USB Library

A software defined, industry-standard, USB library that allows you to control an USB bus via xCORE ports.

The library provides functionality to act as a USB *device*.

This library is aimed primarily for use with xCORE U-Series or the xCORE-200 Series devices but it does also support xCORE L-Series devices.

Features

- USB 2.0 Full-speed (12Mbps) and High-speed (480Mbps) modes.
- Device mode.
- Bulk, control, interrupt and isochronous endpoint types supported.

Typical Resource Usage

This following table shows typical resource usage in some different configurations. Exact resource usage will depend on the particular use of the library by the application.

Configuration	Pins	Ports	Clocks	Ram	Logical cores
USB device (U series)	23 (internal)	11	0	~8.8K	1
USB device (xCORE-200 series)	23 (internal)	11	0	~9.3K	1
USB device (L series)	13	8	0	~8.4K	1

Software version and dependencies

This document pertains to version 3.1.0 of this library. It is known to work on version 14.1.1 of the xTIMEcomposer tools suite, it may work on other versions.

This library depends on the following other libraries:

- lib_logging (>=2.0.0)
- lib_xassert (>=2.0.0)
- lib_gpio (>=1.0.0)

Related application notes

The following application notes use this library:

- AN00125 - USB mass storage device class
- AN00126 - USB printer device class
- AN00127 - USB video device class
- AN00128 - USB Audio device class
- AN00129 - USB HID device class
- AN00130 - Extended USB HID class
- AN00131 - USB CDC-EDC device class
- AN00132 - USB Image device class

1 Hardware setup

1.1 Physical characteristics and setup

Details on the physical characteristics and how to integrate the USB connection to the xCORE device into your system are all contain in the devices datasheet. *Please refer to the device datasheet for this information.*

1.2 Ports/Pins

1.2.1 U-Series

The U-Series of devices have 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 Table 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	P32A28
X0D27		P4E1	P8C1	P16B1	P32A29
X0D28		P4F0	P8C2	P16B2	
X0D29		P4F1	P8C3	P16B3	
X0D30		P4F2	P8C4	P16B4	
X0D31		P4F3	P8C5	P16B5	
X0D32		P4E2	P8C6	P16B6	P32A30
X0D33		P4E3	P8C7	P16B7	P32A31
X0D34	P1K0				
X0D36	P1M0		P8D0	P16B8	
X0D37	P1N0		P8C1	P16B1	
X0D38	P1O0		P8C2	P16B2	
X0D39	P1P0		P8C3	P16B3	

Table 1: U-Series required pin/port connections

1.2.2 xCORE-200 Series

The xCORE 200 series of devices have an integrated USB transceiver. Some ports are used to communicate with the USB transceiver inside the xCORE-200 series packages. These ports/pins should not be used when USB functionality is enabled. The ports/pins are shown in Table 2.

Pin	Port				
	1b	4b	8b	16b	32b
X0D00	P1A0				
X0D02		P4A0	P8A0	P16A0	P32A20
X0D03		P4A1	P8A1	P16A1	P32A21
X0D04		P4B0	P8A2	P16A2	P32A22
X0D13		P4B1	P8A3	P16A3	P32A23
X0D22		P4B2	P8A4	P16A4	P32A24
X0D23		P4B3	P8A5	P16A5	P32A25
X0D34		P4A2	P8A6	P16A6	P32A26
X0D09		P4A3	P8A7	P16A7	P32A27
X0D10	P1C0				
X0D12	P1E0				
X0D13	P1F0				
X0D14		P4C0	P8B0	P16A8	
X0D15		P4C1	P8B1	P16A9	
X0D16		P4D0	P8B2	P16A10	
X0D17		P4D1	P8B3	P16A11	
X0D18		P4D2	P8B4	P16A12	
X0D19		P4D3	P8B5	P16A13	
X0D20		P4C2	P8B6	P16A14	
X0D21		P4C3	P8B7	P16A15	
X0D22	P1G0				
X0D23	P1H0				
X0D24	P1I0				
X0D34	P1K0				

Table 2: xCORE-200 series required pin/port connections

1.2.3 L-Series

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

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

Table 3: ULPI required pin/port connections

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-L device should not be used.

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

2 Usage

The USB library consists of a single main component: the XUD device driver. A typical application will use have the following software architecture:

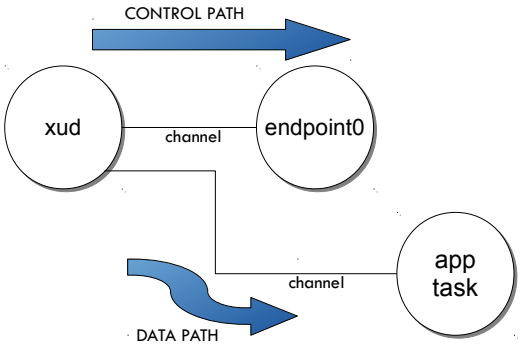

Figure 1: USB task diagram

Here, the application interacts with the USB library in two ways. Data is sent and received directly from the XUD component. This provides the path to the USB Endpoints of the device. Multiple tasks can be connected to the XUD component to handle multiple endpoints in parallel. The application also interacts with the special USB Endpoint 0 which handle configuration calls to the host. Each application will develop its own Endpoint 0 code using the functions provided by the USB library.

2.1 The XUD (XMOS USB device) driver

The XUD component 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, xCORE-200 Series and L-Series devices.

The U-Series and xCORE-200 Series include 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 variant of the component, with identical interfaces, are provided - one for U- and xCORE-200 Series and one for L-Series devices.

The XUD component 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. Connected tasks must be ready to communicate with the XUD component whenever the host demands its attention. If not, the XUD component will NAK.

It is important to note that, for performance reasons, tasks communicate with the XUD component using a combination of both xC channels and shared memory. It is therefore madatory that *all cores that directly communicate with the XUD task must be on the same tile as the task itself*.

The main XUD task is implement by the `xud()` function (for U-series and xCORE-200 series devices) or the `xud_l_series()` function (for L-series devices). The 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.

2.2 Core speed

Due to I/O requirements, the XUD component requires a guaranteed MIPS rate to ensure correct operation. This means that core count restrictions must be observed. The XUD task must run on a core running at least at a speed of 80 MHz.

This means that for an xCORE device running at 400MHz there should be no more than five cores executing at any time when using the XUD. For a 500MHz device 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 component is running. For example, the other tile on an L16 device is unaffected by this restriction.

2.3 Setting up the XUD in your program

In your `main` function, the application must call the xud or `xud_l_series` function:

```
int main()
{
    chan c_ep_out[3], c_ep_in[2];
    par {
        on tile[0]: xud(c_ep_out, 3,
                        c_ep_in, 2,
                        null, XUD_SPEED_HS, XUD_PWR_SELF);

        on tile[0]: Endpoint0(c_ep_out[0], c_ep_in[0]);

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

The XUD is connected to an array of channels for the IN endpoints and an array of channels for the OUT endpoints.

2.4 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.

2.5 PwrConfig

The `PwrConfig` parameter to XUD function indicates if the device is bus or self-powered.

Valid values for this parameter are XUD_PWR_SELF and XUD_PWR_BUS.

When XUD_PWR_SELF is used, the XUD monitors the VBUS input for a valid voltage and reponds appropriately. The USB Specification states that the devices pull-ups must be disabled when a valid VBUS is not present. This is important when submitting a device for compliance testing since this is explicitly tested.

If the device is bus-powered XUD_PWR_SELF can be used since is assumed that the device is not powered up when VBUS is not present and therefore no voltage monitoring is required. In this configuration the VBUS input to the device/PHY need not be present.

XUD_PWR_BUS can be used in order to run on a self-powered board without provision for VBUS wiring to the PHY/device, but this is not advised.

2.6 Endpoint communication with the XUD component

Communication state between a core and the XUD component is encapsulated in an opaque type XUD_ep (see §3.2).

All client calls communicating with the XUD component 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. This function also takes an argument to indicate the transfer-type of the endpoint (bulk, control, isochronous or interrupt) as well as whether the endpoint wishes to be informed about bus-resets (see §2.9).

For example, this code initializes a bulk endpoint:

```
void my_application(chanend c_ep_out) {
   XUD_ep ep_out = XUD_InitEp(chan_ep0_out, XUD_EPTYPE_BUL);
   ...
```

The endpoint types are show in the following table:

Type	Description
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

Table 4: Endpoint types

In addition XUD_STATUS_ENABLE can be ORed ino the endpoint type to indicate an endpoints that wishes to be informed of USB bus resets (see §2.9).

2.7 Blocking sending and receiving data

An application specific endpoint can send data using several functions described in §3.2. In particular XUD_SetBuffer() will send data from the host and XUD_GetBuffer() will receive data from the host. These functions will automatically deal with any low-level complications required such as Packet ID toggling etc.

The XUD_SetBuffer_EpMax function provides a similar function to XUD_SetBuffer function but it breaks 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.

Here is an example of sending a 4 bytes packet to the host:

```
void my_application(chanend c_ep_in) {
  XUD_ep ep_out = XUD_InitEp(chan_ep0_in, XUD_EPTYPE_BUL);
  ...
  char data[4];
  ...
  XUD_SetBuffer(ep_hid, data, 4);
  ...
```

Note that these functions are blocking - they will wait until the host performs the transaction with the device before you program can proceed.

2.8 Asynchronous sending and recieving of data

Functions such as XUD_SetBuffer() and XUD_GetBuffer() block until data has either been successfully sent or received to or from the host. For this reason it is not generally possible to handle multiple endpoints in a single core efficiently (or at all, depending on the protocols involved). The XUD library therefore provides functions to allow the separation of requesting to send/receive a packet and the notification of a successful transfer. This is based on the xC `select` statement language feature.

General operation is as follows:

- An XUD_SetReady_ function is called to mark an endpoint as ready to send or receive data. (see §3.2)
- An `select` statement is used to wait for, and capture, send/receive notifications from the XUD task.

Once an endpoint has been marked ready to send/receive by calling one of the above XUD_SetReady_ functions, a select statement can be used to handle notifications of a packet being sent/received from the XUD tasks. These notifications are communicated via channels and can be handled via the XUD_*_Select functions.

The following example shows these asynchronous functions in use:

```
void ExampleEndpoint(chanend c_ep_out, chanend c_ep_in)  {
   unsigned char rxBuffer[1024];
   unsigned char txBuffer[] = {0, 1, 2, 3, 4};
   int length, returnVal;

   XUD_ep ep_out = XUD_InitEp(c_ep_out, XUD_EPTYPE_BUL);
   XUD_ep ep_in = XUD_InitEp(c_ep_in, XUD_EPTYPE_BUL);

   /* Mark OUT endpoint as ready to receive */
   XUD_SetReady_Out(ep_out, rxBuffer);
   XUD_SetReady_In(ep_in, txBuffer, 5);

   while(1) {
    select {
     case XUD_GetData_Select(c_ep_out, ep_out, length):
         /* Packet from host recieved */
         for(int i = 0; i< length; i++) {
           /* Process packet... */
         }
        /* Mark EP as ready again */
        XUD_SetReady_Out(ep_out, rxBuffer);
        break;

      case XUD_SetData_Select(c_ep_in, ep_in, returnVal):
        /* Packet successfully sent to host */
        /* Create new buffer */
        for(int i = 0; i < 5; i++) {
          txBuffer[i]++;
        }

        /* Mark EP as ready again */
        XUD_SetReady_In(ep_in, txBuffer, 5);
        break;
    }
   }
  }
```

2.9 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 endpoint type when calling the XUD_InitEp() function.

This means that endpoints are notified of USB bus resets (and bus-speed changes). The XUD access functions (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.

2.10 SOF Channel

An application can pass a channel-end to the c_sof parameter of the XUD component. This will cause a word of data to be output every time the device receives a SOF from the host. This can be used for timing information for audio devices etc. If this functionality is not required null should be passed as the parameter. Please note, if a channel-end is passed into XUD component there must be a responsive task ready to receive SOF notifications otherwise the XUD component will be blocked attempting to send these messages.

2.11 USB Test Modes

XUD supports the required test modes for USB Compliance testing.

XUD accepts commands from the endpoint 0 channels (in or out) to signal which test mode to enter via the XUD_SetTestMode() function. The commands are based on the definitions of the Test Mode Selector Codes in the USB 2.0 Specification Table 11-24. The supported test modes are summarised in Table 5.

Value	Test Mode Description
1	Test_J
2	Test_K
3	Test_SE0_NAK
4	Test_Packet

Table 5: Supported Test Mode Selector Codes

The passing other codes endpoints other than 0 to XUD_SetTestMode() could result in undefined behaviour.

As per the USB 2.0 Specification a power cycle or reboot is required to exit the test mode.

2.12 Implementing your own Endpoint 0 handler

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_EPTYPE_CTL | XUD_STATUS_ENABLE);
XUD_ep ep0_in  = XUD_InitEp(chan_ep0_in, XUD_EPTYPE_CTL | XUD_STATUS_ENABLE);
```

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).

2.13 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 relevant USB Specification Documents.

3 API

All USB functions can be accessed via the usb.h header:

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

You will also have to add lib_usb to the USED_MODULES field of your application Makefile.

The application Makefile also needs to add flags to set the XUD_SERIES_SUPPORT define e.g.:

```
XCC_FLAGS = ... -DXUD_SERIES_SUPPORT=XUD_U_SERIES
```

The possible values of this define are XUD_U_SERIES, XUD_X200_SERIES or XUD_L_SERIES to specify U-series, xCORE-200 series or L-series support respectively.

For L-series devices, the USB library uses the hardware clock 0 which is usually reserved as the default clock. To ensure other code using ports clocked of the default clock block still function correctly. The application Makefile should also change the default clock block to a different clock e.g.:

```
XCC_FLAGS = ... -default-clkblk XS1_CLKBLK_5
```

This is *not* required for U-series of xCORE-200 series devices.

3.1 Creating an USB device task instance

Function	xud
Description	USB device driver (U-series). This performs the low-level USB I/O operations. Note that this needs to run in a thread with at least 80 MIPS worst case execution speed.

Type	`void xud(chanend c_epOut[noEpOut],`
	`  static const size_t noEpOut,`
	`  chanend c_epIn[noEpIn],`
	`  static const size_t noEpIn,`
	`  chanend ?c_sof,`
	`  XUD_BusSpeed_t desiredSpeed,`
	`  XUD_PwrConfig pwrConfig)`

Parameters	`c_epOut` An array of channel ends, one channel end per output endpoint (USB OUT transaction); this includes a channel to obtain requests on Endpoint 0. `noEpOut` The number of output endpoints, should be at least 1 (for Endpoint 0). `c_epIn` 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. `noEpIn` The number of input endpoints, should be at least 1 (for Endpoint 0). `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 ms. If tokens are not read, the USB layer will lock up. If no SOF tokens are required `null` should be used for this parameter. `desiredSpeed` This parameter specifies what speed the device will attempt to run at i.e. full-speed (ie 12Mbps) or high-speed (480Mbps) if supported by the host. Pass XUD_SPEED_HS if high-speed is desired or XUD_SPEED_FS if not. Low speed USB is not supported by XUD. `pwrConfig` Specifies whether the device is bus or self-powered. When self-powered the XUD will monitor the VBUS line for host disconnections. This is required for compliance reasons. Valid values are XUD_PWR_SELF and XUD_PWR_BUS.

Function	xud_l_series
Description	USB device driver (L-series). This performs the low-level USB I/O operations. Note that this needs to run in a thread with at least 80 MIPS worst case execution speed.

Type	`void`
	`xud_l_series(chanend c_epOut[noEpOut],`
	`static const size_t noEpOut,`
	`chanend c_epIn[noEpIn],`
	`static const size_t noEpIn,`
	`chanend ?c_sof,`
	`client output_gpio_if ?p_usb_rst,`
	`XUD_BusSpeed_t desiredSpeed,`
	`XUD_PwrConfig pwrConfig)`

Parameters	`c_epOut` An array of channel ends, one channel end per output endpoint (USB OUT transaction); this includes a channel to obtain requests on Endpoint 0. `noEpOut` The number of output endpoints, should be at least 1 (for Endpoint 0). `c_epIn` 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. `noEpIn` The number of input endpoints, should be at least 1 (for Endpoint 0). `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 ms. If tokens are not read, the USB layer will lock up. If no SOF tokens are required `null` should be used for this parameter. `p_usb_rst` This is a GPIO interface which should be current to the external phy reset line. See the GPIO library for details on the interface.es. `desiredSpeed` This parameter specifies what speed the device will attempt to run at i.e. full-speed (ie 12Mbps) or high-speed (480Mbps) if supported by the host. Pass XUD_SPEED_HS if high-speed is desired or XUD_SPEED_FS if not. Low speed USB is not supported by XUD. `pwrConfig` Specifies whether the device is bus or self-powered. When self-powered the XUD will monitor the VBUS line for host disconnections. This is required for compliance reasons. Valid values are XUD_PWR_SELF and XUD_PWR_BUS.

3.2 The XUD Endpoint API

3.2.1 Supporting types

Type	XUD_Result_t
Description	Type containing the result of a endpoint function call.
Values	XUD_RES_RST A USB reset has occurred. XUD_RES_OKAY Operation completed successfully. XUD_RES_ERR An error has occurred.

3.2.2 Setting up the endpoint

Type	XUD_ep
Description	Opaque type representing endpoint identifiers.

Function	XUD_InitEp
Description	Initialises an XUD_ep.
Type	`XUD_ep XUD_InitEp(chanend c_ep, XUD_EpType epType)`
Parameters	c_ep — Endpoint channel to be connected to the XUD library. epType — Indicates the type of the endpoint. Legal types include: XUD_EPTYPE_CTL (Endpoint 0), XUD_EPTYPE_BUL (Bulk endpoint), XUD_EPTYPE_ISO (Isochronous endpoint), XUD_EPTYPE_INT (Interrupt endpoint), XUD_EPTYPE_DIS (Endpoint not used).
Returns	Endpoint identifier

3.2.3 OUT endpoint data handling

Function	XUD_GetBuffer
Description	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. It pauses until data is available.

Type	`XUD_Result_t XUD_GetBuffer(XUD_ep ep_out,`
	`unsigned char buffer[],`
	`unsigned &length)`

Parameters	`ep_out` The OUT endpoint identifier (created by `XUD_InitEP`). `buffer` The buffer in which to store data received from the host. The buffer is assumed to be word aligned. `length` The number of bytes written to the buffer
Returns	XUD_RES_OKAY on success

3.2.4 OUT endpoint data handling (asynchronous)

Function	XUD_SetReady_Out
Description	Marks an OUT endpoint as ready to receive data.
Type	`int XUD_SetReady_Out(XUD_ep ep, unsigned char buffer[])`
Parameters	ep — The OUT endpoint identifier (created by XUD_InitEp). buffer — The buffer in which to store data received from the host. The buffer is assumed to be word aligned.
Returns	XUD_RES_OKAY on success

Function	XUD_GetData_Select
Description	Select handler function for receiving OUT endpoint data in a select.

Type	`void XUD_GetData_Select(chanend c,`
	`XUD_ep ep,`
	`unsigned &length,`
	`XUD_Result_t &result)`

Parameters	c — The chanend related to the endpoint ep — The OUT endpoint identifier (created by XUD_InitEp). length — Passed by reference. The number of bytes written to the buffer, result — XUD_Result_t passed by reference. XUD_RES_OKAY on success

3.2.5 IN endpoint data handling

Function	XUD_SetBuffer
Description	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.

Type	`XUD_Result_t XUD_SetBuffer(XUD_ep ep_in,`
	`unsigned char buffer[],`
	`unsigned datalength)`

Parameters	`ep_in` The endpoint identifier (created by `XUD_InitEp`). `buffer` The buffer of data to transmit to the host. `datalength` The number of bytes in the buffer.
Returns	XUD_RES_OKAY on success

Function	XUD_SetBuffer_EpMax
Description	Similar to XUD_SetBuffer but breaks up data transfers into smaller packets. 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.

Type	`XUD_Result_t XUD_SetBuffer_EpMax(XUD_ep ep_in,`
	`unsigned char buffer[],`
	`unsigned datalength,`
	`unsigned epMax)`

Parameters	`ep_in` The IN endpoint identifier (created by `XUD_InitEp`). `buffer` The buffer of data to transmit to the host. `datalength` The number of bytes in the buffer. `epMax` The maximum packet size in bytes.
Returns	XUD_RES_OKAY on success

3.2.6 IN endpoint data handling (asynchronous)

Function	XUD_SetReady_In	
Description	Marks an IN endpoint as ready to transmit data.	
Type	`XUD_Result_t XUD_SetReady_In(XUD_ep ep, unsigned char buffer[], int len)`	
Parameters	ep	The IN endpoint identifier (created by XUD_InitEp).
	buffer	The buffer to transmit to the host. The buffer is assumed be word aligned.
	len	The length of the data to transmit.
Returns	XUD_RES_OKAY on success	

Function	XUD_SetData_Select	
Description	Select handler function for transmitting IN endpoint data in a select.	
Type	`void XUD_SetData_Select(chanend c, XUD_ep ep, XUD_Result_t &result)`	
Parameters	c	The chanend related to the endpoint
	ep	The IN endpoint identifier (created by XUD_InitEp).
	result	Passed by reference. XUD_RES_OKAY on success

3.3 Endpoint0 utility functions

Function	XUD_DoGetRequest
Description	Performs a combined XUD_SetBuffer and XUD_GetBuffer. It transmits the buffer of the given length over the ep_in endpoint to answer an IN request, and then waits for a 0 length Status OUT transaction on ep_out. This function is normally called to handle Get control requests to Endpoint 0.

Type	`XUD_Result_t XUD_DoGetRequest(XUD_ep ep_out,`
	`XUD_ep ep_in,`
	`unsigned char buffer[],`
	`unsigned length,`
	`unsigned requested)`

Parameters	`ep_out` The endpoint identifier that handles Endpoint 0 OUT data in the XUD manager. `ep_in` The endpoint identifier that handles Endpoint 0 IN data in the XUD manager. `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. `length` Length of data to be sent. `requested` The length that the host requested, (Typically pass the value wLength).
Returns	XUD_RES_OKAY on success

Function	XUD_DoSetRequestStatus
Description	This function sends an empty packet back on the next IN request with PID1. It is normally used by Endpoint 0 to acknowledge success of a control transfer.
Type	`XUD_Result_t XUD_DoSetRequestStatus(XUD_ep ep_in)`
Parameters	`ep_in` The Endpoint 0 IN identifier to the XUD manager.
Returns	XUD_RES_OKAY on success

Function	XUD_SetDevAddr
Description	Sets the device's address. This function must be called by Endpoint 0 once a setDeviceAddress request is made by the host. Must be run on USB core
Type	`XUD_Result_t XUD_SetDevAddr(unsigned addr)`
Parameters	addr New device address.

Function	XUD_SetStall
Description	Mark an endpoint as STALLed. It is cleared automatically if a SETUP received on the endpoint. Must be run on same tile as XUD core
Type	`void XUD_SetStall(XUD_ep ep)`
Parameters	ep XUD_ep type.

Function	XUD_SetStallByAddr
Description	Mark an endpoint as STALL based on its EP address. Cleared automatically if a SETUP received on the endpoint. Note: the IN bit of the endpoint address is used. Must be run on same tile as XUD core
Type	`void XUD_SetStallByAddr(int epNum)`
Parameters	epNum Endpoint number.

Function	XUD_ClearStall
Description	Mark an endpoint as NOT STALLed. Must be run on same tile as XUD core
Type	`void XUD_ClearStall(XUD_ep ep)`
Parameters	ep XUD_ep type.

Function	XUD_ClearStallByAddr
Description	Mark an endpoint as NOT STALLed based on its EP address. Note: the IN bit of the endpoint address is used. Must be run on same tile as XUD core
Type	`void XUD_ClearStallByAddr(int epNum)`
Parameters	`epNum` Endpoint number.

Function	XUD_ResetEndpoint
Description	This function will complete a reset on an endpoint. Can take one or two XUD_ep as parameters (the second parameter can be set to `null`). The return value should be inspected to find the new bus-speed. In Endpoint 0 typically two endpoints are reset (IN and OUT). In other endpoints `null` can be passed as the second parameter.
Type	`XUD_BusSpeed_t XUD_ResetEndpoint(XUD_ep one, XUD_ep &?two)`
Parameters	`one` IN or OUT endpoint identifier to perform the reset on. `two` Optional second IN or OUT endpoint structure to perform a reset on.
Returns	Either XUD_SPEED_HS - the host has accepted that this device can execute at high speed, or XUD_SPEED_FS - the device is runnig at full speed.

Function	XUD_SetTestMode
Description	Enable a specific USB test mode in XUD. Must be run on same tile as XUD core
Type	`void XUD_SetTestMode(XUD_ep ep, unsigned testMode)`
Parameters	`ep` XUD_ep type (must be endpoint 0 in or out) `testMode` The desired test-mode

Type	USB_BmRequestType_t
Description	Data structure describing a USB request type.

Fields	`unsigned char Recipient`
	Where the request is directed to:.
	• 0b00000: Device * 0b00001: Specific interface * 0b00010: Specific endpoint * 0b00011: Other element in device
	
	`unsigned char Type`
	The type of the request:.
	• 0b00: Standard request * 0b01: Class specific request * 0b10: Request by vendor specific driver
	
	`unsigned char Direction`
	The direction of the request:.
	• 0 (Host->Dev) * 1 (Dev->Host)

Type	USB_SetupPacket_t
Description	Setup packet data structure.

Fields	`USB_BmRequestType_t bmRequestType`
	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).

Function	USB_GetSetupPacket
Description	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

Function	USB_StandardRequests
Description	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, unsigned char ?devDesc_hs[], int devDescLength_hs, unsigned char ?cfgDesc_hs[], int cfgDescLength_hs, unsigned char ?devDesc_fs[], int devDescLength_fs, unsigned char ?cfgDesc_fs[], int cfgDescLength_fs, char *unsafe strDescs[], int strDescsLength, USB_SetupPacket_t &sp, XUD_BusSpeed_t usbBusSpeed)`

Continued on next page

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 standard
	
	`devDescLength_fs`
	Length of device descriptor in bytes. If 0 the HS device descriptor is used.
	
	`cfgDesc_fs`
	Configuration descriptor
	
	`cfgDescLength_fs`
	Length of config descriptor in bytes. If 0 the HS config descriptor is used.
	
	`strDescs`
	
	`strDescsLength`
	
	`sp` `USB_SetupPacket_t` (passed by ref) in which the setup data is returned
	
	`usbBusSpeed`
	The current bus speed (XUD_SPEED_HS or XUD_SPEED_FS)

Returns	Returns XUD_RES_OKAY on success.

APPENDIX A - Known Issues

There are no known issues with this library.

APPENDIX B - USB library change log

B.1 3.1.0

- ADDED: Bulk read benchmark to AN00136
- **CHANGE: Throughput performance of bulk example (AN00136) dramatically improved using async** API of host libusb
- CHANGE: Standard descriptor structs now packed (and only available from C)
- RESOLVED: Initialisation issue on xCORE-200

B.2 3.0.0

- Initial version of lib_usb. The code has been moved over from the old module_xud (sc_xud), module_usb (sc_usb) and module_usb_shared (sc_usb) repositories. Please see those repos for old changes.
- Split XUD_Manager in separate xud functions for U/X200 series and L series for a simpler interface.
- Removed the EpTypeTable argument from XUD_Manager. Now endpoints register their type via an extra argument to XUD_InitEp. This makes multiple endpoints programs easier to maintain.