XMOS USB Device Design Guide

Document Number: XM003113A

Publication Date: 2014/7/15

XMOS®

SYNOPSIS

This document is a design guide for creating both Full and High Speed USB 2.0 devices on XMOS devices.

Table of Contents

1 Overview

IN THIS CHAPTER

This document describes using the XMOS USB Device (XUD) Library and provides a worked example of a USB Human Interface Device (HID) Class compliant mouse using the library.

This library is aimed primarily for use with xCORE-USB (U-Series) devices but it does also support L-Series devices (see §7).

For full XUD API listing and documentation please see the document *XMOS USB Device (XUD) Library*.

This document assumes familiarity with the XMOS xCORE architecture, the Universal Serial Bus 2.0 Specification (and related specifications), the XMOS tool chain and XC language.

1.1 Features

- Support for USB 2.0 full and high speed devices.

1.2 Memory requirements

The approximate memory usage for the USB device module including the XUD library is as follows:

	Usage
Stack	2kB
Program	12kB

1.3 Resource requirements

The resources used by the device application and libraries on the xCORE-USB are shown below:

Resource	Requirements
Logical Cores	2 plus 1 per endpoint
Channels	2 for Endpoint0 and 1 additional per IN and OUT endpoint
Timers	4 timers
Clock blocks	Clock blocks 4 and 5

1.4 Core speed

Due to I/O timing requirements, the library requires a guaranteed MIPS rate to ensure correct operation. This means that core count restrictions must be observed. The XUD core must run at at least 80 MIPS.

This means that for an xCORE device running at 500MHz no more than six cores shall execute at any one time when using the XUD.

This restriction is only a requirement on the tile on which the XUD is running. For example, a different tile on an U16 device is unaffected by this restriction.

1.5 Ports and pins

The U-Series of processors has an integrated USB transceiver. Some ports are used to communicate with the USB transceiver inside the U-Series packages. These ports/pins should not be used when USB functionality is enabled. The ports/pins are shown in Figure 1.

Pin	Port				
	1b	4b	8b	16b	32b
X0D02		P4A0	P8A0	P16A0	P32A20
X0D03		P4A1	P8A1	P16A1	P32A21
X0D04		P4B0	P8A2	P16A2	P32A22
X0D05		P4B1	P8A3	P16A3	P32A23
X0D06		P4B2	P8A4	P16A4	P32A24
X0D07		P4B3	P8A5	P16A5	P32A25
X0D08		P4A2	P8A6	P16A6	P32A26
X0D09		P4A3	P8A7	P16A7	P32A27
X0D23	P1H0				
X0D25	P1J0				
X0D26		P4E0	P8C0	P16B0	
X0D27		P4E1	P8C1	P16B1	
X0D28		P4F0	P8C2	P16B2	
X0D29		P4F1	P8C3	P16B3	
X0D30		P4F2	P8C4	P16B4	
X0D31		P4F3	P8C5	P16B5	
X0D32		P4E2	P8C6	P16B6	
X0D33		P4E3	P8C7	P16B7	
X0D34	P1K0				
X0D36	P1M0		P8D0	P16B8	
X0D37	P1N0		P8C1	P16B1	
X0D38	P1O0		P8C2	P16B2	
X0D39	P1P0		P8C3	P16B3	

Figure 1: U-Series required pin/port connections

2 Hardware requirements

IN THIS CHAPTER

The XMOS USB Device Library supports both the xCORE-USB (U-Series) devices and the xCORE General Purpose (L-Series) devices. However, not all development kits support implementing USB devices.

2.1 Recommended hardware

2.1.1 U16 sliceKIT

The USB device capabilities are best evaluated using the U16 Slicekit Modular Development Platform. The required boards are:

- `XP-SKC-U16` (Slicekit U16 Core Board) plus `XA-SK-USB-AB` (USB Slice)
- Optionally: `XA-SK-MIXED SIGNAL` (Mixed Signal Slice) for the HID Class USB Device Demo

2.2 Demonstration applications

2.2.1 HID class USB device demo

This application demonstrates how to write a Human Interface Device (HID) Class Device; a mouse.

- Package: HID Class USB Device Demo
- Application: app_hid_mouse_demo

3 System Overview

IN THIS CHAPTER

The XMOS USB device support is divided into the XMOS USB Device (XUD) Library (module_xud) and the USB Device Helper Functions (module_usb_device)

3.1 XUD library

For full XUD API listing and documentation please see the document *XMOS USB Device (XUD) Library*

The XUD Library performs all the low-level I/O operations required to meet the USB 2.0 specification. This processing goes up to and includes the transaction level. It removes all low-level timincg requirements from the application, allowing quick development of all manner of USB devices.

The XUD Library allows the implementation of both full-speed and high-speed USB 2.0 devices on U-Series and L-Series devices.

The U-Series includes an integrated USB transceiver. For the L-Series the implementation requires the use of an external ULPI transceiver such as the SMSC USB33XX range. Two libraries, with identical interfaces, are provided - one for U-Series and one for L-Series devices.

The XUD Library runs in a single core with endpoint and application cores communicating with it via a combination of channel communication and shared memory variables.

There is one channel per IN or OUT endpoint. Endpoint 0 (the control endpoint) requires two channels, one for each direction. Note, that throughout this document the USB nomenclature is used: an OUT endpoint is used to transfer data from the host to the device, an IN endpoint is used when the host requests data from the device.

An example task diagram is shown in Figure 2. Circles represent cores running with arrows depicting communication channels between these cores. In this configuration there is one core that deals with endpoint 0, which has both the input and output channel for endpoint 0. IN endpoint 1 is dealt with by a second core, and OUT endpoint 2 and IN endpoint 5 are dealt with by a third core. Cores must be ready to communicate with the XUD Library whenever the host demands its attention. If not, the XUD Library will NAK.

It is important to note that, for performance reasons, cores communicate with the XUD Library using both XC channels and shared memory communication. Therefore, *all cores using the XUD Library must be on the same tile as the library itself*.

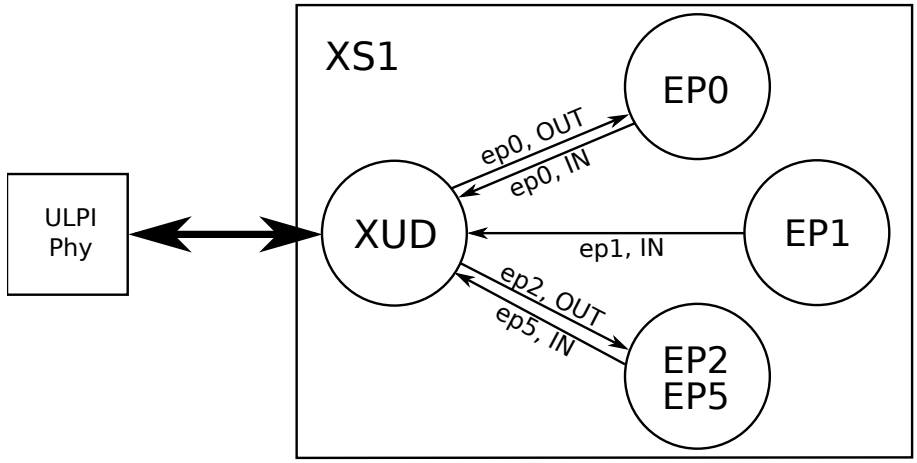

Figure 2: XUD Overview

3.1.1 XUD core

The main XUD task is `XUD_Manager()` (see X) that performs power-signalling/handshaking on the USB bus, and passes packets on for the various endpoints.

This function should be called directly from the top-level `par` statement in `main()` to ensure that the XUD Library is ready within the 100ms allowed by the USB specification.

3.1.2 Endpoint communication with `XUD_Manager()`

Communication state between a core and the XUD Library is encapsulated in an opaque type `XUD_ep` (see X).

All client calls communicating with the XUD Library pass in this type. These data structures can be created at the start of execution of a client core with using `XUD_InitEp()` that takes as an argument the endpoint channel connected to the XUD Library.

Endpoint data is sent/received using `XUD_SetBuffer()` (see X) and receive data using `XUD_GetBuffer()` (see X).

These functions will automatically deal with any low-level complications required such as Packet ID toggling etc.

3.1.3 Endpoint type table

The endpoint type table should take an array of `XUD_EpType` to inform XUD about endpoints being used. This is mainly used to indicate the transfer-type of each endpoint (bulk, control, isochronous or interrupt) as well as whether the endpoint wishes to be informed about bus-resets (see §3.1.4).

3.1.4 Status reporting

Status reporting on an endpoint can be enabled so that bus state is known. This is achieved by ORing `XUD_STATUS_ENABLE` into the relevant endpoint in the endpoint type table.

This means that endpoints are notified of USB bus resets (and bus-speed changes). The XUD access functions discussed previously (`XUD_SetBuffer()`, `XUD_GetBuffer()`) return XUD_RES_RST if a USB bus reset is detected.

After a reset notification has been received, the endpoint must call the `XUD_ResetEndpoint()` function. This will return the current bus speed.

See *XMOS USB Device (XUD) Library* for full details.

3.2 USB device helper functions

The USB Device Helper Functions provide a set of standard functions to aid the creation of USB devices. USB devices must provide an implementation of endpoint 0 and can optionally provide a number of other IN and OUT endpoints.

3.2.1 Standard requests and endpoint 0

Endpoint 0 must deal with enumeration and configuration requests from the host. Many enumeration requests are compulsory and common to all devices, most of them being requests for mandatory descriptors (Configuration, Device, String, etc.). Since these requests are common across most (if not all) devices, some useful functions are provided to deal with them.

Firstly, the function `USB_GetSetupPacket()` is provided. This makes a call to the standard XUD function `XUD_GetSetupBuffer()` with the 8 byte Setup packet which it parses into a `USB_SetupPacket_t` structure (see §4.1.1) for further inspection. The `USB_SetupPacket_t` structure passed by reference to `USB_GetSetupPacket()` is populated by the function.

At this point the request is in a reasonable state to be parsed by endpoint 0. Please see Universal Serial Bus 2.0 specification for full details of setup packet and request structure.

A `USB_StandardRequests()` (see §4.2.2) function provides a bare-minimum implementation of the mandatory requests required to be implemented by a USB device. It is not intended that this replace a good knowledge of the requests required, since the implementation does not guarantee a fully USB compliant device. Each request could well be required to be over-ridden for a device implementation. For example,

a USB Audio device could well require a specialised version of SET_INTERFACE since this could mean that audio will be streamed imminently.

Please see Universal Serial Bus 2.0 spec for full details of these requests.

The function inspects this USB_SetupPacket_t structure and includes a minimum implementation of the Standard Device requests. To see the requests handled and a listing of the basic functionality associated with the request see §4.2.3.

4 API

IN THIS CHAPTER

The XMOS USB Device library is provided by `module_xud` and the USB Device Helper Functions are provided by `module_usb_device`.

The APIs of `module_xud` is listed in *XMOS USB Device (XUD) Library*. The API of `module_usb_device` is detailed in this section.

Please note, both `module_xud` and `module_usb_device` depend on the module `module_usb_shared`

4.1 module_usb_shared

4.1.1 USB_SetupPacket_t

This structure closely matches the structure defined in the USB 2.0 Specification:

```
typedef struct USB_SetupPacket
{
    USB_BmRequestType_t bmRequestType;    /* (1 byte) Specifies direction of dataflow,
                                             type of rquest and recipient */
    unsigned char bRequest;               /* Specifies the request */
    unsigned short wValue;                /* Host can use this to pass info to the
                                             device in its own way */
    unsigned short wIndex;                /* Typically used to pass index/offset such
                                             as interface or EP no */
    unsigned short wLength;               /* Number of data bytes in the data stage
                                             (for Host -> Device this this is exact
                                             count, for Dev->Host is a max. */
} USB_SetupPacket_t;
```

4.2 module_usb_device

4.2.1 USB_GetSetupPacket()

USB_GetSetupPacket()

Receives a Setup data packet and parses it into the passed USB_SetupPacket_t structure.

Type

```
XUD_Result_t USB_GetSetupPacket(XUD_ep ep_out,
                                XUD_ep ep_in,
                                USB_SetupPacket_t &sp)
```

Parameters

ep_out	OUT endpint from XUD
ep_in	IN endpoint to XUD
sp	SetupPacket structure to be filled in (passed by ref)

Returns

Returns XUD_RES_OKAY on success, XUD_RES_RST on bus reset

4.2.2 USB_StandardRequests()

This function takes a populated USB_SetupPacket_t structure as an argument.

USB_StandardRequests()

This function deals with common requests This includes Standard Device Requests listed in table 9-3 of the USB 2.0 Spec all devices must respond to these requests, in some cases a bare minimum implementation is provided and should be extended in the devices EP0 code It handles the following standard requests appropriately using values passed to it:.

Get Device Descriptor (using devDesc_hs/devDesc_fs arguments)

Get Configuration Descriptor (using cfgDesc_hs/cfgDesc_fs arguments)

String requests (using strDesc argument)

Get Device_Qualifier Descriptor

Get Other-Speed Configuration Descriptor

Set/Clear Feature (Endpoint Halt)

Get/Set Interface

Set Configuration

If the request is not recognised the endpoint is marked STALLED

Type

```
XUD_Result_t USB_StandardRequests(XUD_ep ep_out,
                                  XUD_ep ep_in,
                                  ?_ARRAY_OF(unsigned char,
                                  devDesc_hs),
                                  int devDescLength_hs,
                                  ?_ARRAY_OF(unsigned char,
                                  cfgDesc_hs),
                                  int cfgDescLength_hs,
                                  ?_ARRAY_OF(unsigned char,
                                  devDesc_fs),
                                  int devDescLength_fs,
                                  ?_ARRAY_OF(unsigned char,
                                  cfgDesc_fs),
                                  int cfgDescLength_fs,
                                  char *unsafe strDescs[],
                                  int strDescsLength,
                                  USB_SetupPacket_t &sp,
                                  XUD_BusSpeed_t usbBusSpeed)
```

Parameters

`ep_out` Endpoint from XUD (ep 0)

`ep_in` Endpoint from XUD (ep 0)

`devDesc_hs` The Device descriptor to use, encoded according to the USB standard

`devDescLength_hs` Length of device descriptor in bytes

`cfgDesc_hs` Configuration descriptor

`cfgDescLength_hs` Length of config descriptor in bytes

`devDesc_fs` The Device descriptor to use, encoded according to the USB

4.2.3 Standard Device Request Types

`USB_StandardRequests()` handles the following Standard Device Requests:

- `SET_ADDRESS`
  - The device address is set in XUD (using `XUD_SetDevAddr()`).
- `SET_CONFIGURATION`
  - A global variable is updated with the given configuration value.
- `GET_STATUS`
  - The status of the device is returned. This uses the device Configuration descriptor to return if the device is bus powered or not.
- `SET_CONFIGURATION`
  - A global variable is returned with the current configuration last set by `SET_CONFIGURATION`.
- `GET_DESCRIPTOR`
  - Returns the relevant descriptors. See §6.4 for further details. Note, some changes of returned descriptor will occur based on the current bus speed the device is running.
    - `DEVICE`
    - `CONFIGURATION`
    - `DEVICE_QUALIFIER`
    - `OTHER_SPEED_CONFIGURATION`
    - `STRING`

In addition the following test mode requests are dealt with (with the correct test mode set in XUD):

- `SET_FEATURE`
  - `TEST_J`
  - `TEST_K`
  - `TEST_SE0_NAK`
  - `TEST_PACKET`
  - `FORCE_ENABLE`

4.2.4 Standard Interface Requests

`USB_StandardRequests()` handles the following Standard Interface Requests:

- `SET_INTERFACE`
  - A global variable is maintained for each interface. This is updated by a `SET_INTERFACE`. Some basic range checking is included using the value `numInterfaces` from the ConfigurationDescriptor.
- `GET_INTERFACE`
  - Returns the value written by `SET_INTERFACE`.

4.2.5 Standard Endpoint Requests

`USB_StandardRequests()` handles the following Standard Endpoint Requests:

- `SET_FEATURE`
- `CLEAR_FEATURE`
- `GET_STATUS`

If parsing the request does not result in a match, the request is not handled, the Endpoint is marked "Halted" (Using `XUD_SetStall_Out()` and `XUD_SetStall_In()`) and the function returns XUD_RES_ERR.

The function returns XUD_RES_OKAY if a request was handled without error (See also `Status Reporting`).

5 Programming guide

IN THIS CHAPTER

This section provides information on how to create an application using the USB Device library.

5.1 Includes

The application needs to include `xud.h`.

5.2 Declarations

Create a table of endpoint types for both IN and OUT endpoints. These must each include one for endpoint 0.

```
#define XUD_EP_COUNT_OUT  1
#define XUD_EP_COUNT_IN   2

/* Endpoint type tables */
XUD_EpType epTypeTableOut[XUD_EP_COUNT_OUT] = {
    XUD_EPTYPE_CTL | XUD_STATUS_ENABLE
};
XUD_EpType epTypeTableIn[XUD_EP_COUNT_IN] = {
    XUD_EPTYPE_CTL | XUD_STATUS_ENABLE, XUD_EPTYPE_BUL
};
```

The endpoint types are:

- `XUD_EPTYPE_ISO`: Isochronous endpoint
- `XUD_EPTYPE_INT`: Interrupt endpoint
- `XUD_EPTYPE_BUL`: Bulk endpoint

- `XUD_EPTYPE_CTL`: Control endpoint
- `XUD_EPTYPE_DIS`: Disabled endpoint

And `XUD_STATUS_ENABLE` is ORed in to the endpoints that wish to be informed of USB bus resets (see §3.1.4).

5.3 `main()`

Within the `main()` function it is necessary to allocate the channels to connect the endpoints and then create the top-level `par` containing the XUD_Manager, endpoint 0 and any application specific endpoints.

```
int main()
{
    chan c_ep_out[XUD_EP_COUNT_OUT], c_ep_in[XUD_EP_COUNT_IN];
    par {
        XUD_Manager(c_ep_out, XUD_EP_COUNT_OUT,
                    c_ep_in, XUD_EP_COUNT_IN,
                    null, epTypeTableOut, epTypeTableIn,
                    null, null, null, XUD_SPEED_HS, null);
        Endpoint0(c_ep_out[0], c_ep_in[0]);

        // Application specific endpoints
        ...
    }
    return 0;
}
```

The XUD_Manager connects to one end of every channel while the other end is passed to an endpoint (either endpoint 0 or an application specific endpoint). Application specific endpoints are connected using channel ends so the IN and OUT channel arrays need to be extended for each endpoint.

5.4 Endpoint addresses

Endpoint 0 uses index 0 of both the endpoint type table and the channel array. The address of other endpoints must also correspond to their index in the endpoint table and the channel array.

5.5 Sending and receiving data

An application specific endpoint can send data using `XUD_SetBuffer()` and receive data using `XUD_GetBuffer()`.

5.6 Endpoint 0 implementation

It is necessary to create an implementation for endpoint 0 which takes two channels, one for IN and one for OUT. It can take an optional channel for test (see the `Test Modes` section of `XMOS USB Device (XUD) Library`).

```
void Endpoint0(chanend chan_ep0_out, chanend chan_ep0_in, chanend ?
  ↪ c_usb_test)
{
```

Every endpoint must be initialized using the `XUD_InitEp()` function. For endpoint 0 this is looks like:

```
XUD_ep ep0_out = XUD_InitEp(chan_ep0_out);
XUD_ep ep0_in  = XUD_InitEp(chan_ep0_in);
```

Typically the minimal code for endpoint 0 loops making call to `USB_GetSetupPacket()`, parses the `USB_SetupPacket_t` for any class/applicaton specific requests. Then makes a call to `USB_StandardRequests()`. And finally, calls `XUD_ResetEndpoint()` if there have been a bus-reset. For example:

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

    if(result == XUD_RES_OKAY)
    {
        switch(sp.bmRequestType.Type)
        {
            case BM_REQTYPE_TYPE_CLASS:
                switch(sp.bmRequestType.Receipient)
                {
                    case BM_REQTYPE_RECIP_INTER:
                        // Optional class specific requests.
                        break;

                    ...
                }

                break;

            ...
        }

        result = USB_StandardRequests(ep0_out, ep0_in,
                devDesc, devDescLen, ...);
    }

    if(result ==  XUD_RES_RST)
        usbBusSpeed = XUD_ResetEndpoint(ep0_out, ep0_in);
}
```

The code above could also over-ride any of the requests handled in `USB_StandardRequests()` for custom functionality.

Note, class specific code should be inserted before `USB_StandardRequests()` is called since if `USB_StandardRequests()` cannot handle a request it marks the Endpoint stalled to indicate to the host that the request is not supported by the device.

`USB_StandardRequests()` takes char array parameters for device descriptors for both high and full-speed. Note, if `null` is passed as the full-speed descriptor the high-speed descriptor is used in full-speed mode and vice versa.

Note that on reset the `XUD_ResetEndpoint()` function returns the negotiated USB speed (i.e. full or high speed).

5.7 Device descriptors

USB device descriptors must be provided for each USB device. They are used to identify the USB device's vendor ID, product ID and detail all the attributes of the advice as specified in the USB 2.0 standard. It is beyond the scope of this document to give details of writing a descriptor, please see the relavant USB Specification Documents.

5.8 Worked example

For more details see the worked HID Class example (§6).

6 Example application

IN THIS CHAPTER

This section contains a full worked example of a High Speed USB 2.0 HID Class device. The example code in this document is intended for xCORE-USB (U-Series) devices. The code would be very similar for an xCORE General Purpose (L-Series) devices with external ULPI transceiver, with only the declarations and call to `XUD_Manager()` being different.

The full source for this demo is released as the `HID Class USB Device Demo` available through the XMOS xTIMEcomposer tool. The tool can be downloaded from `www.xmos.com`.

6.1 Declarations

```
#include <xs1.h>

#include "xud.h"

#define XUD_EP_COUNT_OUT  1
#define XUD_EP_COUNT_IN   2

/* Endpoint type tables */
XUD_EpType epTypeTableOut[XUD_EP_COUNT_OUT] = {
    XUD_EPTYPE_CTL | XUD_STATUS_ENABLE
};
XUD_EpType epTypeTableIn[XUD_EP_COUNT_IN] = {
    XUD_EPTYPE_CTL | XUD_STATUS_ENABLE, XUD_EPTYPE_BUL
};
```

6.2 Main program

The main function creates three tasks: the XUD manager, endpoint 0, and HID process. An array of channels is used for both in and out endpoints, endpoint 0 requires both, the HID process is simply an IN endpoint sending mouse data to the host.

```
int main()
{
    chan c_ep_out[XUD_EP_COUNT_OUT], c_ep_in[XUD_EP_COUNT_IN];
    par {
        XUD_Manager(c_ep_out, XUD_EP_COUNT_OUT,
                    c_ep_in, XUD_EP_COUNT_IN,
                    null, epTypeTableOut, epTypeTableIn,
                    null, null, null, XUD_SPEED_HS, null);
        Endpoint0(c_ep_out[0], c_ep_in[0]);
        hid_mouse(c_ep_in[1]);
    }
    return 0;
}
```

Since we do not require SOF notifications `null` is passed into the `c_sof` parameter. `XUD_SPEED_HS` is passed for the `desiredSpeed` parameter as we wish to run as a high-speed device. Test mode support is not important for this example to `null` is also passed to the `c_usb_testmode` parameter.

6.3 HID response function

This function responds to the HID requests—it draws a square using the mouse moving 40 pixels in each direction in sequence every 100 requests. Change this function to feed other data back (for example based on user input). It demonstrates the use of `XUD_SetBuffer`.

```
void hid_mouse(chanend c_ep1) {
    char buffer[] = {0, 0, 0, 0};
    int counter = 0;
    int state = 0;

    XUD_ep ep = XUD_Init_Ep(c_ep1);

    counter = 0;
    while(1) {
        counter++;
        if(counter == 100) {
            if(state == 0) {
                buffer[1] = 40;
                buffer[2] = 0;
                state+=1;
            } else if(state == 1) {
                buffer[1] = 0;
                buffer[2] = 40;
                state+=1;
            } else if(state == 2) {
                buffer[1] = -40;
                buffer[2] = 0;
                state+=1;
            } else if(state == 3) {
                buffer[1] = 0;
                buffer[2] = -40;
                state = 0;
            }
            counter = 0;
        } else {
            buffer[1] = 0;
            buffer[2] = 0;
        }

        XUD_SetBuffer(c_ep, buffer, 4) < 0;
    }
}
```

Note, this endpoint does not receive or check for status data. It always performs IN transactions. Since it's behaviour is not modified based on bus speed the mouse cursor will move more slowly when connected via a full-speed port. Ideally the device would either modify its required polling rate in its descriptors (`bInterval` in the endpoint descriptor) or the counter value it is using in the `hid_mouse()` function.

Should processing take longer that the host IN polls, the `XUD_Manager` core will simply NAK the host. The `XUD_SetBuffer()` function will return when the packet transmission is complete.

6.4 Device Descriptors

The `USB_StandardRequests()` function expects descriptors be declared as arrays of characters. Descriptors are looked at in depth in this section.

6.4.1 Device descriptor

The device descriptor contains basic information about the device. This descriptor is the first descriptor the host reads during its enumeration process and it includes information that enables the host to further interrogate the device. The descriptor includes information on the descriptor itself, the device (USB version, vendor ID etc.), its configurations and any classes the device implements.

For the HID Mouse example this descriptor looks like the following:

```
static unsigned char devDesc[] =
{
    0x12,                   /* 0  bLength */
    USB_DESCTYPE_DEVICE,    /* 1  bdescriptorType */
    0x00,                   /* 2  bcdUSB */
    0x02,                   /* 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 */
    0x00,                   /* 16 iSerialNumber */
    0x01                    /* 17 bNumConfigurations */
```

6.4.2 Device qualifier descriptor

Devices which support both full and high-speeds must implement a device qualifier descriptor. The device qualifier descriptor defines how fields of a high speed device's descriptor would look if that device is run at a different speed. If a high-speed device is running currently at full/high speed, fields of this descriptor reflect how device descriptor fields would look if speed was changed to high/full. Please refer to section 9.6.2 of the USB 2.0 specification for further details.

For a full-speed only device this is not required.

Typically a device qualifier descriptor is derived mechanically from the device descriptor. The `USB_StandardRequest` function will build a device qualifier from the device descriptors passed to it based on the speed the device is currently running at.

6.4.3 Configuration descriptor

The configuration descriptor contains the devices features and abilities. This descriptor includes Interface and Endpoint Descriptors. Every device must have at least one configuration, in our example there is only one configuration. The configuration descriptor is presented below:

```
static unsigned char cfgDesc[] = {
    0x09,                   /* 0  bLength */
    0x02,                   /* 1  bDescriptortype */
    0x22, 0x00,             /* 2  wTotalLength */
    0x01,                   /* 4  bNumInterfaces */
    0x01,                   /* 5  bConfigurationValue */
    0x03,                   /* 6  iConfiguration */
    0x80,                   /* 7  bmAttributes */
    0xC8,                   /* 8  bMaxPower */

    0x09,                   /* 0  bLength */
    0x04,                   /* 1  bDescriptorType */
    0x00,                   /* 2  bInterfacecNumber */
    0x00,                   /* 3  bAlternateSetting */
    0x01,                   /* 4: bNumEndpoints */
    0x03,                   /* 5: bInterfaceClass */
    0x00,                   /* 6: bInterfaceSubClass */
    0x02,                   /* 7: bInterfaceProtocol*/
    0x00,                   /* 8  iInterface */

    0x09,                   /* 0  bLength. Note this is currently
                               replicated in hidDescriptor[] below */
    0x21,                   /* 1  bDescriptorType (HID) */
    0x10,                   /* 2  bcdHID */
    0x11,                   /* 3  bcdHID */
    0x00,                   /* 4  bCountryCode */
    0x01,                   /* 5  bNumDescriptors */
    0x22,                   /* 6  bDescriptorType[0] (Report) */
    0x48,                   /* 7  wDescriptorLength */
    0x00,                   /* 8  wDescriptorLength */

    0x07,                   /* 0  bLength */
    0x05,                   /* 1  bDescriptorType */
    0x81,                   /* 2  bEndpointAddress */
    0x03,                   /* 3  bmAttributes */
    0x40,                   /* 4  wMaxPacketSize */
    0x00,                   /* 5  wMaxPacketSize */
    0x01                    /* 6  bInterval */
```

6.4.4 Other speed configuration descriptor

A other speed configuration for similar reasons as the device qualifier descriptor. The `USB_StandardRequests()` function generates this descriptor from the Configuration Descriptors passed to it based on the bus speed it is currently running at. For the HID mouse example we used the same configuration Descriptors if running on full-speed or high-speed.

6.4.5 String descriptors

An array of strings supplies all the strings that are referenced from the descriptors (using fields such as 'iInterface', 'iProduct' etc.). Note that String 0 is always language ID descriptor.

The `USB_StandardRequests()` function deals with requests for strings using the table of strings passed to it. The string table for the HID mouse example is shown below:

```
static char * unsafe stringDescriptors[]=
{
    "\x09\x04",             // Language ID string (US English)
    "XMOS",                 // iManufacturer
    "Example HID Mouse",    // iProduct
    "Config",               // iConfiguration
```

Note that the `null` values and length `0` is passed for the full-speed descriptors, this means that the same descriptors will be used whether the device is running in full or high-speed.

6.5 Application and class specific requests

Although the `USB_StandardRequests()` function deals with many of the requests the device is required to handle in order to be properly enumerated by a host, typically a USB device will have Class (or Application) specific requests that must be handled.

In the case of the HID mouse there are three mandatory requests that must be handled:

- `GET_DESCRIPTOR`
  - `HID` Return the HID descriptor
  - `REPORT` Return the HID report descriptor
- `GET_REPORT` Return the HID report data

Please refer to the HID Specification and related documentation for full details of all HID requests.

The HID report descriptor informs the hosts of the contents of the HID reports that it will be sending to the host periodically. For a mouse this could include X/Y axis values, button presses etc. Tools for building these descriptors are available for download on the usb.org website.

The HID report descriptor for the HID mouse example is shown below:

```
static unsigned char hidReportDescriptor[] =
{
    0x05, 0x01,          // Usage page (desktop)
    0x09, 0x02,          // Usage (mouse)
    0xA1, 0x01,          // Collection (app)
    0x05, 0x09,          // Usage page (buttons)
    0x19, 0x01,
    0x29, 0x03,
    0x15, 0x00,          // Logical min (0)
    0x25, 0x01,          // Logical max (1)
    0x95, 0x03,          // Report count (3)
    0x75, 0x01,          // Report size (1)
    0x81, 0x02,          // Input (Data, Absolute)
    0x95, 0x01,          // Report count (1)
    0x75, 0x05,          // Report size (5)
    0x81, 0x03,          // Input (Absolute, Constant)
    0x05, 0x01,          // Usage page (desktop)
    0x09, 0x01,          // Usage (pointer)
    0xA1, 0x00,          // Collection (phys)
    0x09, 0x30,          // Usage (x)
    0x09, 0x31,          // Usage (y)
    0x15, 0x81,          // Logical min (-127)
    0x25, 0x7F,          // Logical max (127)
    0x75, 0x08,          // Report size (8)
    0x95, 0x02,          // Report count (2)
    0x81, POSITION_TYPE, // Input (Data, Rel=0x6, Abs=0x2)
    0xC0,                // End collection
    0x09, 0x38,          // Usage (Wheel)
    0x95, 0x01,          // Report count (1)
    0x81, 0x02,          // Input (Data, Relative)
    0x09, 0x3C,          // Usage (Motion Wakeup)
    0x15, 0x00,          // Logical min (0)
    0x25, 0x01,          // Logical max (1)
    0x75, 0x01,          // Report size (1)
    0x95, 0x01,          // Report count (1)
    0xB1, 0x22,          // Feature (No preferred, Variable)
    0x95, 0x07,          // Report count (7)
    0xB1, 0x01,          // Feature (Constant)
    0xC0                 // End collection
```

The request for this descriptor (and the other required requests) should be implemented before making the call to `USB_StandardRequests()`. The programmer may decide not to make a call to `USB_StandardRequests` if the request is fully handled. It is possible the programmer may choose to implement some functionality for a request, then allow `USB_StandardRequests()` to finalize.

The complete code listing for the main endpoint 0 task is show below:

```
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_ep ep0_in  = XUD_InitEp(chan_ep0_in);

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

                if ((bmRequestType == USB_BMREQ_H2D_STANDARD_DEV) &&
                    (sp.bRequest == USB_SET_ADDRESS))
                {
                    debug_printf("Address allocated %d\n", sp.wValue);
                }
            }

            switch(bmRequestType)
            {
                /* Direction: Device-to-host
                 * Type: Standard
                 * Recipient: Interface
                 */
                case USB_BMREQ_D2H_STANDARD_INT:

                    if(sp.bRequest == USB_GET_DESCRIPTOR)
                    {
                        /* HID Interface is Interface 0 */
                        if(sp.wIndex == 0)
                        {
                            /* Look at Descriptor Type (high-byte of wValue
                               ↪ ) */
                            unsigned short descriptorType = sp.wValue & 0
                               ↪ xff00;

                            switch(descriptorType)
                            {
                                case HID_HID:
                                    result = XUD_DoGetRequest(ep0_out,
                                       ↪ ep0_in, hidDescriptor, sizeof(
                                       ↪ hidDescriptor), sp.wLength);
                                    break;

                                case HID_REPORT:
                                    result = XUD_DoGetRequest(ep0_out,
```

The skeleton `HidInterfaceClassRequests()` function deals with any outstanding HID requests. See the USB HID Specification for full request details:

```
XUD_Result_t HidInterfaceClassRequests(XUD_ep c_ep0_out, XUD_ep c_ep0_in,
  ↪ USB_SetupPacket_t sp)
{
    unsigned buffer[64];
    unsigned tmp;

    switch(sp.bRequest)
    {
        case HID_GET_REPORT:

            /* Mandatory. Allows sending of report over control pipe */
            /* Send a hid report - note the use of asm due to shared mem */
            asm("ldaw %0, dp[g_reportBuffer]": "=r"(tmp));
            asm("ldw %0, %1[0]": "=r"(tmp) : "r"(tmp));
            buffer[0] = tmp;

            return XUD_DoGetRequest(c_ep0_out, c_ep0_in, (buffer, unsigned
              ↪ char []), 4, sp.wLength);
            break;

        case HID_GET_IDLE:
            /* Return the current Idle rate - optional for a HID mouse */

            /* Do nothing - i.e. STALL */
            break;

        case HID_GET_PROTOCOL:
            /* Required only devices supporting boot protocol devices,
             * which this example does not */

            /* Do nothing - i.e. STALL */
            break;

         case HID_SET_REPORT:
            /* The host sends an Output or Feature report to a HID
             * using a cntrol transfer - optional */

            /* Do nothing - i.e. STALL */
            break;

        case HID_SET_IDLE:
            /* Set the current Idle rate - this is optional for a HID mouse
             * (Bandwidth can be saved by limiting the frequency that an
             * interrupt IN EP when the data hasn't changed since the last
             * report */

            /* Do nothing - i.e. STALL */
            break;

        case HID_SET_PROTOCOL:
            /* Required only devices supporting boot protocol devices,
             * which this example does not */

            /* Do nothing - i.e. STALL */
            break;
    }

    return XUD_RES_ERR;
}
```

If the HID request is not handles, the function returns `XUD_RES_ERR`. This results in `USB_StandardRequests()` being called, and eventually the endpoint being STALLed to indicate an unknown request.

7 L-Series support

IN THIS CHAPTER

The USB Device Library has been designed primarily for use with xCORE-USB (U-Series) devices. However, it does also support L-Series devices. This section describe the resource usage on the L-Series and changes required to build for L-Series devices.

7.1 Resource requirements

The resources used by the USB device and XUD libraries combined on an L-Series device are shown below:

Resource	Requirements
Logical Cores	2 plus one per endpoint
Channels	2 for Endpoint0 and 1 additional per IN and OUT endpoint
Timers	4 timers
Clock blocks	Clock block 0

Note: On the L-Series the XUD library uses clock block 0 and configures it to be clocked by the 60MHz clock from the ULPI transceiver. The ports it uses are in turn clocked from the clock block. Since clock block 0 is the default for all ports when enabled it is important that if a port is not required to be clocked from this 60MHz clock, then it is configured to use another clock block.

7.2 Ports and pins

The ports used for the physical connection to the external ULPI transceiver must be connected as shown in Figure 3.

In addition some ports are used internally when the XUD library is in operation. For example pins X0D2-X0D9, X0D26-X0D33 and X0D37-X0D43 on an XS1-L8-128 device should not be used.

Please refer to the device datasheet for further information on which ports are available.

Pin	Port			Signal
	1b	4b	8b	
X0D12	P1E0			ULPI_STP
X0D13	P1F0			ULPI_NXT
X0D14		P4C0	P8B0	ULPI_DATA[7:0]
X0D15		P4C1	P8B1	
X0D16		P4D0	P8B2	
X0D17		P4D1	P8B3	
X0D18		P4D2	P8B4	
X0D19		P4D3	P8B5	
X0D20		P4C2	P8B6	
X0D21		P4C3	P8B7	
X0D22	P1G0			ULPI_DIR
X0D23	P1H0			ULPI_CLK
X0D24	P1I0			ULPI_RST_N

Figure 3: L-Series required pin/port connections

7.3 Reset requirements

On the L-Series the `XUD_Manager` requires a reset port and a reset clock block to be given.

7.4 Building for L-Series

Note: `module_usb_device` and `module_xud` upon which it depends both support both U-Series and L-Series devices, but the xSOFTip Explorer will only perform resource estimation with the U-Series library.

Note: Also, tools before the 13.0 release do not support automatically changing a target library. Therefore, if using xTIMEcomposer pre-13.0 the `Makefile` generated will have to be modified in order to compile for an L-Series device. Open the `Makefile` and add the line `MODULE_LIBRARIES = xud_l`.