XMOS USB Device (XUD) Library

Document Number: XM003082A

Publication Date: 2014/7/15

XMOS

SYNOPSIS

This document details the XMOS USB Device (XUD) Library. This library enables the development of USB 2.0 devices on the XMOS xCORE architecture.

This document describes the structure of the library, its basic use and resources required.

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.

Table of Contents

1 Overview

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

For the L and G series the implementation requires the use of an external ULPI transceiver such as the SMSC USB33XX range. U-Series devices include an integrated USB transceiver. Three libraries, with identical API, are provided - one each for L, G and U-Series devices.

Please note, G-series is not recommended for new designs.

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 development of all manner of USB 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.

One channel is required per IN or OUT endpoint. Endpoint 0 (the control endpoint) requires two channels, one for each direction. Please 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 1. 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*.

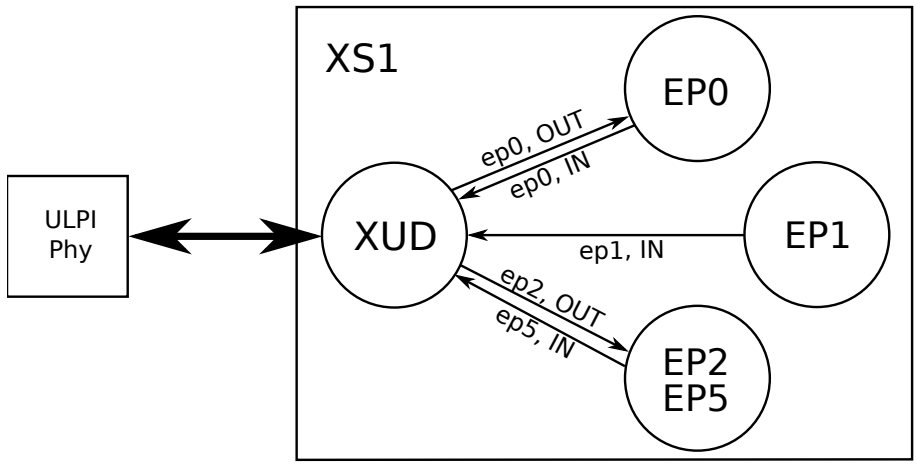

Figure 1: XUD Overview

2 File Arrangement

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

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

lib/xs1b/libxud_l.a
Library for L-Series devices.

lib/xs1b/libxud_u.a
Library for U-Series devices.

lib/xs1b/libxud_g.a
Library for G-Series devices.

src/XUD_EpFunctions.xc
User functions that control the XUD library.

src/XUD_EpFuncs.S
Assembler stubs of access functions.

src/XUD_Ports.xc
Definition of port mapping.

3 Resource Usage

IN THIS CHAPTER

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

3.1 Ports/Pins

3.1.1 G/L-Series

The ports used for the physical connection to the external ULPI transceiver must be connected as shown in Figure 2.

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 2: 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.

3.1.2 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 Figure 3.

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 3: U-Series required pin/port connections

3.2 Core Speed

Due to I/O 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 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_Manager` is running. For example, the other tile on an L16 device is unaffected by this restriction.

3.3 Clock Blocks

3.3.1 G/L-Series

The XUD library uses one clock block (clock block 0) and configures this clock block 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.

3.3.2 U-Series

The XUD library uses two clock blocks (clock blocks 4 and 5). These are clocked from the USB clock. Unlike the G/L devices, clock-block 0 is not used or affected.

3.4 Timers

The XUD library allocates and uses four timers.

3.5 Memory

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

4 Basic Usage

IN THIS CHAPTER

This section outlines the basic usage of XUD. Basic use is termed to mean each endpoint runs in its own dedicated core. Multiple endpoints in a single core are possible, please see Advanced Usage.

4.1 XUD Core: `XUD_Manager()`

This is the main XUD task that interfaces with the USB transceiver. It performs power-signalling/handshaking on the USB bus, and passes packets to/from 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.

XUD_Manager()

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

```
int XUD_Manager(chanend c_epOut[],
                int noEpOut,
                chanend c_epIn[],
                int noEpIn,
                chanend ?c_sof,
                XUD_EpType epTypeTableOut[],
                XUD_EpType epTypeTableIn[],
                out port ?p_usb_rst,
                clock ?clk,
                unsigned rstMask,
                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.

`epTypeTableOut`
See `epTypeTableIn`.

`epTypeTableIn`
This and `epTypeTableOut` are two arrays indicating 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). 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.

`p_usb_rst` The port to used to connect to an external phy reset line. Should be `null` for U-Series.

`clk` The clock block to use for the p_usb_rst port - this should not be clock block 0. Should be `null` for U-Series.

`rstMask` The mask to use when taking an external phy into/out of reset

4.1.1 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 Status Reporting).

Note: endpoints can also be marked as disabled.

Endpoints that are not used will `NAK` any traffic from the host.

4.1.2 `PwrConfig`

The `PwrConfig` parameter to `XUD_Manager()` 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, `XUD_Manager()` 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.

4.2 Endpoint Communication with `XUD_Manager()`

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

XUD_ep

Typedef for endpoint identifiers.

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

XUD_InitEp()

Initialises an XUD_ep.

Type

```
XUD_ep XUD_InitEp(chanend c_ep)
```

Parameters

c_ep	Endpoint channel to be connected to the XUD library.

Returns

Endpoint identifier

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()` core. For developer convenience these calls are wrapped up by XC functions.

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

4.2.1 XUD_GetBuffer()

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. 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, for errors see Status Reporting.

4.2.2 XUD_SetBuffer()

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.

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, for errors see Status Reporting.

4.2.3 XUD_SetBuffer_EpMax()

This 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.

XUD_SetBuffer_EpMax()

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, for errors see Status Reporting.

4.2.4 XUD_DoGetRequest()

XUD_DoGetRequest()

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, for errors see Status Reporting

4.2.5 XUD_DoSetRequestStatus()

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 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, for errors see Status Reporting.

4.2.6 XUD_SetDevAddr()

XUD_SetDevAddr()

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.

4.2.7 XUD_SetStall()

XUD_SetStall()

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.

4.2.8 XUD_SetStallByAddr()

XUD_SetStallByAddr()

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.

4.2.9 XUD_ClearStall()

XUD_ClearStall()

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.

4.2.10 XUD_ClearStallByAddr()

XUD_ClearStallByAddr()

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.

4.2.11 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_GetData, XUD_SetData, etc.) return XUD_RES_RST if a USB bus reset is detected.

This reset notification is important if an endpoint core is expecting alternating INs and OUTs. For example, consider the case where an endpoint is always expecting the sequence OUT, IN, OUT (such as 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 bus-speed it is currently running at.

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

4.2.12 XUD_ResetEndpoint()

XUD_ResetEndpoint()

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.

4.3 SOF Channel

An application can pass a channel-end to the c_sof parameter of XUD_Manager(). 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_Manager() there must be a responsive task ready to receive SOF notifications otherwise the XUD_Manager() task will be blocked attempting to send these messages.

4.4 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 Figure 4.

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.

Figure 4: Supported Test Mode Selector Codes

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

4.4.1 `XUD_SetTestMode()`

XUD_SetTestMode()

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

5 Advanced Usage

IN THIS CHAPTER

Advanced usage is termed to mean the implementation of multiple endpoints in a single core as well as the addition of real-time processing to an endpoint core.

The functions documented in Basic Usage 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
- An `select` statement is used, along with a `select handler` to wait for, and capture, send/receive notifications from the `XUD_Manager` core.

The available `XUD_SetReady_` functions are listed below.

5.1 XUD_SetReady_Out()

XUD_SetReady_Out()

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, for errors see *Status Reporting*.

5.2 XUD_SetReady_In()

XUD_SetReady_In()

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, for errors see *Status Reporting*.

The following functions are also provided to ease integration with more complex buffering schemes than a single packet buffer. A example might be a circular-buffer for an audio stream.

5.3 XUD_SetReady_OutPtr()

XUD_SetReady_OutPtr()

Marks an OUT endpoint as ready to receive data.

Type

```
int XUD_SetReady_OutPtr(XUD_ep ep, unsigned addr)
```

Parameters

ep	The OUT endpoint identifier (created by XUD_InitEp).
addr	The address of 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, for errors see *Status Reporting*.

5.4 XUD_SetReady_InPtr()

XUD_SetReady_InPtr()

Marks an IN endpoint as ready to transmit data.

Type

```
XUD_Result_t XUD_SetReady_InPtr(XUD_ep ep, unsigned addr, int len)
```

Parameters

ep	The IN endpoint identifier (created by XUD_InitEp).
addr	The address of 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, for errors see *Status Reporting*.

Once an endpoint has been marked ready to send/receive by calling one of the above XUD_SetReady_ functions, an XC select statement can be used to handle notifications of a packet being sent/received from XUD_Manager(). These notifications are communicated via channels.

For convenience, select handler functions are provided to handle events in the select statement. These are documented below.

5.5 XUD_GetData_Select()

XUD_GetData_Select()

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, for errors see *Status Reporting*.

5.6 XUD_SetData_Select()

XUD_SetData_Select()

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, for errors see *Status Reporting*.

5.7 Example

A simple example of the functionality described in this section is shown below:

```
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_ep ep_in = XUD_InitEp(c_ep_in);

    /* 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;

        }
    }
}
//:
```

6 Further Reading

The following documents provide further reading regarding programming USB devices on XMOS platforms:

- XMOS USB Device Design Guide
- HID Class USB Device Quick Start Guide
- USB Custom Bulk Device Quick Start Guide

7 Document Version History

Version history for this document.

Figure 5: Version History

Date	Version	Comment
2014-05-16	2.1	Updates to related to enabling test modes
2014-03-18	2.0	Added XUD_Result_t to API documentation Added Further Reading section
2014-01-29	1.2	Minor updates. Added Advanced Usage and PwrConfig sections
2013-04-23	1.1	API updates and changes to Standard Request handling
2011-01-06	1.0	Updates for API changes
2010-07-22	1.0b	Beta Release