HEIGHT;
            image_info.ImagePixWidth = IMG_WIDTH;
            if (IMG_TYPE==COLOR)
                image_info.ImageBitDepth = 24;
            else
                image_info.ImageBitDepth = 8;
            strcpy(image_info.Filename, "image.pnm");

            /* Send image info */
            memcpy (info_buf, &image_info, sizeof(PTPObjectInfo));
            if((result = XUD_SetBuffer(ep_to_host, info_buf, USB_DATA_PKT_SIZE)) == XUD_RES_RST)
            {
                XUD_ResetEndpoint(ep_from_host, ep_to_host);
                break;
            }
            if((result = XUD_SetBuffer(ep_to_host, info_buf+USB_DATA_PKT_SIZE, sizeof(PTPObjectInfo)-
              ↪ USB_DATA_PKT_SIZE)) == XUD_RES_RST)
            {
                XUD_ResetEndpoint(ep_from_host, ep_to_host);
                break;
            }

            operation_response.Code = PTP_RC_OK;
            break;
```

```
        case PTP_OC_GetObject:

            /* Generate image */
            int ncols = IMG_WIDTH;
            if (IMG_TYPE==COLOR) ncols = 3*IMG_WIDTH;

            int index = 0;
            int pkt_size = USB_DATA_PKT_SIZE;
            while (index < (IMG_HEIGHT*ncols)){

                if (((IMG_HEIGHT*ncols)-index) < USB_DATA_PKT_SIZE)
                    pkt_size = (IMG_HEIGHT*ncols)-index;
                for (int i=0; i<pkt_size; i++){
                    if (IMG_TYPE==COLOR){
                        switch (index%3){
                        case 0: data_buf[i] = ST_RED+(index%ncols); break;
                        case 1: data_buf[i] = ST_GREEN+(index%ncols); break;
                        case 2: data_buf[i] = ST_BLUE+(index%ncols); break;
                        }
                    }
                    else data_buf[i] = ST_GRAY+(index%ncols);
                    index++;
                }
                /* Send the image data packet to the host */
                if((result = XUD_SetBuffer(ep_to_host, data_buf, pkt_size)) == XUD_RES_RST)
                {
                    XUD_ResetEndpoint(ep_from_host, ep_to_host);
                    break;
                }
            }
            operation_response.Code = PTP_RC_OK;
            break;

        default:
            operation_response.Code = PTP_RC_Undefined;
            break;

       }

       /* Send response */
       operation_response.Nparam = 0;
       memcpy (cmd_buf, &operation_response, sizeof(PTPContainer));
       if((result = XUD_SetBuffer(ep_to_host, cmd_buf, sizeof(PTPContainer))) == XUD_RES_RST)
       {
           XUD_ResetEndpoint(ep_from_host, ep_to_host);
       }

    }

}
```

From this you can see the following.

- Three buffers `cmd_buf`, `info_buf` and `data_buf` are declared to communicate with the host for receiving commands and transferring data
- A `while (1)` loop which repeatedly deals with a sequence of PTP-compliant operation requests from the host, send image data and response to the device. Commands are processed and an image is autogenerated.
- In each iteration, a PTP command is processed in `switch-case` statements and an appropriate response is sent
- A gray or color gradient image is generated for the command `OC_GetImage`. The image type and the starting gray or color component values are defined in the beginning of `main.xc`
- A blocking call is made to the XMOS USB device library to receive command (using `XUD_GetBuffer`) and send data as well the response (using `XUD_SetBuffer`) to the host machine at every loop iteration
- This simple processing could easily be extended to access an image capture device connected to the xCORE GPIO or communicate with another parallel task

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.

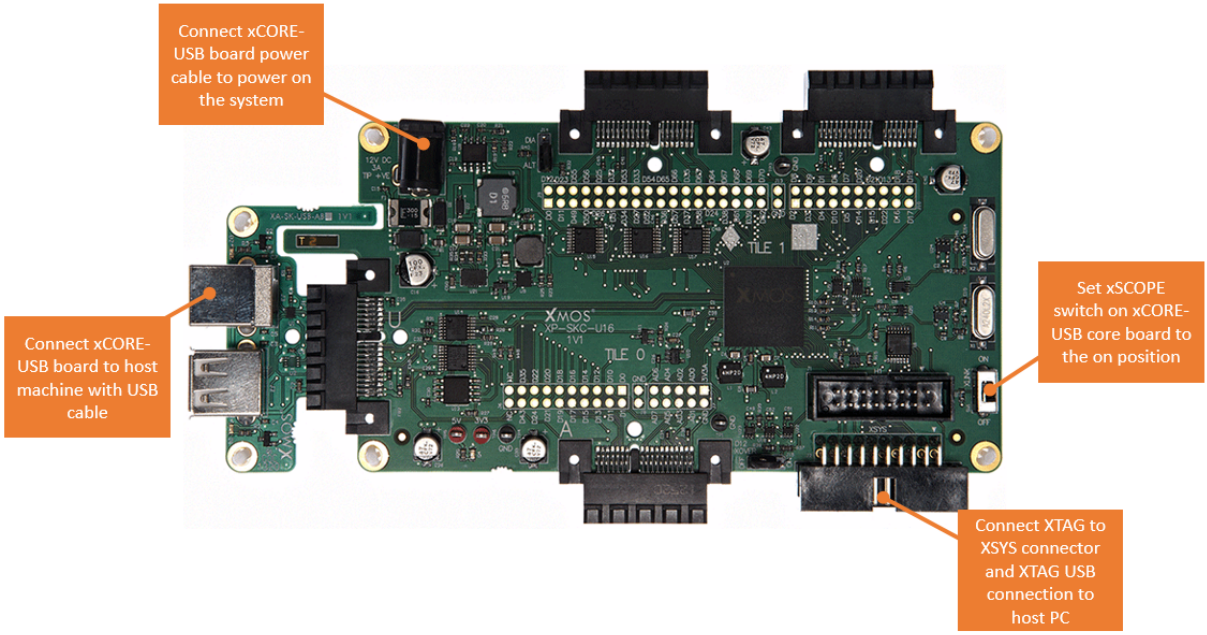

Figure 3: XMOS xCORE-USB sliceKIT

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 xCORE-USB core board should have the power cable connected

APPENDIX B - Host Application Setup

B.1 Test application

This simple host example demonstrates simple PTP transactions between the host processor and the XMOS device in bulk transfer mode.

The application simply sends commands for receiving the image size and the type and then sends a request to get the image. It receives a response for each command and also the image data. The received image data is stored in a file and displayed.

The application was tested on a 64-bit linux platform. The binary and 'libusb' library for this platform along with the source files of the application are provided in the `host` directory. For other platforms, please refer to the application note AN00136: USB Vendor Specific Device[6].

B.2 Licensing

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

B.3 Compilation instructions

If you require to recompile the test program then the instruction to do so is below,

Linux64:

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

[6] https://www.xmos.com/download/public/AN00136%3A-USB-Vendor-Specific-Device%281.0.0%29.pdf

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_usb_image_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 still image USB device should have enumerated on your machine. You can check this by executing 'lsusb' from the command line. It lists the device like the one below:

```
Bus 001 Device 007: ID 20b1:00c1 XMOS Ltd
```

C.3 Running the host demo

To run the example, navigate to 'host/Linux64' and execute './get_image' from the command line.

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

The demo prompts the user for the inputs - image size and type. The output of the demo is as below:

```
XMOS USB image device opened .....
Session opened ....
Image captured ....
Image info got ....
Image written to PNM and JPG files .....
Displaying image .....
Session closed ....
XMOS USB image device closed .....
```

The gradient image received by the host is first saved in PNM[7] format and then converted to a JPG file. The image is displayed. PNM is portable anymap format that was designed to be easily exchanged between platforms. It can be one of these: portable pixmap format (PPM), portable graymap format (PGM) and portable bitmap format (PBM). Sample output gray image files `image.pnm` and `image.jpg` are in the `host/Linux64` directory. The generated color and gray images are shown in Figures 4 and 5.

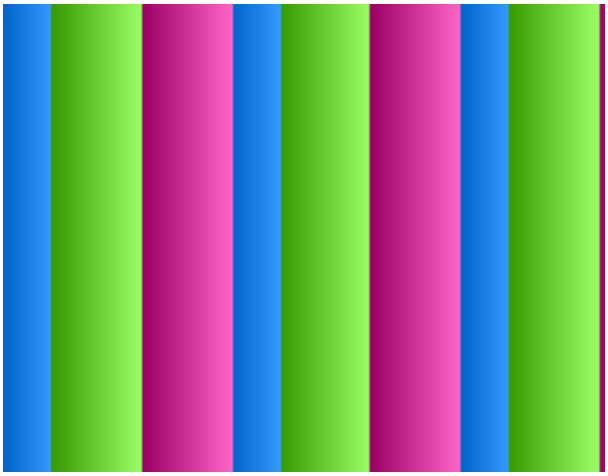

Figure 4: Color image

Figure 5: Gray image

[7] http://en.wikipedia.org/wiki/Netpbm_format

APPENDIX D - 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

USB Still Image Capture device class specification

http://www.usb.org/developers/docs/devclass_docs/usb_still_img10.zip

PIMA 15740 standard, Picture Transfer Protocol (PTP)

http://www.pima.net/standards/it10/IT10_POW.htm

`libusb` library

http://www.libusb.org

APPENDIX E - Full source code listing

E.1 Source code for endpoint0.xc

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

/*
 * @brief Implements endpoint zero for an example image acquisition device.
 */
#include <xs1.h>
#include <string.h>

#include "usb.h"

/* USB Device ID defines */
#define BCD_DEVICE   0x1000
#define VENDOR_ID    0x20B1
#define PRODUCT_ID   0x00C1

/* USB Still Image Capture Class defines */
#define STILL_IMAGE_SUBCLASS 0x01
#define STILL_IMAGE_PROTOCOL 0x01

/* Class-Specific Requests - bRequest values */
#define STILL_IMAGE_CANCEL_REQUEST      0x64
#define STILL_IMAGE_GET_EXT_EVENT_DATA  0x65
#define STILL_IMAGE_DEV_RESET_REQ       0x66
#define STILL_IMAGE_GET_DEV_STATUS      0x67

/* Device Descriptor */
static unsigned char devDesc[] =
{
    0x12,                   /* 0  bLength */
    USB_DESCTYPE_DEVICE,    /* 1  bdescriptorType */
    0x10,                   /* 2  bcdUSB */
    0x01,                   /* 3  bcdUSB */
    0x00,                   /* 4  bDeviceClass */
    0x00,                   /* 5  bDeviceSubClass */
    0x00,                   /* 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 */
    0x01,                   /* 14 iManufacturer */
    0x02,                   /* 15 iProduct */
    0x03,                   /* 16 iSerialNumber */
    0x01                    /* 17 bNumConfigurations */
};



static unsigned char cfgDesc[] = {
    /* Configuration Descriptor */
    0x09,                       /* 0  bLength */
    USB_DESCTYPE_CONFIGURATION, /* 1  bDescriptortype */
    0x27, 0x00,                 /* 2  wTotalLength = 39*/
    0x01,                       /* 4  bNumInterfaces */
    0x01,                       /* 5  bConfigurationValue */
    0x04,                       /* 6  iConfiguration */
    0x80,                       /* 7  bmAttributes - Bus powered */
    0xFA,                       /* 8  bMaxPower - 500 mA*/

    /* Interface Descriptor */
    0x09,                       /* 0  bLength */
    USB_DESCTYPE_INTERFACE,     /* 1  bDescriptorType */
    0x00,                       /* 2  bInterfacecNumber */
    0x00,                       /* 3  bAlternateSetting */
    0x03,                       /* 4: bNumEndpoints */
    USB_CLASS_IMAGE,            /* 5: bInterfaceClass */
    STILL_IMAGE_SUBCLASS,       /* 6: bInterfaceSubClass */
```

```
    STILL_IMAGE_PROTOCOL,       /* 7: bInterfaceProtocol*/
    0x00,                       /* 8  iInterface */

    /* Data-in Bulk Endpoint Descriptor */
    0x07,                       /* 0  bLength */
    USB_DESCTYPE_ENDPOINT,      /* 1  bDescriptorType */
    0x01,                       /* 2  bEndpointAddress - Data EP1 OUT */
    0x02,                       /* 3  bmAttributes - Bulk transfer*/
    0x40,                       /* 4  wMaxPacketSize - set to 64 */
    0x00,                       /* 5  wMaxPacketSize */
    0x00,                       /* 6  bInterval */

    /* Data-out Bulk Endpoint Descriptor */
    0x07,                       /* 0  bLength */
    USB_DESCTYPE_ENDPOINT,      /* 1  bDescriptorType */
    0x81,                       /* 2  bEndpointAddress - Data EP1 IN */
    0x02,                       /* 3  bmAttributes - Bulk transfer*/
    0x40,                       /* 4  wMaxPacketSize */
    0x00,                       /* 5  wMaxPacketSize */
    0x00,                       /* 6  bInterval */

    /* Interrupt Endpoint Descriptor */
    0x07,                       /* 0  bLength */
    USB_DESCTYPE_ENDPOINT,      /* 1  bDescriptorType */
    0x82,                       /* 2  bEndpointAddress - Data EP1 IN */
    0x03,                       /* 3  bmAttributes - Interrupt transfer*/
    0x40,                       /* 4  wMaxPacketSize */
    0x00,                       /* 5  wMaxPacketSize */
    0x01                        /* 6  bInterval */

};


unsafe{
/* String table */
static char * unsafe stringDescriptors[]=
{
    "\x09\x04",                 // Language ID string (US English)
    "XMOS",                     // iManufacturer
    "USB Still Image Capture",  // iProduct
    "0123456789",               // iSerialNumber
    "config",                   // iConfiguration
};}

/* Still Image Class-Specific Requests */
XUD_Result_t StillImageClassRequests(XUD_ep c_ep0_out, XUD_ep c_ep0_in, USB_SetupPacket_t sp)
{
    unsigned buffer[64];
    unsigned TransactionID, length;
    XUD_Result_t result;

    switch(sp.bRequest)
    {
        case STILL_IMAGE_CANCEL_REQUEST:
            /* Receives the transaction ID that was cancelled by the host */
            if((result = XUD_GetBuffer(c_ep0_out, (buffer, unsigned char[]), length)) != XUD_RES_OKAY)
            {
                return result;
            }
            memcpy (&TransactionID, &(buffer, unsigned char[])[2], 4);
            result = XUD_DoSetRequestStatus(c_ep0_in);
            return result;
            break;

        case STILL_IMAGE_GET_EXT_EVENT_DATA:
            /* Transfers the extended information on an asynchronous event stored in the buffer to the host */
            return XUD_DoGetRequest(c_ep0_out, c_ep0_in, (buffer, unsigned char[]), length, sp.wLength);
            break;

        case STILL_IMAGE_DEV_RESET_REQ:
            /* Put the device in the idle state */
            return XUD_RES_RST;
            break;

         case STILL_IMAGE_GET_DEV_STATUS:
```

```
            /* Transfers information regarding the status of the device */
            return XUD_DoGetRequest(c_ep0_out, c_ep0_in, (buffer, unsigned char[]), length, sp.wLength);
            break;

         default:
             /* Error */
             return XUD_RES_ERR;
             break;

    }

}


/* Endpoint 0 Task */
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);

             /* Handle specific requests first */
            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:

                    if(sp.wIndex == 0)
                        /* Returns  XUD_RES_OKAY if handled, XUD_RES_ERR if not handled,
                         *          XUD_RES_RST for bus reset */
                        result = StillImageClassRequests(ep0_out, ep0_in, sp);
                    break;
            }
        }

        /* If we haven't handled the request above then do standard enumeration requests */
        if(result == XUD_RES_ERR)
            /* Returns  XUD_RES_OKAY if handled okay, XUD_RES_RST if USB Reset,
             *          XUD_RES_ERR if request was not handled (i.e. STALLed) */
            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);
    }
}
```

E.2 Source code for main.xc

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

#include <string.h>
#include <stdint.h>
#include "usb.h"
#include "ptp.h"

/* USB endpoint defines */
#define XUD_EP_COUNT_OUT   2
#define XUD_EP_COUNT_IN    2

#define USB_DATA_PKT_SIZE 64    // USB data packet size

/* Image type */
enum {
    GRAY, COLOR
};
#define IMG_TYPE GRAY

/* Image pixel component starting values */
#define ST_GRAY 0
#define ST_RED 0
#define ST_GREEN 100
#define ST_BLUE 200

/* Image size */
#define IMG_HEIGHT 200
#define IMG_WIDTH 250

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


/*
 * This function responds to the USB image data and control requests from the host
 */

void bulk_endpoint(chanend chan_ep_from_host, chanend chan_ep_to_host)
{
    PTPContainer operation_response;
    PTPObjectInfo image_info;
    unsigned char cmd_buf[sizeof(PTPContainer)];
    unsigned char data_buf[USB_DATA_PKT_SIZE];
    unsigned char info_buf[sizeof(PTPObjectInfo)];
    unsigned cmd_length;
    XUD_Result_t result;

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


    while(1)
    {
        /* Receive the operation request from the host */
        if((result = XUD_GetBuffer(ep_from_host, cmd_buf, cmd_length)) == XUD_RES_RST)
        {
            XUD_ResetEndpoint(ep_from_host, ep_to_host);
            continue;
        }
        memcpy (&operation_response, cmd_buf, sizeof(PTPContainer));

        /* Process the command of custom simple protocol similar to PTP */
        uint16_t opcode = operation_response.Code;
        switch (opcode){

        case PTP_OC_OpenSession:
        case PTP_OC_InitiateCapture:
        case PTP_OC_CloseSession:
            operation_response.Code = PTP_RC_OK;
            break;

        case PTP_OC_GetObjectInfo:
            image_info.ObjectFormat = PTP_OFC_Undefined_0x3806;
```

```
            image_info.ImagePixHeight = IMG_HEIGHT;
            image_info.ImagePixWidth = IMG_WIDTH;
            if (IMG_TYPE==COLOR)
                image_info.ImageBitDepth = 24;
            else
                image_info.ImageBitDepth = 8;
            strcpy(image_info.Filename, "image.pnm");

            /* Send image info */
            memcpy (info_buf, &image_info, sizeof(PTPObjectInfo));
            if((result = XUD_SetBuffer(ep_to_host, info_buf, USB_DATA_PKT_SIZE)) == XUD_RES_RST)
            {
                XUD_ResetEndpoint(ep_from_host, ep_to_host);
                break;
            }
            if((result = XUD_SetBuffer(ep_to_host, info_buf+USB_DATA_PKT_SIZE, sizeof(PTPObjectInfo)-
              ↪ USB_DATA_PKT_SIZE)) == XUD_RES_RST)
            {
                XUD_ResetEndpoint(ep_from_host, ep_to_host);
                break;
            }

            operation_response.Code = PTP_RC_OK;
            break;

        case PTP_OC_GetObject:

            /* Generate image */
            int ncols = IMG_WIDTH;
            if (IMG_TYPE==COLOR) ncols = 3*IMG_WIDTH;

            int index = 0;
            int pkt_size = USB_DATA_PKT_SIZE;
            while (index < (IMG_HEIGHT*ncols)){

                if (((IMG_HEIGHT*ncols)-index) < USB_DATA_PKT_SIZE)
                    pkt_size = (IMG_HEIGHT*ncols)-index;
                for (int i=0; i<pkt_size; i++){
                    if (IMG_TYPE==COLOR){
                        switch (index%3){
                        case 0: data_buf[i] = ST_RED+(index%ncols); break;
                        case 1: data_buf[i] = ST_GREEN+(index%ncols); break;
                        case 2: data_buf[i] = ST_BLUE+(index%ncols); break;
                        }
                    }
                    else data_buf[i] = ST_GRAY+(index%ncols);
                    index++;
                }
                /* Send the image data packet to the host */
                if((result = XUD_SetBuffer(ep_to_host, data_buf, pkt_size)) == XUD_RES_RST)
                {
                    XUD_ResetEndpoint(ep_from_host, ep_to_host);
                    break;
                }
            }
            operation_response.Code = PTP_RC_OK;
            break;

        default:
            operation_response.Code = PTP_RC_Undefined;
            break;

       }

       /* Send response */
       operation_response.Nparam = 0;
       memcpy (cmd_buf, &operation_response, sizeof(PTPContainer));
       if((result = XUD_SetBuffer(ep_to_host, cmd_buf, sizeof(PTPContainer))) == XUD_RES_RST)
       {
           XUD_ResetEndpoint(ep_from_host, ep_to_host);
       }

   }
}
```

```
/* The main function runs three cores: the XUD manager, Endpoint 0, and USB image endpoints.
 * An array of channels is used for both IN and OUT endpoints, endpoint zero requires both,
 * USB bulk data IN and OUT endpoints are used to receive operation request and send image data and response.
 */
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;
}
```