XMOS USB Device (XUD) Library

Version 1.0

Publication Date: 2011/01/10

XMOS

1 Introduction

This document details the use of the XMOS USB Device (XUD) Layer Library, which enables the development of high-speed USB 2.0 devices on the XMOS XS-1 architecture.

This document describes the structure of the library, its basic use, and resources required. A worked example that uses the XUD library is shown: a USB Human Interface Device (HID) Class compliant mouse The full source code for the example can be downloaded from the XMOS website.

This document assumes familiarity with the XMOS XS-1 architecture, the Universal Serial Bus 2.0 Specification (and related specifications), XMOS tool chain and XC language.

2 Overview

The XUD library allows the implementation of high-speed USB 2.0 devices using a ULPI transceiver such as the SMSC USB33XX range.

The 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 timing requirements from the application, allowing quick prototyping of all manner of USB devices.

Two libraries, with identical interfaces, are provided for the XS1-G and XS1-L series of processors.

The XUD library runs in a single thread with endpoint and application threads 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 thread diagram is shown Figure 1. Circles represent threads running on the XS1 with arrows depicting communication channels between these threads. In this configuration there is one thread 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 thread, and OUT endpoint 2 and IN endpoint 5 are dealt with by a third thread. Threads 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, threads communicate with the XUD library using both XC channels and shared memory communication. Therefore, *all threads using the XUD library must be on the same core as the library itself.*

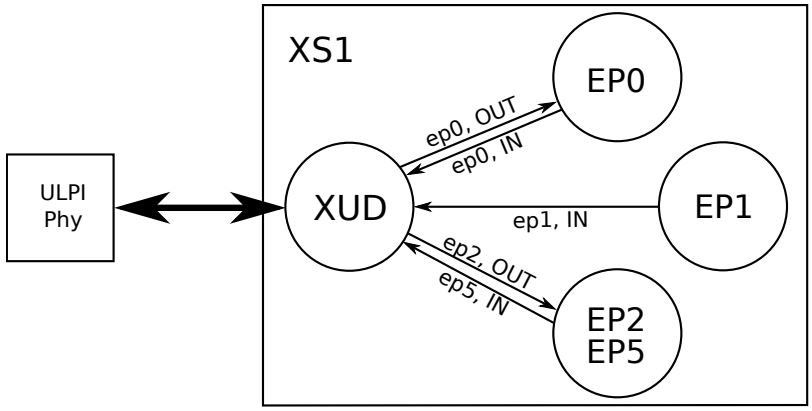

Figure 1: Example thread diagram of application using USB.

3 File Arrangement

The following list gives a brief description of the files that make up the XUD layer:

README XUD README file

EULA End User License Agreement

LICENSING Licensing information

include/xud.h User defines and functions for the XUD library

lib_l/xs1b/libxud.a Library for XS1-L

lib_g/xs1b/libxud.a Library for XS1-G

src/XUD_EpFunctions.xc Functions that control the XUD library

src/XUD_EpFuncs.S Assembler stubs of access functions

4 Resource Usage

The XUD library requires the resources described in the following sections.

4.1 Ports/Pins

The ports used for the physical connection to the ULPI transceiver must be connected as shown in Table 1.

Pin	Port			Signal
	1b	4b	8b	
XD12	P1E0			ULPI_STP
XD13	P1F0			ULPI_NXT
XD14		P4C0	P8B0	ULPI_DATA[0:7]
XD15		P4C1	P8B1	
XD16		P4D0	P8B2	
XD17		P4D1	P8B3	
XD18		P4D2	P8B4	
XD19		P4D3	P8B5	
XD20		P4C2	P8B6	
XD21		P4C3	P8B7	
XD22	P1G0			ULPI_DIR
XD23	P1H0			ULPI_CLK
XD24	P1I0			ULPI_RST_N

Table 1: ULPI required pin/port connections.

In addition some ports are used internally when the XUD library is in operation, for example pins 2-9, 26-33 and 37-43 on an L1 device should not be used, while pins 34-36 may be used as 1-bit ports. See the XS1-L Hardware Design Checklist for further information on which ports are available.

4.2 Thread speed

Due to I/O requirements the library requires a guaranteed MIPS rate to ensure correct operation. This means that thread count restrictions must be observed. The USB thread must run at at least 80 MIPS, and the threads that communicate with the USB thread must have a guaranteed 80 MIPS.

This means that for an XS1 running at 400MHz there should be no more than five threads executing at any one time that USB is being used. For a 500MHz device no more than six threads shall execute at any one time.

This restriction is only a requirement on the core on which USB threads are running. For example, a different core on an L2 device is unaffected by this restriction.

4.3 Clock Blocks

The library uses clock block 0 and configures this clock block to be clocked from 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.

4.4 Timers

Internally the XUD library allocates and uses four timers.

4.5 Memory

The library requires around 9 Kbytes of memory, of which around 6 Kbytes is code or initialized variables that must be stored in either OTP or Flash.

5 Basic Usage

This section outlines the basic usage of XUD and finishes with a worked example of a USB Human Interface Device (HID) Class compliant mouse. Basic use is termed to mean each endpoint has its own dedicated thread. Multiple endpoints in a single thread is possible but currently beyond the scope of this document.

When building, the preprocessor macro `USB_CORE` should be defined as the core number to which the USB phy is attached. On single core applications, the option `-DUSB_CORE=0` can be passed to the compiler. In multi core systems, you should check which core is used for the USB code.

5.1 XUD Thread: `XUD_Manager()`

This function must be called as a thread (normally from a top level `par` statement in `main()`) around 100 ms after power up. This is the main XUD thread that interfaces with the ULPI transceiver. It performs power-signalling/handshaking on the USB bus, and passes packets on for the various endpoints.

```
XUD_Manager(  chanend c_ep_out[],
              int num_out,
              chanend c_ep_in[],
              int num_in,
              chanend ?c_sof,
              int ep_type_table_out[],
              int ep_type_table_in[],
              port p_usb_rst,
              clock clk,
              unsigned int reset_mask,
              unsigned int desired_speed,
              chanend ?c_test_mode)
```

- `chanend c_ep_out[]` An array of channel-ends, one channel-end per output endpoint (USB OUT transaction); this includes a channel to obtain requests on endpoint 0.
- `int num_out` The number of output endpoints, should be at least 1 (for endpoint 0).
- `chanend c_ep_in[]` An array of channel-ends, one channel-end per input endpoint (USB IN transaction); this includes a channel to respond to requests on endpoint 0.
- `int num_in` The number of output endpoints, should be at least 1 (for endpoint 0).
- `chanend ?c_sof` A channel to receive SOF tokens on. This channel must be connected to a process that can receive a token once every 125 μs. If tokens are not read, the USB layer will block up. If no SOF tokens are required "null" should be used as this channel.
- `int ep_type_table_out[]`
- `int ep_type_table_in[]` Two arrays indicating the type of channel-ends. Legal types include:
  - `XUD_EPTYPE_CTL` Endpoint 0.
  - `XUD_EPTYPE_BUL` Bulk endpoint.

  - XUD_EPTYPE_ISO Isochronous endpoint.
  - XUD_EPTYPE_DIS Endpoint not used.

  The first array contains the endpoint types for each of the OUT endpoints, the second array contains the endpoint types for each of the IN endpoints.

- port p_usb_rst The port to send reset signals to.
- clock clk The clock block to use for the USB reset - this should not be clock block 0.
- int reset_mask The mask to use when sending a reset. The mask is ORed into the port to enable reset, and unset when deasserting reset. Use '-1' as a default mask if this port is not shared.
- int desired_speed This parameter specifies the desired speed the application wishes to run at i.e. full (12Mbps) or high (480Mbps) speed.

  Pass XUD_SPEED_HS for high speed, XUD_SPEED_FS for full speed

  Requesting high-speed does not guarantee that the device will be detected and operate at this speed. For example the device may be connected to the host via a full-speed hub and thus run at full-speed. See Status Reporting for full details.

  Low speed USB is not supported.

- chanend ?test_mode currently this should be set to null

5.2 EP Communication with XUD_Manager()

Communication state between a thread and the XUD library is encapsulated in an opaque type:

```
typedef XUD_ep;
```

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 thread with the following call that takes as an argument the endpoint channel connected to the XUD library:

```
XUD_ep XUD_Init_Ep(chanend c_ep)
```

Endpoint data is sent/received using three main functions, XUD_SetData(). XUD_GetData() and XUD_GetSetupData().

These assembly functions implement the low level shared memory/channel communication with the XUD_Manager() thread. For developer convenience these calls are wrapped up by XC functions. It is rare that a developer would need to call the assembly access functions directly.

These functions will automatically deal with any PID toggling required.

5.2.1 XUD_GetBuffer()

This function must be called by a thread that deals with an OUT endpoint. When the host sends data, the low level driver will fill the buffer.

```
int retVal = XUD_GetBuffer(
                  XUD_ep ep_out,
                  char buf[]);
```

- `XUD_ep ep_out` The endpoint structure created by `XUD_Init_Ep`.
- `int buf[]` The buffer to store data in. This is a buffer of integers, containing characters; the buffer must be word aligned.

The function returns the number of bytes written to the buffer (Also see Status Reporting)

5.2.2 XUD_SetBuffer()

This function must be called by a thread that deals with an IN endpoint. When the host asks for data, the low level driver will transmit the buffer to the host.

```
int retVal = XUD_SetBuffer(
                  XUD_ep ep_in,
                  char buf[],
                  int datalength);
```

- `XUD_ep ep_in` The endpoint structure created by `XUD_Init_Ep`.
- `int buf[]` The buffer of data to send out.
- `int datalength` The number of bytes in the buffer.

The function returns 0 to indicate no error (see also Status Reporting)

5.2.3 XUD_SetBuffer_ResetPid()

This function acts like XUD_SetBuffer, but it resets the PID to the requested value. See XUD_SetBuffer for a description of the first three parameters.

This function is useful for control transfers when the PID toggling needs to be reset to DATA0 for the first transfer, PID toggling resumes on the next transaction.

```
int retVal = XUD_SetBuffer_ResetPid(
                  XUD_ep ep_in,
                  unsigned char buffer[],
                  unsigned datalength,
                  unsigned char pid)
```

- unsigned char pid The new PID to use, typically this either PIDn_DATA1 or PIDn_DATA0.

The function returns 0 on success (see also Status Reporting)

5.2.4 XUD_SetBuffer_ResetPid_EpMax()

This function acts like the previous function, XUD_SetBuffer_ResetPid, but it cuts the data up in packets of a fixed maximum size. This is especially useful for control transfers where large descriptors must be sent in typically 64 byte transactions.

See XUD_SetBuffer for a description of the first, third, fourth, and sixth parameter.

```
int retVal = XUD_SetBuffer_ResetPid_EpMax(
                  XUD_ep ep_in,
                  unsigned epNum,
                  unsigned char buffer[],
                  unsigned datalength,
                  unsigned epMax,
                  unsigned char pid)
```

- unsigned epNum Not used, provide 0.
- unsigned epMax The maximum packet size in bytes.

The function returns 0 on success (see also Status Reporting)

5.2.5 XUD_DoGetRequest()

This function performs a combined XUD_SetBuffer and XUD_GetBuffer(). It transmits the buffer of the given length over the c_in channel to answer an IN request, and then waits for an OUT transaction on c_out. This function is normally called to handle requests from endpoint 0.

```
int retVal = XUD_DoGetRequest(
                  XUD_ep ep_out,
                  XUD_ep ep_in,
                  unsigned char buffer[],
                  unsigned datalength,
                  unsigned requested)
```

- XUD_ep ep_out The endpoint structure that handles endpoint 0 OUT data in the XUD manager.
- XUD_ep ep_in The endpoint structure that handles endpoint 0 IN data in the XUD manager.
- unsigned char buffer[] The data to send in response to the IN transaction. Note that this data is chopped up in fragments of at most 64 bytes.
- unsigned datalength The total length to send.
- unsigned requested The length that the host requested, pass the value sp.wLength.

This function returns 0 on success (See also Status Reporting)

5.2.6 XUD_DoSetRequestStatus()

This function sends an empty packet back on the next IN request with PID1. It is normally used by Endpoint 0 to acknowledge success.

```
int retVal = XUD_DoSetRequestStatus(
                  XUD_ep ep_in,
                  unsigned epNum)
```

- XUD_ep ep_in The endpoint structure to the XUD manager for endpoint 0 in.
- unsigned epNum Not used, provide 0.

The function returns 0 on success (Also see Status Reporting)

5.2.7 XUD_SetDevAddr()

This function must be called by endpoint 0 once a setDeviceAddress request is made by the host. It has one parameter, the new device address.

```
void XUD_SetDevAddr(int address);
```

- int address The address.

5.2.8 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-speeds). The XUD access functions discussed previously (XUD_GetData, XUD_SetData etc)return less than 0 in this case.

This reset notification is important if an endpoint thread is expecting alternating INs and OUTs. For example, consider the case where a endpoint is always expecting the sequence OUT, IN, OUT (such a control transfer). If an unplug/reset event was received after the first OUT, the host would return to sending the initial OUT after a replug, while the endpoint would hang on the IN. The endpoint needs to know of the bus reset in order to reset its state machine.

Endpoint 0 therefore requires this functionality since it deals with bi-directional control transfers.

This is also important for high-speed devices, since it is not guaranteed that the host will detect the device as a high-speed device. The device therefore needs to know what speed it is running at.

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

5.2.9 XUD_ResetEndpoint()

This function will complete a reset on an endpoint. One can either pass one or two channel-ends in (the second channel-end can be set to null). The return value should be inspected to find out what type of reset was performed.

```
XUD_ResetEndpoint(
```

```
                    XUD_ep one,
                    XUD_ep ?two);
```

- `XUD_ep one` IN or OUT endpoint structure to perform the reset on.
- `XUD_ep ?two` Optional second IN or OUT endpoint structure to perform a reset on.

In endpoint 0 typically two channels are reset (IN and OUT). In other endpoints `null` can be passed as the second parameter.

The return value is one of:

- `XUD_SPEED_HS` The host has accepted that this device can execute at high speed.
- `XUD_SPEED_FS` The device should run at full speed.

6 Descriptor Requests

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 all devices, a useful function (`DescriptorRequests()`) is provided to deal with them. Although not strictly part of the XUD library and supporting files, its use is so fundamental to a USB device that it is covered in this document.

The `DescriptorRequests()` function receives a 8 bytes Setup packet and parses it into a SetupPacket structure for further inspection:

```
typedef struct setupPacket
{
  BmRequestType bmRequestType;
  unsigned char bRequest;
  unsigned short wValue;
  unsigned short wIndex;
  unsigned short wLength;
} SetupPacket;
```

Please see Universal Serial Bus 2.0 spec for full details of setup packet and request structure.

The function then inspects this SetupPacket structure and deals with the following Standard Device requests:

- GET_DESCRIPTOR
  - DEVICE
  - CONFIGURATION
  - DEVICE_QUALIFIER
  - OTHER_SPEED_CONFIGURATION
  - STRING

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

`DescriptorRequests()` takes various arrays and a reference to a SetupPacket structure as its parameters:

```
int retVal = DescriptorRequests(
                  XUD_ep ep0_out,
                  XUD_ep c_ep0_in,
                  char hi_spd_desc[], int sz_d,
                  char hi_spd_conf_desc[], int sz_c,
                  char full_spd_desc[], int sz_fd,
                  char full_spd_cfg_desc[], int sz_fc,
                  char str_descs[][40],
                  SetupPacket &sp);
```

- `XUD_ep ep0_out`, `XUD_ep ep0_in` Two endpoint communication structures for receiving OUT transactions and responding to IN transactions for endpoint 0. Should be connected to the first two channels passed to `XUD_Manager()`./
- `char hi_spd_desc[], int sz_d` The device descriptor to use, encoded according to the USB standard. The size is passed as an integer.
- `char hi_spd_cfg_desc[], int sz_c` The configuration descriptor to use, encoded according to the USB standard. The size is passed as an integer.
- `char full_spd_desc[], int sz_fd` The device descriptor to use if the high-speed handshake fails, encoded according to the USB standard. The size is passed as an integer.
- `char full_spd_cfg_desc[], int sz_fc` The configuration descriptor to use if the high-speed handshake fails, encoded according to the USB standard. The size is passed as an integer.

- `char str_descs[][40]` The strings to use when enumerating. These strings are referred to from the descriptors.
- `SetupPacket &sp` If '0' is returned, then the setup packet is set to contain a decoded SETUP request on endpoint 0. This is a structure with the following members (that are all described in the USB standard):
  - `bmRequestType.Recipient`
  - `bmRequestType.Type`
  - `bmRequestType.Direction`
  - `bRequest`
  - `wValue`
  - `wIndex`
  - `wLength`

This function returns 0 if a request was handled without error (See also Status Reporting).

Typically the minimal code for endpoint 0 calls `DescriptorRequests` and then deals with the following cases:

```
switch(sp.bmRequestType.Type) {
case BM_REQTYPE_TYPE_STANDARD:
    switch(sp.bmRequestType.Recipient) {
    case BM_REQTYPE_RECIP_INTER:
          switch(sp.bRequest) {
          case SET_INTERFACE: break;
          case GET_INTERFACE: break;
          case GET_STATUS: break;
          }
          break;
    case BM_REQTYPE_RECIP_DEV:
          switch( sp.bRequest ) {
          case SET_CONFIGURATION: break;
          case GET_CONFIGURATION: break;
          case GET_STATUS: break;
          case SET_ADDRESS: break;
          }
          break;
     }
     break;
case BM_REQTYPE_TYPE_CLASS:
     // Optional class specific requests.
     break;
}
```

In some cases the code can simply remember the interface number and the configuration number and report those back, but only if a single interface and configuration are being used. These are single byte requests. The status requests use two bytes, and the simple response is a double zero. The set address command must result in the address being set in the XUD library by calling `XUD_SetDevAddr()` below.

7 Basic Example HS Device: USB HID device

This section contains a full worked example of a HID device. Note, this is provided as a simple example, not a full HID Mouse reference design.

7.1 Declarations

```
#include <xs1.h>
#include <print.h>

#include "xud.h"
#include "usb.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_EpType epTypeTableIn[XUD_EP_COUNT_IN] =
           {XUD_EPTYPE_CTL, XUD_EPTYPE_BUL};

/* USB Port declarations */
out port p_usb_rst       = XS1_PORT_32A;
clock    clk             = XS1_CLKBLK_3;
```

7.2 Main program

The main function fires off three processes: the XUD manager, endpoint 0, and HID. An array of channels is used for both in and out endpoints, endpoint 0 requires both, HID is just an IN endpoint.

```
int main() {
    chan c_ep_out[1], c_ep_in[2];
    par {
        XUD_Manager( c_ep_out, XUD_EP_COUNT_OUT,
                     c_ep_in, XUD_EP_COUNT_IN,
                     null, epTypeTableOut, epTypeTableIn,
                     p_usb_rst, clk, -1, XUD_SPEED_HS, null);
        Endpoint0( c_ep_out[0], c_ep_in[0]);
        hid(c_ep_in[1]);
    }
    return 0;
}
```

7.3 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(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 == 400) {
            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 and its behavior would not change dependant on bus speed, so this is

safe.

Should processing take longer that the host IN polls, the XUD_Manager thread will simply NAK.

7.4 Descriptors

The device descriptor is used by the host to decide what to do with this device. It specifies the manufacture and product, and the device class of this device.

```
static unsigned char hiSpdDesc[] = {
  0x12,  /* 0  bLength */
  0x01,  /* 1  bdescriptorType */
  0x00,  /* 2  bcdUSB */
  0x02,  /* 3  bcdUSB */
  0x00,  /* 4  bDeviceClass */
  0x00,  /* 5  bDeviceSubClass */
  0x00,  /* 6  bDeviceProtocol */
  0x40,  /* 7  bMaxPacketSize */
  0xb1,  /* 8  idVendor */
  0x20,  /* 9  idVendor */
  0x01,  /* 10 idProduct */
  0x01,  /* 11 idProduct */
  0x10,  /* 12 bcdDevice */
  0x00,  /* 13 bcdDevice */
  0x01,  /* 14 iManufacturer */
  0x02,  /* 15 iProduct */
  0x00,  /* 16 iSerialNumber */
  0x01   /* 17 bNumConfigurations */
};
```

The device qualifier descriptor defines how fields of a high speed device's device 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.

Typically this is derived mechanically from the device descriptor.

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

```
unsigned char fullSpdDesc[] = {
  0x0a,  /* 0  bLength */
  DEVICE_QUALIFIER, /* 1  bDescriptorType */
  0x00,  /* 2  bcdUSB */
  0x02,  /* 3  bcdUSB */
```

```
    0x00,   /* 4  bDeviceClass */
    0x00,   /* 5  bDeviceSubClass */
    0x00,   /* 6  bDeviceProtocol */
    0x40,   /* 7  bMaxPacketSize */
    0x01,   /* 8  bNumConfigurations */
    0x00    /* 9  bReserved  */
};
```

The configuration descriptor specifies the capabilities of one configuration—in this case there is only one configuration.

```
static unsigned char hiSpdCfgDesc[] = {
    0x09,   /* 0  bLength */
    0x02,   /* 1  bDescriptortype */
    0x22, 0x00, /* 2  wTotalLength */
    0x01,   /* 4  bNumInterfaces */
    0x01,   /* 5  bConfigurationValue */
    0x04,   /* 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 */
    0x01,   /* 6: bInterfaceSubClass */
    0x02,   /* 7: bInterfaceProtocol*/
    0x00,   /* 8  iInterface */

    0x09,   /* 0  bLength */
    0x21,   /* 1  bDescriptorType (HID) */
    0x10,   /* 2  bcdHID */
    0x01,   /* 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 */
```

```
};
```

A other speed configuration for similar reasons as the device qualifier descriptor.

```
unsigned char fullSpdCfgDesc[] = {
  0x09,   /* 0  bLength */
  OTHER_SPEED_CONFIGURATION, /* 1  bDescriptorType */
  0x12,   /* 2  wTotalLength */
  0x00,   /* 3  wTotalLength */
  0x01,   /* 4  bNumInterface: Number of interfaces*/
  0x00,   /* 5  bConfigurationValue */
  0x00,   /* 6  iConfiguration */
  0x80,   /* 7  bmAttributes */
  0xC8,   /* 8  bMaxPower */

  0x09,   /* 0 bLength */
  0x04,   /* 1 bDescriptorType */
  0x00,   /* 2 bInterfaceNumber */
  0x00,   /* 3 bAlternateSetting */
  0x00,   /* 4 bNumEndpoints */
  0x00,   /* 5 bInterfaceClass */
  0x00,   /* 6 bInterfaceSubclass */
  0x00,   /* 7 bInterfaceProtocol */
  0x00,   /* 8 iInterface */
};
```

An array of strings supplies all the strings that are referenced from the descriptors (using fields such as ‘iInterace’, ‘iProduct’ etc). String 0 is the language descriptor, and is interpreted as “no string supplied” when used as an index value.

```
static unsigned char stringDescriptors[][40] = {
  "\009\004",
    "XMOS",             // iManufacturer
   "Example mouse",     // iProduct
    "",
   "Config name"        // Configuration name
};
```

Finally, HID devices need an extra descriptor that will be requested via endpoint 0. See the USB HID documentation for details.

```
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, 0x06,  // Input (Data, Relative)
    0xC0,        // End collection
    0x09, 0x38,  // Usage (Wheel)
    0x95, 0x01,  // Report count (1)
    0x81, 0x06,  // 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
};
```

7.5 Endpoint 0

Most enumeration requests are dealt with by the DescriptorRequests() function. The complete HID endpoint 0 thread is supplied below:

```
void Endpoint0( chanend chan_ep0_out, chanend chan_ep0_in) {
  unsigned char buffer[1024];
  SetupPacket sp;
  unsigned int current_config = 0;

  XUD_ep c_ep0_out = XUD_Init_Ep(chan_ep0_out);
  XUD_ep c_ep0_in  = XUD_Init_Ep(chan_ep0_in);
```

```
while(1) {
  /* Do standard enumeration requests */
  int retVal = 0;

  retVal = DescriptorRequests(c_ep0_out, c_ep0_in, hiSpdDesc,
    sizeof(hiSpdDesc), hiSpdConfDesc, sizeof(hiSpdConfDesc),
    fullSpdDesc, sizeof(fullSpdDesc), fullSpdConfDesc,
    sizeof(fullSpdConfDesc), stringDescriptors, sp);

  if (retVal)
  {
    /* Request not covered by XUD_DoEnumReqs() so decode ourselves */
    switch(sp.bmRequestType.Type)
    {
      /* Class request */
        case BM_REQTYPE_TYPE_CLASS:
          switch(sp.bmRequestType.Recipient)
          {
            case BM_REQTYPE_RECIP_INTER:

            /* Inspect for HID interface num */
            if(sp.wIndex == 0)
            {
              HidInterfaceClassRequests(c_ep0_out, c_ep0_in, sp);
            }
            break;

          }
          break;

        case BM_REQTYPE_TYPE_STANDARD:
          switch(sp.bmRequestType.Recipient)
          {
            case BM_REQTYPE_RECIP_INTER:

              switch(sp.bRequest)
              {
                /* Set Interface */
                  case SET_INTERFACE:
                    /* No data stage for this request,
                     * just do data stage */
                    XUD_DoSetRequestStatus(c_ep0_in, 0);
                    break;

                  case GET_INTERFACE:
                    buffer[0] = 0;
```

```
                        XUD_DoGetRequest(c_ep0_out, c_ep0_in,
                          buffer,1, sp.wLength );
                        break;

                      case GET_STATUS:
                        buffer[0] = 0;
                        buffer[1] = 0;
                        XUD_DoGetRequest(c_ep0_out, c_ep0_in,
                          buffer, 2, sp.wLength);
                        break;

                      case GET_DESCRIPTOR:
                        if((sp.wValue & 0xff00) ==  0x2200)
                        {
                          retVal = XUD_DoGetRequest(c_ep0_out, c_ep0_in,
hidReportDescriptor,
                            sizeof(hidReportDescriptor),sp.wLength,
                            sp.wLength);
                        }
                        break;

                    }
                    break;

                /* Recipient: Device */
                case BM_REQTYPE_RECIP_DEV:

                  /* Standard Device requests (8) */
                  switch( sp.bRequest )
                  {
                    /* Standard request: SetConfiguration */
                    case SET_CONFIGURATION:

                      /* Set the config */
                      current_config = sp.wValue;

                      /* No data stage for this request,
                         just do status stage */
                      XUD_DoSetRequestStatus(c_ep0_in,  0);
                      break;

                    case GET_CONFIGURATION:
                      buffer[0] = (char)current_config;
                      XUD_DoGetRequest(c_ep0_out, c_ep0_in, buffer,
                        1, sp.wLength);
                      break;
```

```
                case GET_STATUS:
                  buffer[0] = 0;
                  buffer[1] = 0;
                  if (hiSpdConfDesc[7] & 0x40)
                      buffer[0] = 0x1;
                  XUD_DoGetRequest(c_ep0_out, c_ep0_in, buffer,
                    2, sp.wLength);
                  break;

                case SET_ADDRESS:
                  /* Status stage: Send a zero length packet */
                  retVal = XUD_SetBuffer_ResetPid(c_ep0_in,
                    buffer, 0, PIDn_DATA1);

                  /* wait until ACK is received for status stage
                     before changing address */
                  {
                      timer t;
                      unsigned time;
                      t :> time;
                      t when timerafter(time+50000) :> void;
                  }

                  /* Set device address in XUD */
                  XUD_SetDevAddr(sp.wValue);
                  break;

                default:
                  break;

              }
              break;

            default:
              /* Request to a recipient we didn't recognize */
              break;
          }
          break;

        default:
          /* Error */
          break;

      }

    } /* if XUD_DoEnumReqs() */
```

```
      if (retVal == -1)
      {
        XUD_ResetEndpoint(c_ep0_out, c_ep0_in);
      }
    }
}
```

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

```
int HidInterfaceClassRequests(XUD_ep c_ep0_out, XUD_ep c_ep0_in,
  SetupPacket sp)
{
  unsigned char buffer[64];
  switch(sp.bRequest )
  {
    case GET_REPORT:
      break;

    case GET_IDLE:
      break;

    case GET_PROTOCOL:      /* Required only for boot devices */
      break;

    case SET_REPORT:
      XUD_GetBuffer(c_ep0_out, buffer);
      return XUD_SetBuffer_ResetPid(c_ep0_in, buffer, 0, PIDn_DATA1);
      break;

    case SET_IDLE:
      return XUD_SetBuffer_ResetPid(c_ep0_in, buffer, 0, PIDn_DATA1);
      break;

    case SET_PROTOCOL:      /* Required only for boot devices */
      return XUD_SetBuffer_ResetPid(c_ep0_in, buffer, 0, PIDn_DATA1);
      break;

    default:
      /* Error case */
      break;
  }

  return 0;
}
```

Release History

Date	Release	Comment
2010-07-22	1.0b	Beta release
2011-01-06	1.0	Updates for API changes