Using the Device Control Library over USB

The Device Control library provides an API and a set of communication layers that provide a host to device control path which is agnostic of the actual transport used.

Multiple transport layers are provided as part of the library including I2C, xSCOPE over xCONNECT and USB.

This application note provides a worked example of using the USB transport layer to implement a control path that allows a host program to query and set GPIO on the device hardware.

Required tools and libraries

The code in this application note is known to work on version 14.2.1 of the xTIMEcomposer tools suite, it may work on other versions.

The application does not have any dependencies (i.e. it does not rely on any libraries).

Required hardware

The example code provided by this application has been implemented and tested to work on the xCORE Array Microphone board. It is also know to work with the XVSM-2000 Smart Microphone board.

Prerequisites

- This document assumes familiarity with the XMOS xCORE architecture, 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 a description of XMOS related terms found in this document please see the XMOS Glossary[1].

[1] http://www.xmos.com/published/glossary

1 Overview

The application uses a total of four logical cores. Two logical cores take care of implementing USB stack (one for low level transactions and one for handling Endpoint 0 control requests). One logical core is used to run the button and LED server task. The fourth logical core runs the application and communicates with the button/LED server task and EP0 from where it receives and sends and receives commands over the Device Control API from the host.

1.1 Block diagram

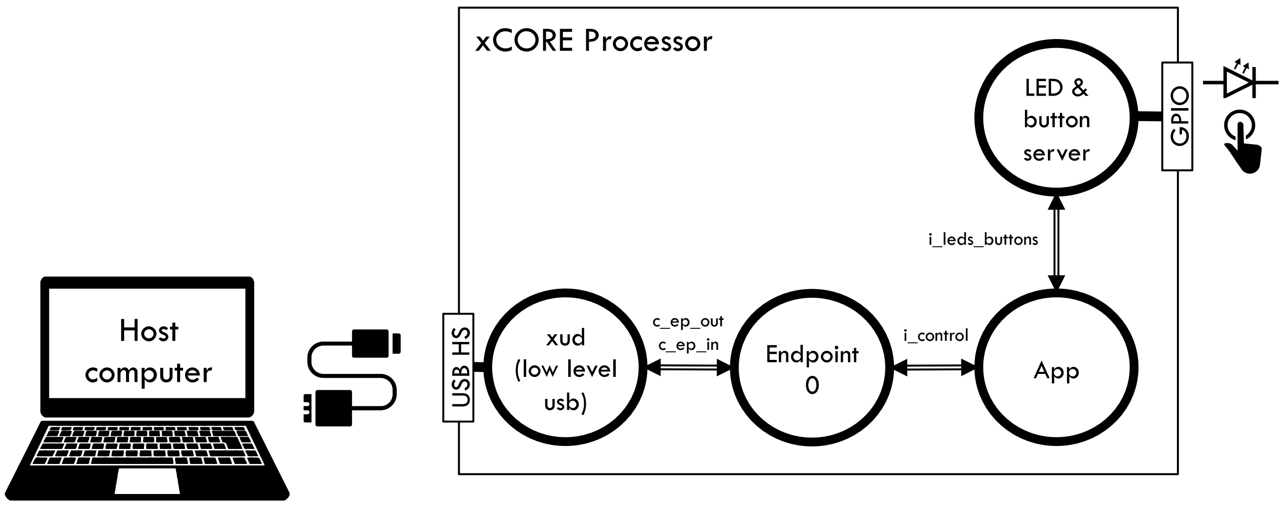

Figure 1: Application block diagram

When implementing a USB device, a device class must be specified. The Device Control protocol is device class agnostic and instead uses the vendor request type (bmRequestType) which conveys messages outside of a device class. For the purpose of this example a vendor specific device class/subclass/protocol of value 0xff/0xff/0xff is used. This allows the host to enumerate the device which is required if vendor requests are to be made.

2 How to use the Device Control library

2.1 The Makefile

To start using the device control library, you need to add lib_device_control to you Makefile:

```
USED_MODULES = .. lib_device_control ...
```

The application in this application note also uses the USB Device library (lib_usb) for access to USB and the Mic Array Board Support library (lib_mic_array_board_support) for access to the buttons and LEDs on the hardware. So the Makefile also includes:

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

2.2 Includes

This application requires the system header that defines XMOS xCORE specific defines for declaring and initializing hardware:

```
#include <platform.h>
#include <assert.h>
#include <xscope.h>
#include <stdio.h>
#include <stdint.h>
#include "usb.h"
#include "hid.h"
#include "descriptors.h"
#include "control.h"
#include "mic_array_board_support.h"
```

The Device Control library library functions and types are defined in control.h. This header must be included in your code to use the library. Low level USB device functionality is provided by usb.h and the USB device functionality is provided by the API described in hid.h and it's associated descriptor set in decriptors.h. Access to the xCORE Array Microphone board GPIO is provided by mic_array_board_support.h.

2.3 Allocating hardware resources

On an xCORE the pins are controlled by ports. The Device Control library itself requires no ports. However the application uses USB and GPIO. The USB I/O resources are implicitly defined by including the USB library whereas the buttons and LEDs ports are explicitly passed to the I/O server task. The specific resource IDs are defined in mic_array_board_support.h which themselves are defined from the platform file MIC-ARRAY-1V0.xn:

```
on tile[0]: mabs_led_ports_t p_leds = MIC_BOARD_SUPPORT_LED_PORTS;
on tile[0]: in port p_buttons =  MIC_BOARD_SUPPORT_BUTTON_PORTS;
```

2.4 Registering Controllable Resources

Before any control can take place the controlled entity must register controllable entities. This is done by populating a table of resource IDs at startup. The register_resources() method is only called once at initialization time. For this application, only one controllable resource is registered.

```
case i_control.register_resources(control_resid_t resources[MAX_RESOURCES_PER_INTERFACE],
                                  unsigned &num_resources):
  resources[0] = RESOURCE_ID;
  num_resources = 1;
  break;
```

2.5 Reading over Device Control

When the host requests a read from a controlled resource a USB control transaction is received by the device low level driver, which implemented in xud(). This causes a setup packet to be sent to endpoint 0 where it is identified as a device-to-host vendor request and is subsequently processed.

```
case USB_BMREQ_D2H_VENDOR_DEV:
  /* application retrieval latency inside the control library call
   * XUD task defers further calls by NAKing USB transactions
   */
  if (control_process_usb_get_request(sp.wIndex, sp.wValue, sp.wLength, request_data, i_control) ==
    ↪ CONTROL_SUCCESS) {
    len = sp.wLength;
    res = XUD_DoGetRequest(ep0_out, ep0_in, request_data, len, len);
```

The Device Control control_process_usb_get_request() function takes the USB parameters wIndex, wValue and wLength along with a reference to the data buffer and converts it into an method call across the interface i_control. The call to the read_command() method is subsequently made on the control interface which is then handled by the application. Once the application has filled the buffer with data and completed it's select case, the data buffer reference is returned to xud by EP0 and the USB control transaction is completed. If the application does not indicate a successful filling of the buffer then the vendor request transaction is stalled to indicate failure by not returning any data.

```
case i_control.read_command(control_resid_t resid, control_cmd_t cmd,
                            uint8_t payload[payload_len], unsigned payload_len) -> control_ret_t ret:
```

2.6 Writing over Device Control

When the host requests a write to a controlled resource a USB control transaction is received by the device low level driver, implemented in xud. This causes a setup packet to be sent to endpoint 0 where it is identified as a host-to-device vendor request and is subsequently processed.

```
case USB_BMREQ_H2D_VENDOR_DEV:
  res = XUD_GetBuffer(ep0_out, request_data, len);
  if (res == XUD_RES_OKAY) {
    if (control_process_usb_set_request(sp.wIndex, sp.wValue, sp.wLength, request_data, i_control) ==
      ↪ CONTROL_SUCCESS) {
      /* zero length data to indicate success
       * on control error, go to standard requests, which will issue STALL
       */
      res = XUD_DoSetRequestStatus(ep0_in);
```

The Device Control control_process_usb_get_request() function takes the USB parameters wIndex, wValue and wLength along with a reference to the data buffer and converts it into an method call across the interface i_control. The call to the write_command() method is subsequently made on the control interface which is then handled by server on the application side. Once the application has handled the passed data as required and completed it's select case a return code is returned and the USB transaction can either be acknowledged or the transaction stalled to indicate failure.

```
case i_control.write_command(control_resid_t resid, control_cmd_t cmd,
                             const uint8_t payload[payload_len], unsigned payload_len) -> control_ret_t ret:
```

2.7 The application main() function

The main() function sets up the tasks within the application.

Firstly, the interfaces and channels are declared. In xC, channels provide a simple way of passing data tokens between concurrent tasks, without the need to worry about route setup or low level control token protocol. XUD is written using a channel interface and so uses this method of communicating.

```
chan c_ep_out[NUM_EP_OUT], c_ep_in[NUM_EP_IN];
```

xC Interfaces also provide a means of concurrent tasks communicating with each other. Interfaces add high level language features on top of the channels and allow remote calling of methods while passing parameters and returning values, all with the benefit of type checking.

Communication between the Endpoint 0 task and the LED and Button server is performed using interfaces.

```
interface control i_control[1];
interface mabs_led_button_if i_leds_buttons[1];
```

The rest of the main() function starts all the tasks in parallel using the xC par construct:

```
par {
  on USB_TILE: par {
    endpoint0(c_ep_out[0], c_ep_in[0], i_control);
    xud(c_ep_out, NUM_EP_OUT, c_ep_in, NUM_EP_IN, null, XUD_SPEED_HS, XUD_PWR_SELF);
  }
  on tile[0]: par {
    app(i_control[0], i_leds_buttons[0]);
    mabs_button_and_led_server(i_leds_buttons, 1, p_leds, p_buttons);
  }
}
```

Note that xud() and endpoint0() are placed on USB_TILE which is tile[1] (ass defined in the Makefile) leaving tile[0] completely free for the application. The mabs_button_and_led_server() task needs to be placed on tile[0] because the GPIO connected to the LEDs and buttons reside on that tile. The Application task app() may be place on either tile.

This code starts all of the tasks concurrently and they then communicate over the channels and interfaces.

2.8 The application app() task

The application here is extremely simple. It consists of a while(1) select{} block containing three cases. These cases are outlined above and handle registration, reading and writing. It waits for calls to these cases from the client side (Endpoint 0) and handles them accordingly.

APPENDIX A - Demo Hardware Setup

To run the demo, connect a USB cable to power the xCORE Array Microphone board and plug the xTAG to the board and connect the xTAG USB cable to your development machine.

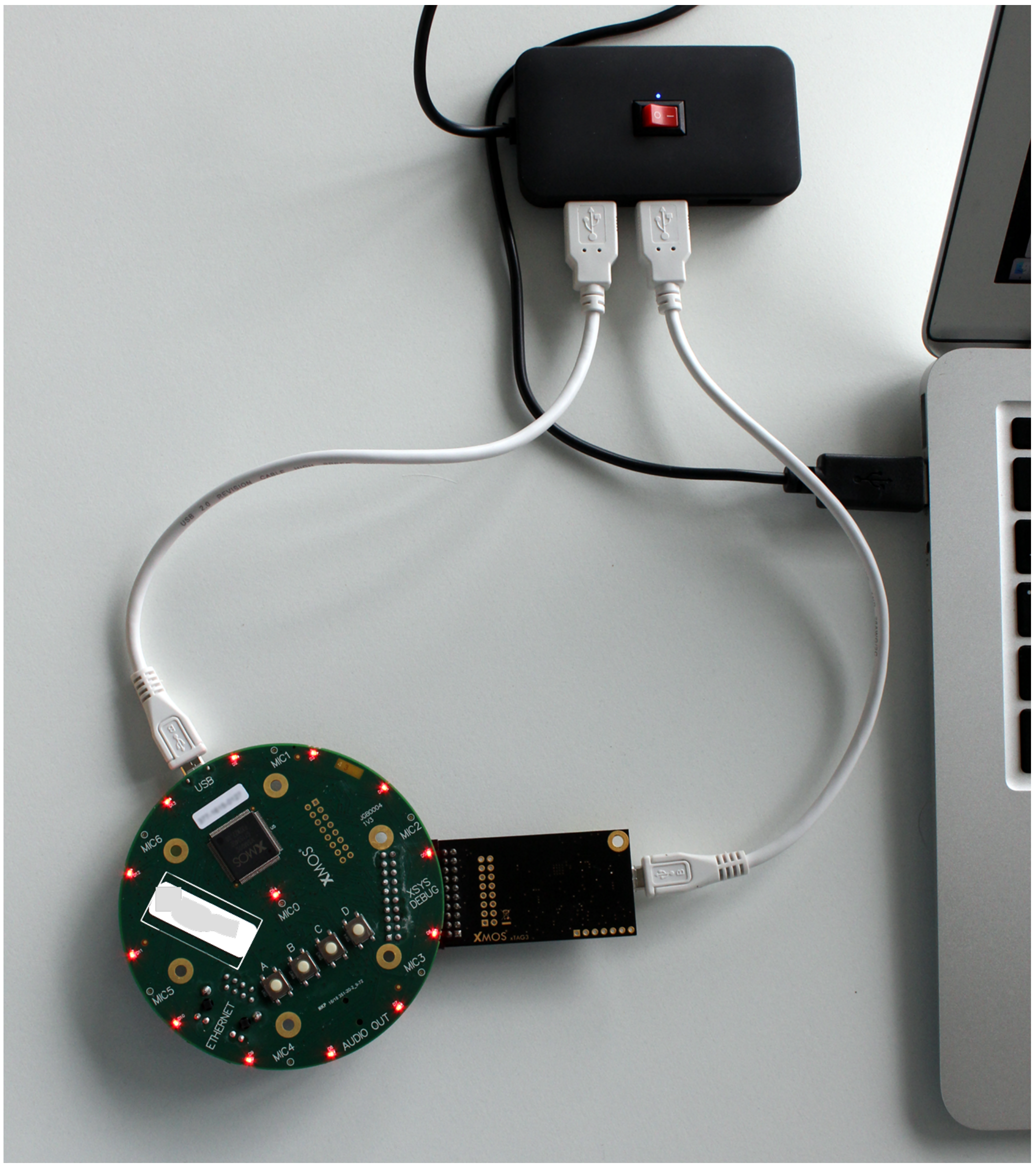

Figure 2: Hardware setup

APPENDIX B - Launching the demo application

Once the demo example has been built either from the command line using xmake from the project directory wherer the Makefile can be found or via the build mechanism of xTIMEcomposer studio it can be executed on the xCORE Array Microphone board.

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 you use the `xrun` tool to download and run the code on the xCORE device:

```
xrun --xscope bin/ANXXXXX_xxxxxx.xe
```

Once this command has executed the application will be running on the xCORE Array Microphone board.

B.2 Launching from xTIMEcomposer Studio

From xTIMEcomposer Studio use the run mechanism to download code to xCORE device. Select the xCORE binary from the `bin/` directory, right click and go to Run Configurations. Double click on xCORE application to create a new run configuration, enable the xSCOPE I/O mode in the dialog box and then select Run.

Once this command has executed the application will be running on the xCORE Array Microphone board.

B.3 Running the application

Once the application is started using either of the above methods there should be output printed to the console:

```
started
```

The device is now ready to receive and handle control requests from the host.

B.4 Building the host Application

The host application is supported on Windows and OSX platforms. To build the application on a Windows platform, it is expected that the Visual Studio compiler is installed (cl.exe). On the OSX platform, the Xcode command line tools must be installed.

To build the application, from a suitable command line shell with the compiler environment set, type either for Windows:

```
nmake -f Makefile.Win32
```

or for OSX:

```
make -f Makefile.OSX
```

The build will result in a binary being compiled into /bin/a.out.

B.5 Running the host application

To run under windows:

```
\bin\a.exe
```

To run under OSX:

```
/bin/a.out
```

The application will attempt to enumerate the device and, on success, will request the user to input a number. This number will be sent across the device control API as a write command and set an appropriate number of LEDs on the device hardware. A number between 0 and 13 is valid. For example:

```
device found
started
Enter number of LEDs to be lit: 11
```

After receiving the command, the last button event will be reported. The buttons are active low so the following report indicates that the last button event was button C being released:

```
Last button event: C value: 1
```

APPENDIX C - References

XMOS Tools User Guide

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

XMOS xCORE Programming Guide

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

XMOS Device Control Library

http://www.xmos.com/support/libraries/lib_device_control

APPENDIX D - Full source code listing

D.1 Source code for main.xc

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved
#include <platform.h>
#include <assert.h>
#include <xscope.h>
#include <stdio.h>
#include <stdint.h>
#include "usb.h"
#include "hid.h"
#include "descriptors.h"
#include "control.h"
#include "mic_array_board_support.h"
#include "app.h"

#define DEBUG_UNIT DEVICE
#include "debug_print.h"

on tile[0]: mabs_led_ports_t p_leds = MIC_BOARD_SUPPORT_LED_PORTS;
on tile[0]: in port p_buttons =  MIC_BOARD_SUPPORT_BUTTON_PORTS;

void endpoint0(chanend c_ep0_out, chanend c_ep0_in, client interface control i_control[1])
{
  USB_SetupPacket_t sp;
  XUD_Result_t res;
  XUD_BusSpeed_t bus_speed;
  XUD_ep ep0_out, ep0_in;
  unsigned char request_data[EP0_MAX_PACKET_SIZE];
  int handled;
  size_t len;

  ep0_out = XUD_InitEp(c_ep0_out, XUD_EPTYPE_CTL | XUD_STATUS_ENABLE);
  ep0_in = XUD_InitEp(c_ep0_in, XUD_EPTYPE_CTL | XUD_STATUS_ENABLE);

  control_init();
  control_register_resources(i_control, 1);

  while (1) {
    res = USB_GetSetupPacket(ep0_out, ep0_in, sp);
    handled = 0;

    if (res == XUD_RES_OKAY) {

      debug_printf("recipient %d type %d direction %d request %d value %d index %d length %d\n",
        sp.bmRequestType.Recipient, sp.bmRequestType.Type, sp.bmRequestType.Direction,
        sp.bRequest, sp.wValue, sp.wIndex, sp.wLength);

      switch ((sp.bmRequestType.Direction << 7) | (sp.bmRequestType.Type << 5) | (sp.bmRequestType.Recipient))
        ↪  {

        case USB_BMREQ_H2D_VENDOR_DEV:
          res = XUD_GetBuffer(ep0_out, request_data, len);
          if (res == XUD_RES_OKAY) {
            if (control_process_usb_set_request(sp.wIndex, sp.wValue, sp.wLength, request_data, i_control) ==
              ↪ CONTROL_SUCCESS) {
              /* zero length data to indicate success
               * on control error, go to standard requests, which will issue STALL
               */
              res = XUD_DoSetRequestStatus(ep0_in);
              handled = 1;
            }
          }
          break;

        case USB_BMREQ_D2H_VENDOR_DEV:
          /* application retrieval latency inside the control library call
           * XUD task defers further calls by NAKing USB transactions
           */
          if (control_process_usb_get_request(sp.wIndex, sp.wValue, sp.wLength, request_data, i_control) ==
            ↪ CONTROL_SUCCESS) {
            len = sp.wLength;
            res = XUD_DoGetRequest(ep0_out, ep0_in, request_data, len, len);
            handled = 1;
            /* on control error, go to standard requests, which will issue STALL */
          }
          break;
      }

      if (!handled) {
        /* if we haven't handled the request about then do standard enumeration requests */
        debug_printf("not handled, passing to standard requests\n");
        unsafe {
          res = USB_StandardRequests(ep0_out, ep0_in, devDesc,
            sizeof(devDesc), cfgDesc, sizeof(cfgDesc),
            null, 0, null, 0,
            stringDescriptors, sizeof(stringDescriptors) / sizeof(stringDescriptors[0]),
            sp, bus_speed);
        }
      }
    }

    if (res == XUD_RES_RST) {
      bus_speed = XUD_ResetEndpoint(ep0_out, ep0_in);
    }
  }
}

enum {
  EP_OUT_ZERO,
  NUM_EP_OUT
};

enum {
  EP_IN_ZERO,
  NUM_EP_IN
};

int main(void)
{
  chan c_ep_out[NUM_EP_OUT], c_ep_in[NUM_EP_IN];
  interface control i_control[1];
  interface mabs_led_button_if i_leds_buttons[1];
  par {
    on USB_TILE: par {
      endpoint0(c_ep_out[0], c_ep_in[0], i_control);
      xud(c_ep_out, NUM_EP_OUT, c_ep_in, NUM_EP_IN, null, XUD_SPEED_HS, XUD_PWR_SELF);
    }
    on tile[0]: par {
      app(i_control[0], i_leds_buttons[0]);
      mabs_button_and_led_server(i_leds_buttons, 1, p_leds, p_buttons);
    }
  }
  return 0;
}
```

D.2 Source code for app.xc

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved
#include <stdio.h>
#include <stdint.h>
#include <assert.h>
#include "control.h"
#include "mic_array_board_support.h"
#include "app.h"

void app(server interface control i_control, client interface mabs_led_button_if i_leds_buttons)
{
  unsigned num_commands;
  int i;

  printf("started\n");
#ifdef ERRONEOUS_DEVICE
  printf("generate errors\n");
#endif
```

```
  num_commands = 0;

  while (1) {
    select {
      case i_control.register_resources(control_resid_t resources[MAX_RESOURCES_PER_INTERFACE],
                                        unsigned &num_resources):
        resources[0] = RESOURCE_ID;
        num_resources = 1;
        break;

      case i_control.write_command(control_resid_t resid, control_cmd_t cmd,
                                   const uint8_t payload[payload_len], unsigned payload_len) -> control_ret_t
                                     ↪ ret:
        num_commands++;
#ifdef ERRONEOUS_DEVICE
        if ((num_commands % 3) == 0)
          resid += 1;
#endif
        printf("%u: W %d %d %d,", num_commands, resid, cmd, payload_len);
        for (i = 0; i < payload_len; i++) {
          printf(" %02x", payload[i]);
        }
        printf("\n");
        if (resid != RESOURCE_ID) {
          printf("unrecognised resource ID %d\n", resid);
          ret = CONTROL_ERROR;
          break;
        }
        for (i = 0; i < MIC_BOARD_SUPPORT_LED_COUNT; i++){
          if (i < payload[0]) i_leds_buttons.set_led_brightness(i, 255);
          else i_leds_buttons.set_led_brightness(i, 0);
        }
        ret = CONTROL_SUCCESS;
        break;

      case i_control.read_command(control_resid_t resid, control_cmd_t cmd,
                                  uint8_t payload[payload_len], unsigned payload_len) -> control_ret_t ret:
        num_commands++;
#ifdef ERRONEOUS_DEVICE
        if ((num_commands % 3) == 0)
          resid += 1;
#endif
        printf("%u: R %d %d %d\n", num_commands, resid, cmd, payload_len);
        if (resid != RESOURCE_ID) {
          printf("unrecognised resource ID %d\n", resid);
          ret = CONTROL_ERROR;
          break;
        }
        if (payload_len != 4) {
          printf("expecting 4 read bytes, not %d\n", payload_len);
          ret = CONTROL_ERROR;
          break;
        }
        unsigned button;
        mabs_button_state_t button_state;
        i_leds_buttons.get_button_event(button, button_state);
        printf("button, button_state= %d, %d\n", button, button_state);
        payload[0] = button;
        payload[1] = button_state;
        payload[2] = 0x56;
        payload[3] = 0x78;
        ret = CONTROL_SUCCESS;
        break;
    }
  }
}
```