Application Note: AN00135

USB Test and Measurement Device

This application note shows how to create a USB Test and Measurement class device on an XMOS multicore microcontroller.

The code associated with this application note uses the XMOS USB Device Library and associated USB class descriptors to create a standard USB test and measurement class (USBTMC) device running over high speed USB. The code supports the minimal standard requests associated with this class of USB devices.

The application demonstrates VISA compliant USBTMC client host software (such as NI LabVIEW, NI MAX, pyUsbtmc etc.) request test and measurement data using a subset of SCPI commands implemented on xCORE device. The application also integrates an open source SCPI library and thus provides a framework to implement the needed SCPI commands easily on a USBTMC xCORE device.

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

1 Overview

1.1 Introduction

The USB Test and Measurement class (USBTMC) specification allows the test and measurement devices with a USB interface so that messages can be exchanged between the host and devices. The subclass specification communicates to the devices based on IEEE 488.1 (GPIB) and IEEE 488.2 (Standard Commands for Programmable Instruments - SCPI) standards.

These types of devices enumerate as Test and Measurement class USB devices on the host. A USBTMC device must be able to support the control endpoint, Bulk-OUT and Bulk-IN endpoints. The Interrupt-IN endpoint is optional and is used by the device to send notifications to the Host.

Examples of test and measurement devices includes

- ADCs, DACs, Sensors, DAQs
- Digital instrument cards, IEE-488 based communication devices

Each USBTMC device implements its own command set based on USBTMC subclass (IEEE488) standards. Host can use any VISA compliant application to send these commands to the device and read the response from the device.

1.2 Block diagram

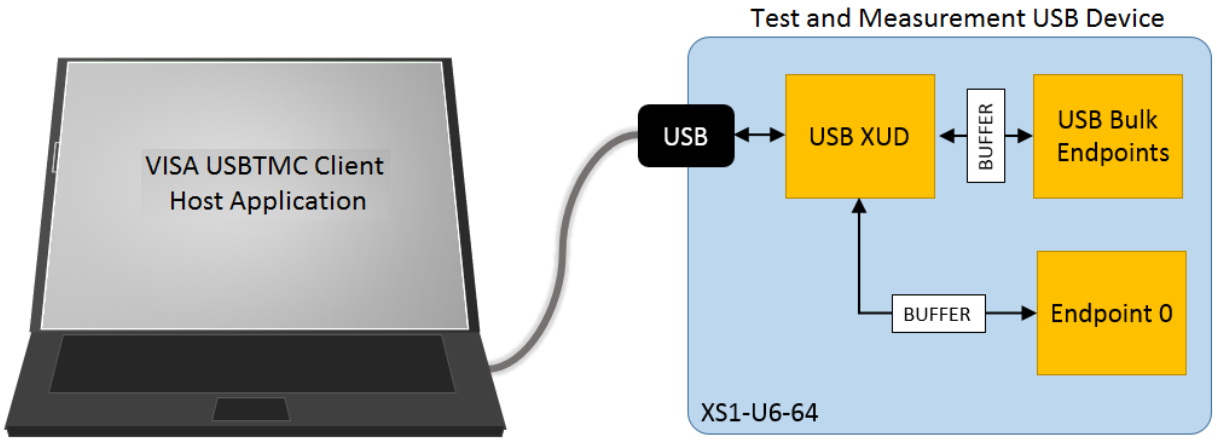

Figure 1: Block diagram of USBTMC device application example

2 USB Test and Measurement Class Device application note

The application in this note uses the XMOS USB device library and creates a USBTMC device which responds to IVI-VISA compliant host application messages. The device implements basic message requests for Bulk-IN and Bulk-OUT endpoints. Interrupt-IN endpoint is optional and is not used in this application. The application integrates an open source SCPI parser library and implements a minimal subset of SCPI commands to communicate with host. More SCPI commands can be easily added to this application by following the steps detailed in the `Using SCPI library to add more SCPI commands` appendix section.

For the USBTMC device class application example, the system comprises three tasks running on separate logical cores of an xCORE-USB multicore microcontroller.

The tasks perform the following operations.

- A task containing the USB library functionality to communicate over USB
- A task implementing Endpoint0 responding to standard USB control requests
- A task implementing the USBTMC class specific message handling application code for both Bulk-IN and Bulk-OUT endpoints

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 USBTMC application example.

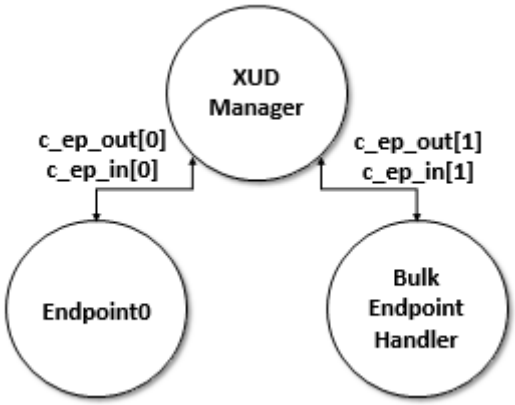

Figure 2: Task diagram of USBTMC device

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 xud.h header file:

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

2.2 Declaring resource and setting up the USB components

main.xc contains the logical cores instantiation needed for USBTMC device. There are some defines in it that are used to configure the XMOS USB device library. These are displayed below.

This set of defines describes the endpoint configuration for this device. This example has bi-directional communication with the host machine via the standard endpoint0 and endpoints for implementing the bulk-in and bulk-out endpoints.

```
/* USB Endpoint Defines */
#define XUD_EP_COUNT_OUT   2    //Includes EP0 (1 out EP0 + USBTMC data output EP)
#define XUD_EP_COUNT_IN    2    //Includes EP0 (1 in EP0)

/* Prototype for Endpoint0 function in endpoint0.xc */
```

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: usbtmc_bulk_endpoints(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 start up and run the Endpoint0 code: Endpoint0()
- There is a function to deal with the USBTMC device custom bulk endpoints usbtmc_bulk_endpoints()
- 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.

```
#define BCD_DEVICE   0x0126
#define VENDOR_ID    0x20B1
#define PRODUCT_ID   0x2337

/* Vendor specific class defines */
#define VENDOR_SPECIFIC_CLASS    0x00
#define VENDOR_SPECIFIC_SUBCLASS 0x00
#define VENDOR_SPECIFIC_PROTOCOL 0x00

#define MANUFACTURER_STR_INDEX  0x0001
#define PRODUCT_STR_INDEX       0x0002
```

2.5 USBTMC Class specific defines

The USBTMC 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 USBTMC device running on an xCORE-USB microcontroller.

```
#define VENDOR_SPECIFIC_CLASS    0x00
#define VENDOR_SPECIFIC_SUBCLASS 0x00
#define VENDOR_SPECIFIC_PROTOCOL 0x00
```

These are defined in the USB standard as required in the device description for TMC class 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 USBTMC 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. This descriptor contains the vendor specific defines described above.

```
static unsigned char devDesc[] =
{
    0x12,                   /* 0  bLength */
    USB_DESCTYPE_DEVICE,    /* 1  bdescriptorType */
    0x00,                   /* 2  bcdUSB version */
    0x02,                   /* 3  bcdUSB version */
    VENDOR_SPECIFIC_CLASS, /* 4  bDeviceClass - Specified by interface */
    VENDOR_SPECIFIC_SUBCLASS, /* 5  bDeviceSubClass  - Specified by interface */
    VENDOR_SPECIFIC_PROTOCOL, /* 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 */
    MANUFACTURER_STR_INDEX, /* 14 iManufacturer - index of string*/
    PRODUCT_STR_INDEX,      /* 15 iProduct  - index of string*/
    0x05,                   /* 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 our USBTMC 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 USBTMC device provided in this example the configuration descriptor which is read by the host is as follows.

```
static unsigned char cfgDesc[] = {
    /* */
    0x09,                   /* 0  bLength */
    USB_DESCTYPE_CONFIGURATION, /* 1  bDescriptortype = configuration*/
    0x20, 0x00,             /* 2  wTotalLength of all descriptors */
    0x01,                   /* 4  bNumInterfaces */
    0x01,                   /* 5  bConfigurationValue */
    0x00,                   /* 6  iConfiguration - index of string*/
    0x80,                   /* 7  bmAttributes - Self powered*/
    0x64,                   /* 8  bMaxPower - 200mA */

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

    /* */
    0x07,                   /* 0  bLength */
    USB_DESCTYPE_ENDPOINT,/* 1  bDescriptorType */
    0x01,                   /* 2  bEndpointAddress - EP2, OUT*/
    XUD_EPTYPE_BUL,         /* 3  bmAttributes */
    0x00,                   /* 4  wMaxPacketSize - Low */
    0x02,                   /* 5  wMaxPacketSize - High */
    0x01,                   /* 6  bInterval */

    /* */
    0x07,                   /* 0  bLength */
    USB_DESCTYPE_ENDPOINT,/* 1  bDescriptorType */
    0x81,                   /* 2  bEndpointAddress - EP1, IN*/
    XUD_EPTYPE_BUL,         /* 3  bmAttributes */
    0x00,                   /* 4  wMaxPacketSize - Low */
    0x02,                   /* 5  wMaxPacketSize - High */
    0x01,                   /* 6  bInterval */

};
```

This descriptor is in the format described by the USB 2.0 standard and contains the encoding for the endpoints related to control endpoint 0 and also the descriptors that describe the 2 bulk endpoints which form our USBTMC device. Note that bInterfaceProtocol value is controlled by the define USBTMC_SUB_CLASS_SUPPORT and it is set to 0 implying that no subclass specification applies and the USBTMC interface is not required to have an Interrupt-IN endpoint for this application example.

2.8 USB string descriptors

The final descriptor for our USBTMC 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 as accessed via shared memory */
static char * unsafe stringDescriptors[]=
{
  "\x09\x04",             // Language ID string (US English)
  "XMOS",                 // iManufacturer
  "XMOS TestMeasuement device",          // iProduct
  "USBTMC",                       // iInterface
  "Config",           // iConfiguration string
  "XD0701MQFZkDt_"
};
```

The last value corresponds to the iSerialNumber of the device, which should contain a value > 0 to form a valid GUID.

2.9 USBTMC 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.

```
/* Endpoint 0 Task */
void Endpoint0(chanend chan_ep0_out, chanend chan_ep0_in)
{
    USB_SetupPacket_t sp;
    XUD_BusSpeed_t usbBusSpeed;
    unsigned bmRequestType;

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

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

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

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

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

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

                    /* Check for USBTMC interface number (Sec 4.2.1.8) */
                    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;

                    /* Direction: Device-to-host and Host-to-device
                     * Type: Class
                     * Recipient: Endpoint
                     */
                    case USB_BMREQ_H2D_CLASS_EP:
                    case USB_BMREQ_D2H_CLASS_EP:

                        /* Check for USBTMC interface number (Sec 4.2.1.8) */
                        if(sp.wIndex == 0)
                        {
                            /* Returns  XUD_RES_OKAY if handled,
                             *          XUD_RES_ERR if not handled,
                             *          XUD_RES_RST for bus reset */
                            result = ControlEndpointClassRequests(ep0_out, ep0_in, sp);
                        }
                        break;
            }
        } //if(result == XUD_RES_OKAY)
```

The control interface class requests are handled using the below function.

```
XUD_Result_t ControlInterfaceClassRequests(XUD_ep ep_out, XUD_ep ep_in, USB_SetupPacket_t sp)
{
    XUD_Result_t result = XUD_RES_ERR;

    static struct dev_capabilities {
        unsigned char USBTMC_status;
        unsigned char reserved_1;
        unsigned char bcdUSBTMC[2];
        unsigned char USBTMC_Int_Capabilities;
        unsigned char USBTMC_Dev_Capabilities;
        unsigned char reserved_2[6];
        unsigned char reserved_3[12];
    } dev_capabilities;

    switch(sp.bRequest)
    {
        case INITIATE_CLEAR:
            /* Add reuest specific functionality (4.2.1.6) here */
            return result;
            break;

        case CHECK_CLEAR_STATUS:
            /* Add reuest specific functionality (4.2.1.7) here */
            return result;
            break;

        case GET_CAPABILITIES:
            dev_capabilities.USBTMC_status = 0x01; //SUCCESS
            dev_capabilities.bcdUSBTMC[0] = 0x00;
            dev_capabilities.bcdUSBTMC[1] = 0x02;
            dev_capabilities.USBTMC_Int_Capabilities = 0x00;
            dev_capabilities.USBTMC_Dev_Capabilities = 0x00;

            /* Send the response buffer to the host */
            if ((result = XUD_DoGetRequest(ep_out, ep_in, (dev_capabilities, char[]), sizeof(dev_capabilities)
              ↪ , sp.wLength)) != XUD_RES_OKAY)
            {
                return result;
            }

            return result;
            break;

        case INDICATOR_PULSE:
            /* Add reuest specific functionality (4.2.1.9) here */
            return result;
            break;

      default:
            break;
    }

    return XUD_RES_ERR;
```

This application implements GET_CAPABILITIES request and uses it to send our device capabilities to the host.

2.10 Handling requests to the USBTMC bulk endpoints

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

```
void usbtmc_bulk_endpoints(chanend c_ep_out,chanend c_ep_in)
{
    unsigned char host_transfer_buf[BUFFER_SIZE];
    unsigned host_transfer_length = BUFFER_SIZE;//0;
    unsigned char scpi_cmd[SCPI_CMD_BUF_SIZE];
    XUD_Result_t result;
    int scpi_parse_res = 0;
    unsigned scpi_cmd_len = 0;
    unsigned char msg_id;   //MsgID field with Offset 0

    /* Initialise the XUD endpoints */
    XUD_ep ep_out = XUD_InitEp(c_ep_out, XUD_EPTYPE_BUL);
    XUD_ep ep_in  = XUD_InitEp(c_ep_in, XUD_EPTYPE_BUL);

    /* Mark OUT endpoint as ready to receive */
    XUD_SetReady_Out (ep_out, host_transfer_buf);

    SCPI_initialize_parser();

    while(1)
    {
        select
        {
            case XUD_GetData_Select(c_ep_out, ep_out, host_transfer_length, result):
                /* Packet from host recieved */
                if (result == XUD_RES_RST) {
                    XUD_ResetEndpoint(ep_in, null);
                }
                msg_id = host_transfer_buf[0];

                switch (msg_id) {
                    case DEV_DEP_MSG_OUT:
                    {
                        scpi_cmd_len = host_transfer_buf[4];    //TODO: Handle 4-byte TransferSize for
                          ↪ scpi_cmd_len
                        SCPI_get_cmd(&host_transfer_buf[12], scpi_cmd_len, scpi_cmd);
                    }
                    break;

                    case REQUEST_DEV_DEP_MSG_IN:
                    {
                        /* Prepare response message buffer */
                        scpi_parse_res = SCPI_parse_cmd(scpi_cmd, scpi_cmd_len, &host_transfer_buf[12],
                          ↪ host_transfer_length);
                        host_transfer_buf[4] = host_transfer_length;
                        host_transfer_buf[5] = 0x00;
                        host_transfer_buf[6] = 0x00;
                        host_transfer_buf[8] = 0x01;    //Set (EOM=1)
                        host_transfer_length += 12;     //Bulk-IN header size
                        host_transfer_buf[host_transfer_length++] = 0x00; //Set Alighnment byte

                        XUD_SetReady_In (ep_in, host_transfer_buf, host_transfer_length);
                    }
                    break;

                    case VENDOR_SPECIFIC_OUT:
                        /* Handle any vendor specific command messages */
                    break;

                    case REQUEST_VENDOR_SPECIFIC_IN:
                        /* Handle any vendor specific command messages */
                    break;

                    default:
                    break;
                }

                /* Mark EP as ready again */
                XUD_SetReady_Out (ep_out, host_transfer_buf);
            break ;

            case XUD_SetData_Select(c_ep_in, ep_in, result):
```

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.
- Bulk-OUT endpoint is set as ready to receive the data from the host
- SCPI parser library is initialized using SCPI`_initialize_parser` wrapper api call
- This task operates inside a `while (1)` loop which has a select handler to repeatedly deal with events related to bulk endpoints
- The select event handler gets the data from bulk out endpoint, decodes the USBTMC device dependent command messages using SCPI command parser
- The function SCPI`_parse_cmd` parses the USBTMC device dependent command message, prepares a response message and sends the response using the bulk-in endpoint using XUD`_SetReady_In`
- The Bulk-OUT endpoint is set as ready once again to receive the data from the host
- This vendor specific command requests and responses could also be handled in a similar manner

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.

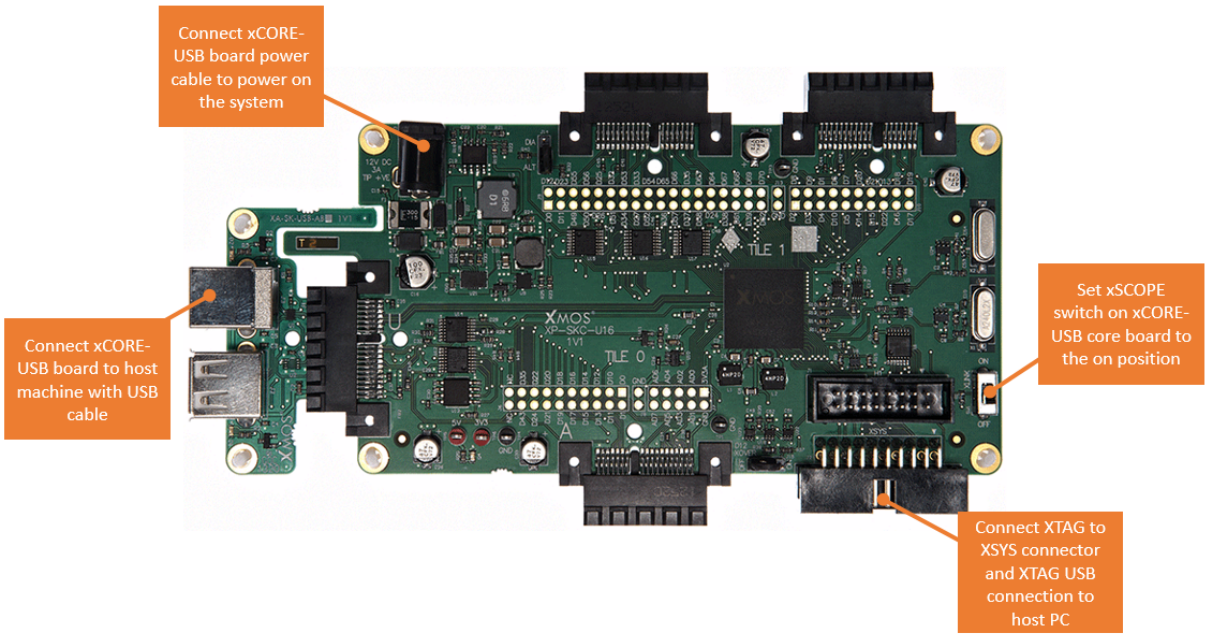

Figure 3: XMOS xCORE-USB sliceKIT

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

APPENDIX B - Launching the demo application

Once the demo example has been built either from the command line using xmake or via the build mechanism of xTIMEcomposer studio the application 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.

B.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_tmc_demo.xe          <-- Download and execute the xCORE code
```

Once this command has executed the USBTMC device should have enumerated on the host machine

B.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 USBTMC device should have enumerated on your machine

B.3 Running the USBTMC demo

- Ensure the USBTMC device enumeration on a Linux (Ubuntu 12.04 LTS) host is fine:

```
xmos@xmos:~$ lsusb
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 012: ID 20b1:2337 XMOS Ltd
```

- List the details of the device descriptor:

```
root@xmos-lenovo:/home/xmos# lsusb -v
> lsusb -v
Bus 001 Device 012: ID 20b1:2337 XMOS Ltd
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB               2.00
  bDeviceClass            0 (Defined at Interface level)
  .
  .
  idVendor           0x20b1 XMOS Ltd
  idProduct          0x2337
  bcdDevice            1.26
  iManufacturer           1 XMOS
  iProduct                2 XMOS TestMeasuement device
  .
  .
Interface Descriptor:
  .
  .
  bInterfaceClass       254 Application Specific Interface
  bInterfaceSubClass      3 Test and Measurement
  bInterfaceProtocol      0
  iInterface              3 USBTMC
  Endpoint Descriptor:
  .
  .
  bEndpointAddress     0x01  EP 1 OUT
  bmAttributes            2
    Transfer Type            Bulk
  .
  .
bEndpointAddress     0x81  EP 1 IN
bmAttributes            2
    Transfer Type            Bulk
  .
  .
```

- Open a *Terminal* window (using root privileges) and communicate with the XMOS USBTMC device:

```
root@xmos-lenovo:/home/xmos# python
Python 2.7.3 (default, Aug  1 2012, 05:16:07)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import usb.core
>>> import usb.util
>>>
>>> dev = usb.core.find(idVendor=0x20b1, idProduct=0x2337)
>>> import usbtmc
>>> instr =  usbtmc.Instrument(0x20b1, 0x2337)
```

- Identify the USBTMC device using IDN query:

```
>>>
>>> print(instr.ask("*IDN?"))
XMOS, USBTMC, 1, 01-01

>>>
>>> print(instr.ask("*RST"))
SCPI command not implemented
>>>
```

- Query the DC voltage of the device using the SCPI command as follows:

```
>>> print(instr.ask("*MEASure:VOLTage:DC?"))
10
```

Note: The DC voltage measurement function is stubbed inside the device code to return a sample voltage value. This logic can be replaced with instrument data such as actual ADC results.

APPENDIX C - Using NI LabVIEW to communicate with XMOS USBTMC device

NI LabVIEW software can communicate with USBTMC devices using VISA functions (Read, Write, Open, Close) in the same way as you would communicate with GPIB instruments. You can create a simple block diagram as below.

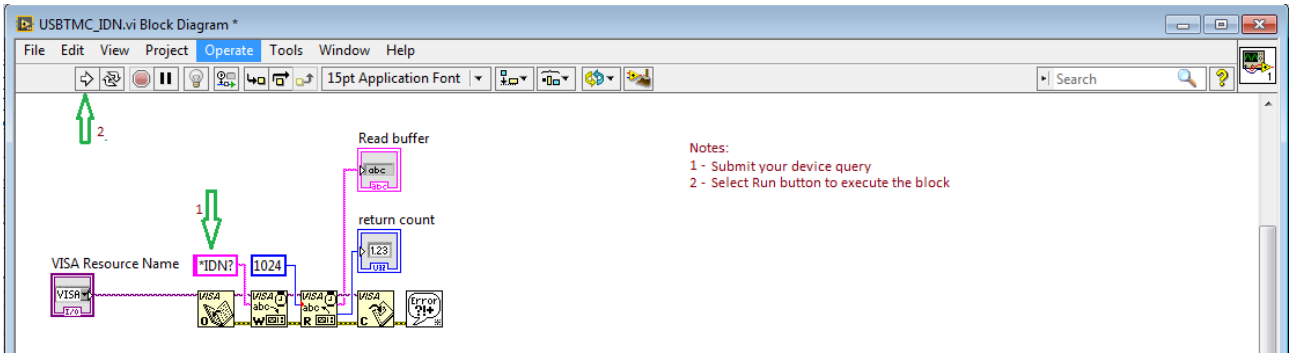

Figure 4: NI LabVIEW block diagram to communicate to XMOS USBTMC device

This example opens a VISA session, uses a SCPI command to query the ID of the device and the response is read back before closing the session. To test this block, connect your device to a host and ensure it successfully enumerates as USBTMC device as follows.

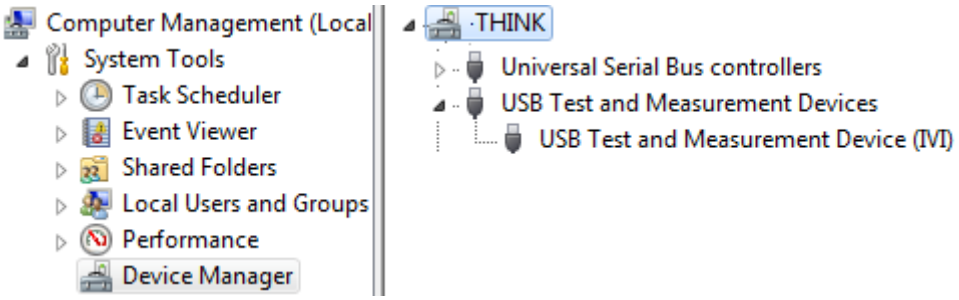

Figure 5: XMOS device enumerated as USBTMC device on a Windows 7 host

Navigate to the NI LabVIEW front panel window of the block diagram and select XMOS USBTMC device in the VISA IO resource listing.

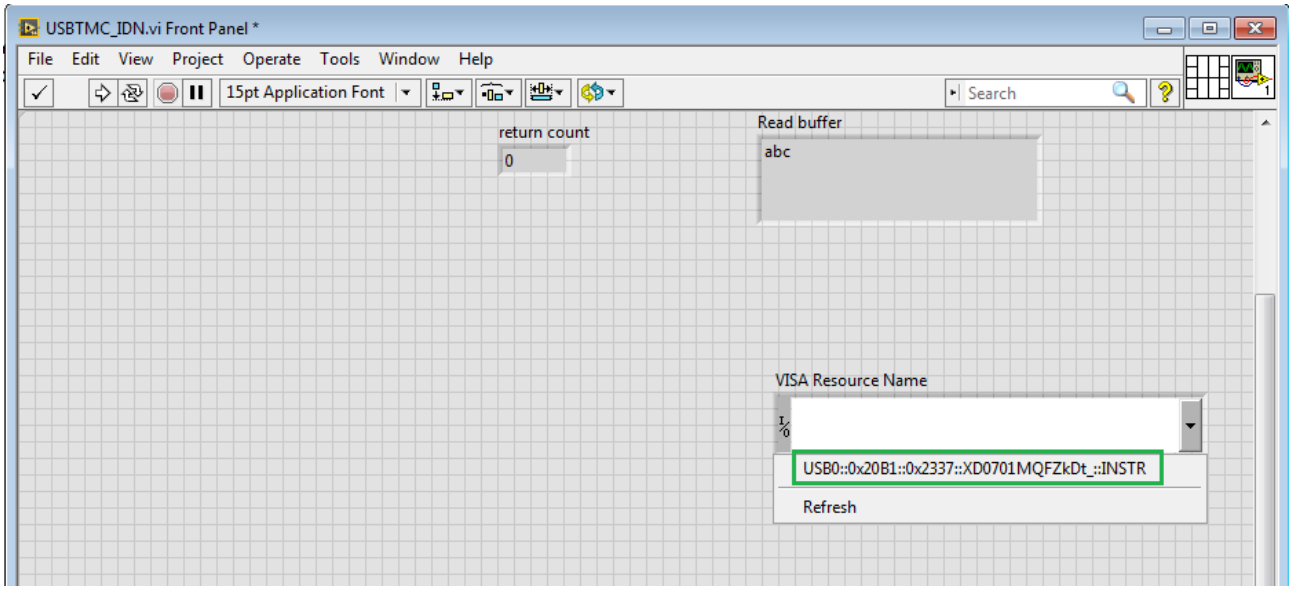

Figure 6: NI LabVIEW front panel to select XMOS USBTMC device

Click *Run* button (Right arrow). The device id of the manufacturer, which is *XMOS USBTMC device with version number* in this case is populated in the results buffer as follows.

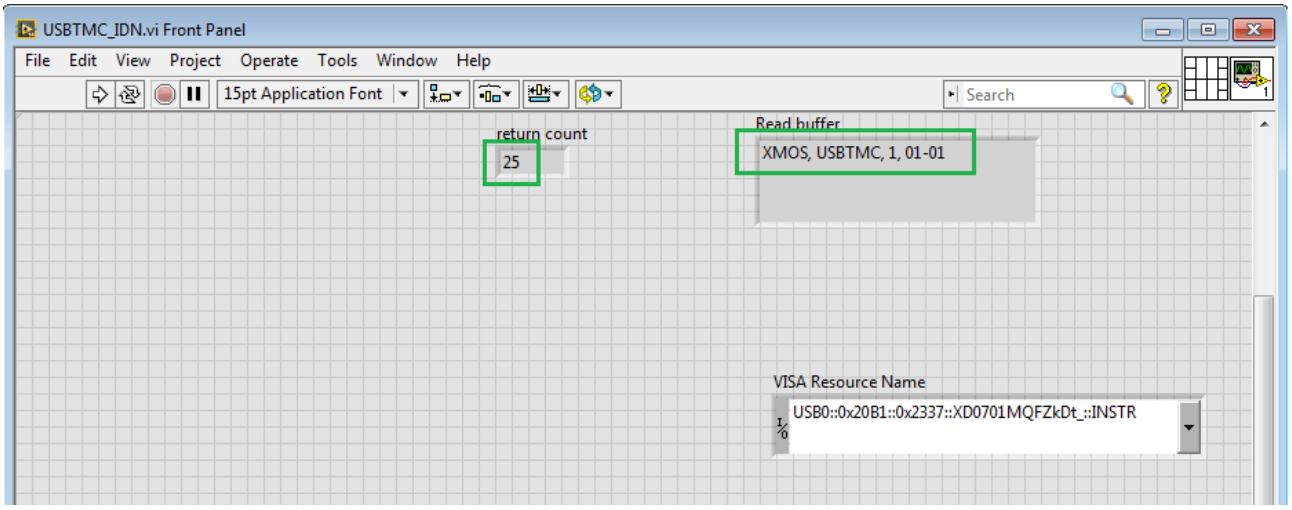

Figure 7: NI LabVIEW front panel displaying results from XMOS USBTMC device

Similarly you can alter the query from the block diagram to contain a different SCPI command (for e.g., `*MEASure:VOLTage:DC?`), submit it to your device and record the new results.

APPENDIX D - Using SCPI library to add more SCPI commands

The application supports very minimum SCPI commands on the device. Adding more SCPI commands is a simple configuration because the device code ports an Open source SCPI library. In order to add more SCPI commands, follow the below steps.

1. Navigate to `src\scpi_parser\custom\scpi_parser_config.c` file
2. Add SCPI command to the `scpi_commands` structure and attach a callback function
3. Declare the callback function in `src\scpi_parser\custom\scpi_cmds.h` file
4. Source file `src\scpi_parser\custom\scpi_cmds.c` contains the definition of callback functions. These functions interface with device measurement logic in order to capture the instrument results. Define the callback function logic for the configured SCPI command in this file.
5. When the USBTMC host client software uses the corresponding SCPI command, this callback function is invoked and the computed result is sent back to the host as response message using the existing framework.

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

USB Test and Measurement Class Specification (USBTMC); Rev 1.0

http://sdpha2.ucsd.edu/Lab_Equip_Manuals/USBTMC_1_00.pdf

USB Test and Measurement Class, Subclass USB488 Specification (USBTMC-USB488); Rev 1.0

http://sdpha2.ucsd.edu/Lab_Equip_Manuals/usbtmc_usb488_subclass_1_00.pdf

Open Source SCPI device library

https://github.com/j123b567/scpi-parser

APPENDIX F - Full source code listing

F.1 Source code for endpoint0.xc

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

/*
 * @brief Implements endpoint zero for an example USBTMC test device class
 */

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

/* USB Device ID Defines */
#define BCD_DEVICE   0x0126
#define VENDOR_ID    0x20B1
#define PRODUCT_ID   0x2337

/* Vendor specific class defines */
#define VENDOR_SPECIFIC_CLASS    0x00
#define VENDOR_SPECIFIC_SUBCLASS 0x00
#define VENDOR_SPECIFIC_PROTOCOL 0x00

#define MANUFACTURER_STR_INDEX  0x0001
#define PRODUCT_STR_INDEX       0x0002

/*0 - USBTMC interface. No subclass specification applies. */
/*1 - USBTMC USB488 interface; not supported in this demo */
#define USBTMC_SUB_CLASS_SUPPORT    0x00

/* 4. Requests */
/* 4.2 Class-Specific Requests - bRequest values */
#define INITIATE_ABORT_BULK_OUT     0x01
#define CHECK_ABORT_BULK_OUT_STATUS 0x02
#define INITIATE_ABORT_BULK_IN      0x03
#define CHECK_ABORT_BULK_IN_STATUS  0x04
#define INITIATE_CLEAR              0x05
#define CHECK_CLEAR_STATUS          0x06
#define GET_CAPABILITIES            0x07
/* 8 - 63 reserved */
#define INDICATOR_PULSE             0x40    /* Optional */

/* 4.3 USB488 subclass specific requests */
#define READ_STATUS_BYTE        0x80        /* Required. 128 - Returns the IEEE 488 Status Byte */
#define REN_CONTROL             0xA0        /* Optional. 160 - Mechanism to enable or disable local controls
  ↪ on a device */
#define GO_TO_LOCAL             0xA1        /* Optional. 161 - Mechanism to enable local controls on a device
  ↪ */
#define LOCAL_LOCKOUT           0xA2        /* Optional. 162 - Mechanism to disable local controls on a device
  ↪  */

/* Table 16 -- USBTMC_status values */
typedef enum USBTMC_status
{
    STATUS_SUCCESS = 0x01,
    STATUS_PENDING = 0x02,
    STATUS_FAILED = 0x80,
    STATUS_TRANSFER_NOT_IN_PROGRESS = 0x81,
    STATUS_SPLIT_NOT_IN_PROGRESS = 0x82,
    STATUS_SPLIT_IN_PROGRESS = 0x83,
} USBTMC_status_t;

/* USB Device Descriptor */
static unsigned char devDesc[] =
{
    0x12,                   /* 0  bLength */
    USB_DESCTYPE_DEVICE,    /* 1  bdescriptorType */
    0x00,                   /* 2  bcdUSB version */
    0x02,                   /* 3  bcdUSB version */
    VENDOR_SPECIFIC_CLASS, /* 4  bDeviceClass - Specified by interface */
    VENDOR_SPECIFIC_SUBCLASS, /* 5  bDeviceSubClass  - Specified by interface */
    VENDOR_SPECIFIC_PROTOCOL, /* 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 */
    MANUFACTURER_STR_INDEX, /* 14 iManufacturer - index of string*/
    PRODUCT_STR_INDEX,      /* 15 iProduct  - index of string*/
    0x05,                   /* 16 iSerialNumber  - index of string*/
    0x01                    /* 17 bNumConfigurations */
};

/* USB Configuration Descriptor */
static unsigned char cfgDesc[] = {
    /* */
    0x09,                 /* 0  bLength */
    USB_DESCTYPE_CONFIGURATION, /* 1  bDescriptortype = configuration*/
    0x20, 0x00,           /* 2  wTotalLength of all descriptors */
    0x01,                 /* 4  bNumInterfaces */
    0x01,                 /* 5  bConfigurationValue */
    0x00,                 /* 6  iConfiguration - index of string*/
    0x80,                 /* 7  bmAttributes - Self powered*/
    0x64,                 /* 8  bMaxPower - 200mA */

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

    /* */
    0x07,                 /* 0  bLength */
    USB_DESCTYPE_ENDPOINT,/* 1  bDescriptorType */
    0x01,                 /* 2  bEndpointAddress - EP2, OUT*/
    XUD_EPTYPE_BUL,       /* 3  bmAttributes */
    0x00,                 /* 4  wMaxPacketSize - Low */
    0x02,                 /* 5  wMaxPacketSize - High */
    0x01,                 /* 6  bInterval */

    /* */
    0x07,                 /* 0  bLength */
    USB_DESCTYPE_ENDPOINT,/* 1  bDescriptorType */
    0x81,                 /* 2  bEndpointAddress - EP1, IN*/
    XUD_EPTYPE_BUL,       /* 3  bmAttributes */
    0x00,                 /* 4  wMaxPacketSize - Low */
    0x02,                 /* 5  wMaxPacketSize - High */
    0x01,                 /* 6  bInterval */

};

unsafe{
  /* String table - unsafe as accessed via shared memory */
  static char * unsafe stringDescriptors[]=
  {
    "\x09\x04",             // Language ID string (US English)
    "XMOS",                 // iManufacturer
    "XMOS TestMeasuement device",         // iProduct
    "USBTMC",                       // iInterface
    "Config",           // iConfiguration string
    "XD0701MQFZkDt_"
  };
}

XUD_Result_t ControlInterfaceClassRequests(XUD_ep ep_out, XUD_ep ep_in, USB_SetupPacket_t sp)
{
    XUD_Result_t result = XUD_RES_ERR;

    static struct dev_capabilities {
        unsigned char USBTMC_status;
```

```
        unsigned char reserved_1;
        unsigned char bcdUSBTMC[2];
        unsigned char USBTMC_Int_Capabilities;
        unsigned char USBTMC_Dev_Capabilities;
        unsigned char reserved_2[6];
        unsigned char reserved_3[12];
    } dev_capabilities;

    switch(sp.bRequest)
    {
        case INITIATE_CLEAR:
            /* Add reuest specific functionality (4.2.1.6) here */
            return result;
            break;

        case CHECK_CLEAR_STATUS:
            /* Add reuest specific functionality (4.2.1.7) here */
            return result;
            break;

        case GET_CAPABILITIES:
            dev_capabilities.USBTMC_status = 0x01; //SUCCESS
            dev_capabilities.bcdUSBTMC[0] = 0x00;
            dev_capabilities.bcdUSBTMC[1] = 0x02;
            dev_capabilities.USBTMC_Int_Capabilities = 0x00;
            dev_capabilities.USBTMC_Dev_Capabilities = 0x00;

            /* Send the response buffer to the host */
            if ((result = XUD_DoGetRequest(ep_out, ep_in, (dev_capabilities, char[]), sizeof(dev_capabilities)
              ↪ , sp.wLength)) != XUD_RES_OKAY)
            {
                return result;
            }

            return result;
            break;

        case INDICATOR_PULSE:
            /* Add reuest specific functionality (4.2.1.9) here */
            return result;
            break;

      default:
            break;
    }

    return XUD_RES_ERR;
}

XUD_Result_t ControlEndpointClassRequests(XUD_ep ep_out, XUD_ep ep_in, USB_SetupPacket_t sp)
{
    USBTMC_status_t result = STATUS_FAILED;

    switch(sp.bRequest)
    {
        case INITIATE_ABORT_BULK_OUT:
            /* Parse request specific setup packet and return EP0 response packet (4.2.1.2) */
            return result;
            break;

        case CHECK_ABORT_BULK_OUT_STATUS:
            /* Add reuest specific functionality (4.2.1.3) here */
            return result;
            break;

        case INITIATE_ABORT_BULK_IN:
            /* Add reuest specific functionality (4.2.1.4) here */
            return result;
            break;

        case CHECK_ABORT_BULK_IN_STATUS:
            /* Add reuest specific functionality (4.2.1.5) here */
            return result;
            break;
```

```
      default:
            break;
    }

    return XUD_RES_ERR;
}

/* Endpoint 0 Task */
void Endpoint0(chanend chan_ep0_out, chanend chan_ep0_in)
{
    USB_SetupPacket_t sp;
    XUD_BusSpeed_t usbBusSpeed;
    unsigned bmRequestType;

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

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

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

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

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

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

                    /* Check for USBTMC interface number (Sec 4.2.1.8) */
                    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;

                    /* Direction: Device-to-host and Host-to-device
                     * Type: Class
                     * Recipient: Endpoint
                     */
                    case USB_BMREQ_H2D_CLASS_EP:
                    case USB_BMREQ_D2H_CLASS_EP:

                        /* Check for USBTMC interface number (Sec 4.2.1.8) */
                        if(sp.wIndex == 0)
                        {
                            /* Returns  XUD_RES_OKAY if handled,
                             *          XUD_RES_ERR if not handled,
                             *          XUD_RES_RST for bus reset */
                            result = ControlEndpointClassRequests(ep0_out, ep0_in, sp);
                        }
                        break;
            }
        } //if(result == XUD_RES_OKAY)
```

```
        /* If we haven't handled the request about then do standard enumeration requests */
        if(result == XUD_RES_ERR )
        {
            /* Returns  XUD_RES_OKAY if handled okay,
             * XUD_RES_ERR if request was not handled (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 usbtmc_endpoints.xc

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

#include "usb.h"
#include "scpi_parser_wrapper.h"

#define BUFFER_SIZE 128 //512   //8
#define SCPI_CMD_BUF_SIZE   40

/* 3.2 Bulk-OUT endpoint */
#define DEV_DEP_MSG_OUT             0x01
#define REQUEST_DEV_DEP_MSG_IN      0x02
#define VENDOR_SPECIFIC_OUT         0x7E    /* MsgId = 126 */
#define REQUEST_VENDOR_SPECIFIC_IN  0x7F    /* MsgId = 127 */
#define TRIGGER                     0x80    /* MsgId = 128 */

/* 3.3 Bulk-IN endpoint */
#define DEV_DEP_MSG_IN              0x02    /* MsgId = 002 */
#define VENDOR_SPECIFIC_IN          0x7F    /* MsgId = 127 */

/* This function receives the USBTMC endpoint transfers from the host */
void usbtmc_bulk_endpoints(chanend c_ep_out,chanend c_ep_in)
{
    unsigned char host_transfer_buf[BUFFER_SIZE];
    unsigned host_transfer_length = BUFFER_SIZE;//0;
    unsigned char scpi_cmd[SCPI_CMD_BUF_SIZE];
    XUD_Result_t result;
    int scpi_parse_res = 0;
    unsigned scpi_cmd_len = 0;
    unsigned char msg_id;   //MsgID field with Offset 0

    /* Initialise the XUD endpoints */
    XUD_ep ep_out = XUD_InitEp(c_ep_out, XUD_EPTYPE_BUL);
    XUD_ep ep_in  = XUD_InitEp(c_ep_in, XUD_EPTYPE_BUL);

    /* Mark OUT endpoint as ready to receive */
    XUD_SetReady_Out (ep_out, host_transfer_buf);

    SCPI_initialize_parser();

    while(1)
    {
        select
        {
            case XUD_GetData_Select(c_ep_out, ep_out, host_transfer_length, result):
                /* Packet from host recieved */
                if (result == XUD_RES_RST) {
                    XUD_ResetEndpoint(ep_in, null);
                }
                msg_id = host_transfer_buf[0];
```

```
                switch (msg_id) {
                    case DEV_DEP_MSG_OUT:
                    {
                        scpi_cmd_len = host_transfer_buf[4];    //TODO: Handle 4-byte TransferSize for
                          ↪ scpi_cmd_len
                        SCPI_get_cmd(&host_transfer_buf[12], scpi_cmd_len, scpi_cmd);
                    }
                    break;

                    case REQUEST_DEV_DEP_MSG_IN:
                    {
                        /* Prepare response message buffer */
                        scpi_parse_res = SCPI_parse_cmd(scpi_cmd, scpi_cmd_len, &host_transfer_buf[12],
                          ↪ host_transfer_length);
                        host_transfer_buf[4] = host_transfer_length;
                        host_transfer_buf[5] = 0x00;
                        host_transfer_buf[6] = 0x00;
                        host_transfer_buf[8] = 0x01;    //Set (EOM=1)
                        host_transfer_length += 12;     //Bulk-IN header size
                        host_transfer_buf[host_transfer_length++] = 0x00; //Set Alighnment byte

                        XUD_SetReady_In (ep_in, host_transfer_buf, host_transfer_length);
                    }
                    break;

                    case VENDOR_SPECIFIC_OUT:
                        /* Handle any vendor specific command messages */
                    break;

                    case REQUEST_VENDOR_SPECIFIC_IN:
                        /* Handle any vendor specific command messages */
                    break;

                    default:
                    break;
                }

                /* Mark EP as ready again */
                XUD_SetReady_Out (ep_out, host_transfer_buf);
            break ;

            case XUD_SetData_Select(c_ep_in, ep_in, result):
                /* Packet successfully sent to host */
                if (result == XUD_RES_RST) {
                    XUD_ResetEndpoint(ep_in, null);
                }

                /* Mark EP OUT as ready again */
                XUD_SetReady_Out (ep_out, host_transfer_buf);
            break ;
        }
    }
}
```

F.3 Source code for main.xc

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

#include "usb.h"

/* USB Endpoint Defines */
#define XUD_EP_COUNT_OUT   2    //Includes EP0 (1 out EP0 + USBTMC data output EP)
#define XUD_EP_COUNT_IN    2    //Includes EP0 (1 in EP0)

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

/* Global report buffer, global since used by Endpoint0 core */
unsigned char g_reportBuffer[] = {0, 0, 0, 0};

/* The main function runs three cores: the XUD manager, Endpoint 0, and a USBTMC endpoints. An array of
```

```
   channels is used for both IN and OUT endpoints */
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: usbtmc_bulk_endpoints(c_ep_out[1],c_ep_in[1]);

    }
    return 0;
}
```