lib_xud: USB device library

Publication Date: 2024/10/25
Document Number: XM-012026-UG v2.4.0

IN THIS DOCUMENT

1 Introduction

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* XTC tool chain and XC language.

This library is for use with *xcore-200* series (XS2 architecture) or *xcore.ai* series (XS3 architecture) devices only, previous generations of *xcore* devices (i.e. XS1 architecture) are not supported.

`lib_xud` is intended to be used with the XCommon CMake , the *XMOS* application build and dependency management system.

2 Overview

xcore.ai devices and selected *xcore-200* devices include an integrated USB transceiver. `lib_xud` allows the implementation of both full-speed and high-speed USB 2.0 devices on these devices. The library provides an identical API for all devices.

The library performs all of 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 thread with endpoint and application tasks 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 Fig. 1. Circles represent threads running 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 it's attention. If not, the XUD library will *NAK*.

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

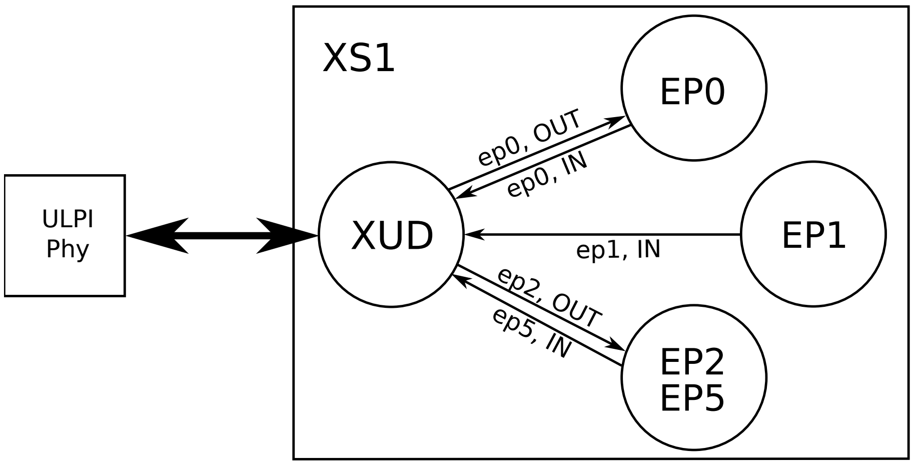

Fig. 1: XUD Overview

3 File Arrangement

The following list gives a brief description of the files that make up `lib_xud`:

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

lib/src/core
: Main logic for XUD functionality.

lib/src/user
: Definitions and source that define the client side interface.

4 Resource Usage

This section describes the resources required by `lib_xud`.

4.1 Ports/pins

xcore.ai series

The *xcore.ai* series of devices have an integrated USB transceiver. Some ports are used to communicate with the USB transceiver inside the *xcore.ai* series packages.

These ports/pins should not be used when USB functionality is enabled. The ports/pins are shown in Table 1.

Table 1: *xcore.ai* series required pin/port connections

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
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	
X0D23	P1H0				
X0D24	P1I0				
X0D25	P1IJ				
X0D34	P1K0				

xcore-200 series

Selected *xcore-200* series 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.

Table 2: *xcore-200* series required pin/port connections

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
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				
X0D25	P1IJ				
X0D34	P1K0				

4.2 Thread frequency

Due to I/O requirements, the `lib_xud` requires a guaranteed MIPS rate to ensure correct operation. This means that thread count restrictions must be observed. The XUD thread must run at at least 85 MIPS.

This means that for an *xcore* device running at 600 MHz there should be no more than seven cores executing at any time when using `lib_xud`.

xcore devices allow setting threads to "priority" mode. Priority threads are guaranteed 20% of the processor bandwidth. If XUD is assigned a priority core then up to eight cores may be used with the remaining seven getting (600 * 0.8) / 7 = 68.6MIPS each.

This restriction is only a requirement on the tile on which the `XUD_Main()` is running. For example, the other tile on an dual-tile device is unaffected by this restriction.

Note: At points of execution `XUD_Main()` will run in "fast mode", this is a requirement to meet timing.

4.3 Clock blocks

`lib_xud` uses two clock blocks, one for receive and one for transmit. Clocks blocks 4 and 5 are used for transmit and receive respectively. These clock blocks are configured such that they are clocked by the 60 MHz clock from the USB transceiver. The ports used by `lib_xud` are in turn clocked from these clock blocks.

4.4 Timers

`lib_xud` internally allocates and uses four timers.

4.5 Memory

`lib_xud` requires approximately 16 Kbytes of memory, of which around 15 Kbytes is code or initialised variables that must be stored in boot memory.

5 Basic usage

Basic use is termed to mean each endpoint runs in its own dedicated thread. Multiple endpoints in a single thread are possible, please see *Advanced usage*.

Operation is synchronous in nature: The endpoint tasks make calls to blocking functions and waits for the transfer to complete before proceeding.

5.1 XUD IO task

`XUD_Main()` is the main task that interfaces with the USB transceiver. It performs connection and handshaking on the USB bus as well as other bus-states such as suspend and resume. It also handles passing 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 (assuming a bus-powered device).

int **`XUD_Main`**(chanend c_epOut[], int noEpOut, chanend c_epIn[], int noEpIn, NULLABLE_RESOURCE(chanend, c_sof), XUD_EpType epTypeTableOut[], XUD_EpType epTypeTableIn[], XUD_BusSpeed_t desiredSpeed, XUD_PwrConfig pwrConfig)

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.

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.
- **`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.

Endpoint type tables

The endpoint type tables are arrays of type `XUD_EpType` and are used to inform `lib_xud` about the endpoints in use. This information is 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*). Two tables are required, one for IN and one for OUT endpoints.

Suitable values are provided in the `XUD_EpTransferType` *enum*:

- `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

OUT endpoint N will use index N of the output-endpoint-table, IN endpoint 0x8N will use index N of the inpout-endpoint-table. Endpoint 0 must exist in both tables.

..note:

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

`PwrConfig`

The `PwrConfig` parameter to `XUD_Main()` 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_Main()` monitors the *VBUS* input for a valid voltage and responds 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_BUS` can be used since it 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 and is not USB specification compliant.

5.2 VBUS monitoring

For self-powered devices it is important that `lib_xud` is aware of the *VBUS* state. This allows the device to disconnect its pull-up resistors from D+/D- and ensure the device does not have any voltage on the D+/D- pins when *VBUS* is not present. Compliance testing specifically checks for this in the *USB Back Voltage* test.

Warning: Failure to conform to this requirement will lead to an uncompliant device and likely lead to interoperability issues.

USB-enabled *xcore-200* series devices have a dedicated *VBUS* pin which should be wired up as per the data-sheet recommendations including over-voltage protection.

For increasing flexibility, *xcore.ai* series devices do not have a dedicated *VBUS* pin. A generic IO port/pin should be used for this purpose (with appropriate external circuitry - see data-sheet recommendations).

`lib_xud` makes a call to a function `XUD_HAL_GetVBusState()` that which should be implemented to match the target hardware. For example:

```
on tile[XUD_TILE]: in port p_vbus = XS1_PORT_1P;

unsigned int XUD_HAL_GetVBusState(void)
{
    unsigned vBus;
    p_vbus :> vBus;
    return vBus;
}
```

The function should return 1 if *VBUS* is present, otherwise 0. In the case of the example above, the validity of *VBUS* is directly represented by the value on port 1P, however, this may not be the case for all hardware implementations, it could be inverted or even require a read from an external IO expander, for example.

Note: *VBUS* need not be connected if the device is wholly powered by USB i.e. a bus powered device.

5.3 USB_TILE define

In order that `lib_xua` may instantiate resources on the correct tile (typically ports) it requires a `USB_TILE` define to be set. The default value for the define is `tile[0]` so a developer only needs to set this if `XUD_Main()` is executing on a tile other than 0.

There are two ways of setting this define, either in the application *CMakeLists.txt* for example:

```
set(APP_COMPILER_FLAGS -DXUD_TILE=tile[0])
```

Or, following *XMOS* software library convention, providing a *xud_conf.h* file in the application codebase. This header file will be automatically detected by the build system and used by `lib_xud`. Example content for this header file is as follows:

```
#indef _XUD_CONF_H_
#define _XUD_CONF_H_

#define XUD_TILE tile[0]

#endif
```

5.4 Data transfer

Communication state between an endpoint client task and the XUD IO task is encapsulated in an opaque type:

typedef unsigned int **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 task with the following call that takes as an argument the endpoint channel connected to the XUD library:

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

Initialises an XUD_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_SetBuffer()`, `XUD_GetBuffer()` and `XUD_GetSetupBuffer()`.

These functions implement the low-level shared memory/channel communication with the `XUD_Main()` task.

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

`XUD_SetBuffer()`

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

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.

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`_.

`XUD_GetBuffer()`

XUD_Result_t **`XUD_GetBuffer`**(*XUD_ep* ep_out, unsigned char buffer[], REFERENCE_PARAM(unsigned, length))

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.

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`_.

`XUD_GetSetupBuffer()`

XUD_Result_t **`XUD_GetSetupBuffer`**(*XUD_ep* ep_out, unsigned char buffer[], REFERENCE_PARAM(unsigned, length))

Request setup data from usb buffer for a specific endpoint, pauses until data is available.

Parameters

- **`ep_out`** – The OUT endpoint identifier (created by `XUD_InitEP`).
- **`buffer`** – A char buffer passed by ref into which data is returned.
- **`length`** – Length of the buffer received (expect 8 bytes)

Returns

XUD_RES_OKAY on success, for errors see `Status Reporting`_.

For user convenience these functions are wrapped up in functions that match commonly required packet sequences:

`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_Result_t **`XUD_SetBuffer_EpMax`**(*XUD_ep* ep_in, unsigned char buffer[], unsigned datalength, unsigned 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.

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`_.

`XUD_DoGetRequest()`

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

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.

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_`

`XUD_DoSetRequestStatus()`

XUD_Result_t **`XUD_DoSetRequestStatus`**(*XUD_ep* ep_in)

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.

Parameters

- **`ep_in`** – The Endpoint 0 IN identifier to the XUD manager.

Returns

XUD_RES_OKAY on success, for errors see `Status Reporting_`.

5.5 Data transfer example

A simple endpoint task is shown below demonstrating basic data transfer to the host.

```
void ExampleEndpoint(chanend c_ep_in)
{
    char buffer[512];

    XUD_ep ep_to_host = XUD_InitEp(chan_ep_in);

    while(1)
    {
        XUD_SetBuffer(ep_to_host, buffer, 512);
    }
}
```

5.6 Status reporting

An endpoint can register for "status reporting" such that bus state can be 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 therefore bus-speed changes). The `lib_xud` access functions discussed previously (`XUD_GetBuffer`, `XUD_SetBuffer`, etc) return `XUD_RES_RST` if a USB bus reset is detected.

This reset notification is important if an endpoint task is expecting alternating IN and OUT transactions. For example, consider the case where an endpoint is always expecting the sequence OUT, IN, OUT (such as a control transfer or a request response protocol). If an unplug/reset event was received after the first OUT, the host would return to sending the initial OUT after a re-plug, whilst the endpoint task would hang trying to send a response the IN. The endpoint needs to know of the bus reset in order to reset its state machine.

Note: Endpoint 0 *requires* this functionality to be enabled since it deals with bi-directional control transfers

This functionality is also important for high-speed devices, since it is not guaranteed that a host will enumerate the device as a high-speed device, say if it's plugged via full-speed hub.

The device typically 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 as a `XUD_BusSpeed_t` with the value `XUD_SPEED_FS` ;or `XUD_SPEED_HS`.

`XUD_ResetEndpoint()`

XUD_BusSpeed_t **`XUD_ResetEndpoint`**(*XUD_ep* one,
NULLABLE_REFERENCE_PARAM(*XUD_ep*, two))

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.

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, `XUD_SPEED_FS` - the device is running at full speed, or `XUD_SPEED_KILL` to indicate that the USB stack has been shut down by another part of the user code (using XUD_Kill). If the last value is returned, the endpoint code should call XUD_CloseEndpoint and then terminate.

5.7 Status reporting example

A simple endpoint task is shown below demonstrating basic data transfer to the host and bus status inspection.

```
void ExampleEndpoint(chanend c_ep_in)
{
    char buffer[512];
    XUD_Result_t result;

    XUD_ep ep_to_host = XUD_InitEp(chan_ep_to_host);

    while(1)
    {
        if((result = XUD_SetBuffer(ep_to_host, buffer, 512)) == XUD_RES_RST)
        {
            XUD_ResetEndpoint(ep_from_host, ep_to_host);
        }
    }
}
```

5.8 SOF channel

An application can pass an optional channel-end to the `c_sof` parameter of `XUD_Main()`. This will cause a word of data to be output every time the device receives a SOF (*Start Of Frame*) packet from the host. This can be used for timing information in audio devices etc.

If this functionality is not required `null` should be passed as the parameter.

Note: If an optional channel-end is passed into `XUD_Main()` there must be a responsive task ready to receive SOF notifications otherwise the `XUD_Main()` task will be blocked attempting to send these messages leading to it being unresponsive to the host.

5.9 Halting

The USB specification requires the ability for an endpoint to send a *STALL* response to the host if an endpoint is halted, or if control pipe request is not supported. `lib_xud` provides various functions to support this. In some cases it is convenient to use the `XUD_ep` whilst in other cases it is easier to use the endpoint address. Functions to use either are provided.

`XUD_SetStall()`

void **`XUD_SetStall`**(*XUD_ep* ep)

Mark an endpoint as STALLed. It is cleared automatically if a SETUP received on the endpoint.

Warning: Must be run on same tile as XUD core

Parameters

- **ep** – XUD_ep type.

`XUD_SetStallByAddr()`

void **`XUD_SetStallByAddr`**(int epNum)

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.

Warning: Must be run on same tile as XUD core

Parameters

- **epNum** – Endpoint number.

`XUD_ClearStall()`

void **`XUD_ClearStall`**(*XUD_ep* ep)

Mark an endpoint as NOT STALLed.

Warning: Must be run on same tile as XUD core

Parameters

- **ep** – XUD_ep type.

`XUD_ClearStallByAddr()`

void **`XUD_ClearStallByAddr`**(int epNum)

Mark an endpoint as NOT STALLed based on its EP address. Note: the IN bit of the endpoint address is used.

Warning: Must be run on same tile as XUD core

Parameters

- **epNum** – Endpoint number.

5.10 USB test modes

`lib_xud` supports the required test modes for USB Compliance testing.

`lib_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 3.

Table 3: Supported *Test Mode Selector Codes*

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

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 selected test mode.

`XUD_SetTestMode()`

void **`XUD_SetTestMode`**(*XUD_ep* ep, unsigned testMode)

Enable a specific USB test mode in XUD.

Warning: Must be run on same tile as XUD core

Parameters

- **`ep`** – XUD_ep type (must be endpoint 0 in or out)
- **`testMode`** – The desired test-mode

6 Control Endpoints

`lib_xud` provides helper functions that provide a set of standard functionality to aid the creation of USB devices.

Control transfers are typically used for command and status operations. They are essential to set up a USB device with all enumeration functions being performed using Control transfers. Control transfers are characterised by the use of a *SETUP* transaction.

USB devices must provide an implementation of a control endpoint at endpoint 0. Endpoint 0 must deal with enumeration and configuration requests from the host. Many enumeration requests are compulsory and common to all devices, with most of them being requests for mandatory descriptors (Configuration, Device, String, etc).

Since these requests are common across most (if not all) devices, some useful functions are provided to deal with them.

6.1 Helper Functions

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

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

A `USB_StandardRequests()` function provides a bare-minimum implementation of the mandatory requests required to be implemented by a USB device. The function inspects this `USB_SetupPacket_t` structure and includes a minimum implementation of the Standard Device requests. The rest of this section documents the requests handled and lists the basic functionality associated with the request.

It is not intended that this replace a good knowledge of the requests required, since `USB_StandardRequests()` has no knowledge about the specific class that is implemented, and hence does not guarantee a fully USB compliant device.

Each request could well be required to be over-ridden for a device implementation. For example,a USB Audio device could well require a specialised version of *SET_INTERFACE* since this could mean that audio streaming will commence imminently.

The `USB_StandardRequests()` function takes as parameters arrays representing the device descriptor, configuration descriptor, and a string table as well as a `USB_SetupPacket_t` and the current bus-speed.

Note: `USB_StandardRequests()` takes two references for device and configuration descriptors - this allows for different functionality based on bus-speed. `USB_StandardRequests()` forms valid *Device Qualifier* and *Other Speed Configuration* descriptors from these arrays.

USB_SetupPacket_t

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

```
typedef struct USB_SetupPacket
{
```

(continues on next page)

(continued from previous page)

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

USB_GetSetupPacket()

XUD_Result_t **USB_GetSetupPacket**(*XUD_ep* ep_out, *XUD_ep* ep_in, REFERENCE_PARAM(USB_SetupPacket_t, sp))

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

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

USB_StandardRequests()

This function takes a populated `USB_SetupPacket_t` structure as an argument.

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

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

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.

This section now details the actual requests handled by this function. If parsing the request does not result in a match, the request is not handled, the Endpoint is marked "Halted" (Using `XUD_SetStall_Out()` and `XUD_SetStall_In()`) and the function returns `XUD_RES_ERR`. The function returns `XUD_RES_OKAY` if a request was handled without error (See also *Status reporting example*).

`USB_StandardRequests()`: Standard Device Requests

The `USB_StandardRequests()` function handles the following *Standard Device Requests*:

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

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

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

USB_StandardRequests(): Standard Interface Requests

The USB_StandardRequests() function handles the following Standard Interface Requests:

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

USB_StandardRequests(): Standard Endpoint Requests

The USB_StandardRequests() function handles the following Standard Endpoint Requests:

- SET_FEATURE
- CLEAR_FEATURE
- GET_STATUS

6.2 Control Endpoint Example

The code listing below shows a simple example of a endpoint 0 implementation showing the use of USB_SetupPacket_t, USB_SetSetupPacket() and USBStandardRequests():

```
void Endpoint0(chanend c_ep0_out, chanend c_ep0_in)
{
    USB_SetupPacket_t sp;
    XUD_BusSpeed_t usbBusSpeed;
    XUD_ep ep0_out = XUD_InitEp(c_ep0_out);
    XUD_ep ep0_in  = XUD_InitEp(c_ep0_in);

    while(1)
    {
        XUD_Result_t result = USB_GetSetupPacket(ep0_out, ep0_in, sp);

        if(result == XUD_RES_OKAY)
        {
            result = USB_StandardRequests(ep0_out, ep0_in,
                        devDesc_HS, sizeof(devDesc_HS),
                        cfgDesc_HS, sizeof(cfgDesc_HS),
                        devDesc_FS, sizeof(devDesc_FS),
                        cfgDesc_FS, sizeof(cfgDesc_FS),
                        stringTable, sizeof(stringTable),
                        sp, usbBusSpeed);
        }

        /* USB bus reset detected, reset EP and get new bus speed */
        if(result == XUD_RES_RST)
        {
            usbBusSpeed = XUD_ResetEndpoint(ep0_out, ep0_in);
        }
    }
}
```

Note: For conciseness the declarations of the arrays representing the device and configuration descriptors and the string table are not shown.

.. _sec_programming:

7 Programming guide

This section provides information on how to create an basic application using `lib_xud`.

7.1 Includes

The application needs to include the header file *xud.h*.

7.2 Declarations

Arrays representinge end endpoint types for both IN and OUT endpoints should be declared. These must each include one for endpoint 0, for example:

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

The endpoint types are:

- `XUD_EPTYPE_ISO`: Isochronous endpoint
- `XUD_EPTYPE_INT`: Interrupt endpoint
- `XUD_EPTYPE_BUL`: Bulk endpoint
- `XUD_EPTYPE_CTL`: Control endpoint
- `XUD_EPTYPE_DIS`: Disabled endpoint

`XUD_STATUS_ENABLE` is ORed in to the endpoints that wish to be informed of USB bus resets (see *Status reporting example*).

7.3 `main()`

Within the `main()` function it is necessary to allocate the channels to connect the endpoints and then create the top-level par containing calls to `XUD_Main()`, an endpoint 0 task and any application specific endpoint tasks, for example:

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

        Endpoint0(c_ep_out[0], c_ep_in[0]);

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

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

7.4 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.

7.5 Sending and receiving data

An application specific endpoint can send data using `XUD_SetBuffer()` and receive data using `XUD_GetBuffer()` etc as described in *Basic usage*.

7.6 Endpoint 0 implementation

It is necessary to create an implementation for endpoint 0 which takes two channels, one for IN and one for OUT. It can take an optional channel for *test* (see *USB test modes*):

A typical prototype might for such a funciton might look like the following:

```
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 should like the following:

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

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

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

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

                    ...
                }

                break;
            ...
        }

        result = USB_StandardRequests(ep0_out , 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: On bus reset the `XUD_ResetEndpoint()` function returns the negotiated USB speed (i.e. full or high speed).

7.7 Device descriptors

Every USB device must provide a set of descriptors. 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 specifications.

It is beyond the scope of this document to give details of writing a descriptor, please see the relevant USB Specification Documents.

8 Example application

This section contains a fully worked example of implementing a USB mouse device compliant to the Human Interface Device (HID) Class mouse device.

The application operates as a simple mouse which when running moves the mouse pointer on the host machine. This demonstrates the simple way in which PC peripheral devices can easily be deployed using an *xcore* device.

Note: This application note provides a standard USB HID class device and as a result does not require drivers to run on Windows, macOS or Linux.

The full source for this application is provided along side the `lib_xud` software download in the *examples/app_hid_house* directory.

Note: The example code provides implementations in C and XC. This section concentrates on the XC version.

8.1 Required hardware

This application note is designed to run on *XMOS xcore-200* or *xcore.ai* series devices.

The example code provided has been implemented and tested on the *XK-EVK-XU316* board but there is no dependency on this board and it can be modified to run on any development board which uses an *xcore-200* or *xcore.ai* series device.

8.2 Declarations

```
#include "xud_device.h"
#include "hid_descs.h"

/* Number of Endpoints used by this app */
#define EP_COUNT_OUT   1
#define EP_COUNT_IN    2

/* Endpoint type tables - informs XUD what the transfer types for each Endpoint in use and also
 * if the endpoint wishes to be informed of USB bus resets
 */
XUD_EpType epTypeTableOut[EP_COUNT_OUT] = {XUD_EPTYPE_CTL | XUD_STATUS_ENABLE};
XUD_EpType epTypeTableIn[EP_COUNT_IN] =   {XUD_EPTYPE_CTL | XUD_STATUS_ENABLE, XUD_EPTYPE_BUL};
```

8.3 `main()`

The `main()` function creates three tasks: the XUD Io task, endpoint 0, and a task for handling the HID endpoint. An array of channels is used for both IN and OUT endpoints, endpoint 0 requires both, the HID task simply implements an IN endpoint sending mouse data to the host.

```
int main()
{
    chan c_ep_out[EP_COUNT_OUT];
    chan c_ep_in[EP_COUNT_IN];

    par
    {
        on tile[0]: XUD_Main(c_ep_out, EP_COUNT_OUT, c_ep_in, EP_COUNT_IN, null,
                             epTypeTableOut, epTypeTableIn, XUD_SPEED_HS, XUD_PWR_BUS);

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

        on tile[0]: hid_mouse(c_ep_in[1]);
    }
```

(continues on next page)

(continued from previous page)

```
    return 0;
} //
```

Since this example does not require *SOF* notifications `null` is passed into the `c_sof` parameter. `XUD_SPEED_HS` is passed for the `desiredSpeed` parameter such that the device attempts to run as a high-speed device.

8.4 HID endpoint task

This function responds to the HID requests - it moves the mouse cursor in square by moving 40 pixels in each direction in sequence every 100 requests using a basic state-machine. This function could be easily changed to feed other data back (for example based on user input).

```
void hid_mouse(chanend chan_ep_hid)
{
    int counter = 0;
    enum {RIGHT, DOWN, LEFT, UP} state = RIGHT;

    XUD_ep ep_hid = XUD_InitEp(chan_ep_hid);

    for(;;)
    {
        /* Move the pointer around in a square (relative) */
        if(counter++ >= 500)
        {
            int x;
            int y;

            switch(state)
            {
                case RIGHT:
                    x = 40;
                    y = 0;
                    state = DOWN;
                    break;

                case DOWN:
                    x = 0;
                    y = 40;
                    state = LEFT;
                    break;

                case LEFT:
                    x = -40;
                    y = 0;
                    state = UP;
                    break;

                case UP:
                default:
                    x = 0;
                    y = -40;
                    state = RIGHT;
                    break;
            }

            /* Unsafe region so we can use shared memory. */
            unsafe
            {
                /* global buffer 'g_reportBuffer' defined in hid_defs.h */
                char * unsafe p_reportBuffer = g_reportBuffer;

                p_reportBuffer[1] = x;
                p_reportBuffer[2] = y;

                /* Send the buffer to the host.  Note this will return when complete */
                XUD_SetBuffer(ep_hid, (char *) p_reportBuffer, sizeof(g_reportBuffer));
                counter = 0;
            }
        }
    }
} //
```

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

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

8.5 Device descriptors

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

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

Device descriptor

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

```
static unsigned char devDesc[] =
{
    0x12,                  /* 0  bLength */
    USB_DESCTYPE_DEVICE,   /* 1  bdescriptorType */
    0x00,                  /* 2  bcdUSB */
    0x02,                  /* 3  bcdUSB */
    0x00,                  /* 4  bDeviceClass */
    0x00,                  /* 5  bDeviceSubClass */
    0x00,                  /* 6  bDeviceProtocol */
    0x40,                  /* 7  bMaxPacketSize */
    (VENDOR_ID & 0xFF),    /* 8  idVendor */
    (VENDOR_ID >> 8),      /* 9  idVendor */
    (PRODUCT_ID & 0xFF),   /* 10 idProduct */
    (PRODUCT_ID >> 8),     /* 11 idProduct */
    (BCD_DEVICE & 0xFF),   /* 12 bcdDevice */
    (BCD_DEVICE >> 8),     /* 13 bcdDevice */
    0x01,                  /* 14 iManufacturer */
    0x02,                  /* 15 iProduct */
    0x00,                  /* 16 iSerialNumber */
    0x01                   /* 17 bNumConfigurations */
};
```

Device qualifier descriptor

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

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

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

Configuration descriptor

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

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

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

    0x09,                     /* 0  bLength. Note this is replicated in hidDescriptor[] */
    0x21,                     /* 1  bDescriptorType (HID) */
    0x11,                     /* 2  bcdHID */
    0x01,                     /* 3  bcdHID */
    0x00,                     /* 4  bCountryCode */
    0x01,                     /* 5  bNumDescriptors */
    0x22,                     /* 6  bDescriptorType[0] (Report) */
    0x32,                     /* 7  wDescriptorLength */
    0x00,                     /* 8  wDescriptorLength */

    0x07,                     /* 0  bLength */
    0x05,                     /* 1  bDescriptorType */
    0x81,                     /* 2  bEndpointAddress */
    0x03,                     /* 3  bmAttributes */
    0x40,                     /* 4  wMaxPacketSize */
    0x00,                     /* 5  wMaxPacketSize */
    0x0a                      /* 6  bInterval */
};
```

Other Speed Configuration descriptor

An Other Speed Configuration descriptor is used for similar reasons as the Device Qualifier descriptor. The `USB_StandardRequests()` function generates this descriptor from the Configuration descriptors passed to it based on the bus speed it is currently running at. For the HID mouse example the same Configuration descriptors are uses regardless of bus-speed (i.e. full-speed or high-speed).

String descriptors

An array of strings supplies all the strings that are referenced from the descriptors (using fields such as 'iInterface', 'iProduct' etc.). The string at index 0 must always contain the *Language ID Descriptor*. This descriptor indicates the languages that the device supports for string descriptors.

The `USB_StandardRequests()` function deals with requests for strings using the table of strings passed to it. It handles the conversion of the raw strings to valid USB string descriptors.

The string table for the HID mouse example is shown below:

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

8.6 Application and class specific requests

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

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

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

See the HID Class Specification and related documentation for full details of all HID requests.

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

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

```
static unsigned char hidReportDescriptor[] =
{
    0x05, 0x01,    /* Usage Page (Generic Desktop) */
    0x09, 0x02,    /* Usage (Mouse) */
    0xA1, 0x01,    /* Collection (Application) */
    0x09, 0x01,    /* Usage (Pointer) */
    0xA1, 0x00,    /* Collection (Physical) */
    0x05, 0x09,    /* Usage Page (Buttons) */
    0x19, 0x01,    /* Usage Minimum (01) */
    0x29, 0x03,    /* Usage Maximum (03) */
    0x15, 0x00,    /* Logical Minimum (0) */
    0x25, 0x01,    /* Logical Maximum (1) */
    0x95, 0x03,    /* Report Count (3) */
    0x75, 0x01,    /* Report Size (1) */
    0x81, 0x02,    /* Input (Data,Variable,Absolute); 3 button bits */
    0x95, 0x01,    /* Report Count (1) */
    0x75, 0x05,    /* Report Size (5) */
    0x81, 0x01,    /* Input(Constant); 5 bit padding */
    0x05, 0x01,    /* Usage Page (Generic Desktop) */
    0x09, 0x30,    /* Usage (X) */
    0x09, 0x31,    /* Usage (Y) */
    0x15, 0x81,    /* Logical Minimum (-127) */
    0x25, 0x7F,    /* Logical Maximum (127) */
    0x75, 0x08,    /* Report Size (8) */
    0x95, 0x02,    /* Report Count (2) */
    0x81, 0x06,    /* Input (Data,Variable,Relative); 2 position bytes (X & Y) */
    0xC0,          /* End Collection */
    0xC0           /* End Collection */
};
```

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

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

```
void Endpoint0(chanend chan_ep0_out, chanend chan_ep0_in)
{
    USB_SetupPacket_t sp;

    unsigned bmRequestType;
    XUD_BusSpeed_t usbBusSpeed;

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

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

        if(result == XUD_RES_OKAY)
        {
            /* Set result to ERR, we expect it to get set to OKAY if a request is handled */
            result = XUD_RES_ERR;

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

(continues on next page)

(continued from previous page)

```
            if ((bmRequestType == USB_BMREQ_H2D_STANDARD_DEV) &&
                (sp.bRequest == USB_SET_ADDRESS))
            {
              // Host has set device address, value contained in sp.wValue
            }

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

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

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

                                case HID_REPORT:
                                    result = XUD_DoGetRequest(ep0_out, ep0_in, hidReportDescriptor,
↪sizeof(hidReportDescriptor), sp.wLength);
                                    break;
                            }
                        }
                    }
                    break;

                /* Direction: Device-to-host and Host-to-device
                 * Type: Class
                 * Recipient: Interface
                 */
                case USB_BMREQ_H2D_CLASS_INT:
                case USB_BMREQ_D2H_CLASS_INT:

                    /* Inspect for HID interface num */
                    if(sp.wIndex == 0)
                    {
                        /* Returns  XUD_RES_OKAY if handled,
                         *          XUD_RES_ERR if not handled,
                         *          XUD_RES_RST for bus reset */
                        result = HidInterfaceClassRequests(ep0_out, ep0_in, sp);
                    }
                    break;
            }
        }

        /* If we haven't handled the request about then do standard enumeration requests */
        if(result == XUD_RES_ERR )
        {
            /* Returns  XUD_RES_OKAY if handled okay,
             *          XUD_RES_ERR if request was not handled (STALLed),
             *          XUD_RES_RST for USB Reset */
             unsafe{
            result = USB_StandardRequests(ep0_out, ep0_in, devDesc,
                        sizeof(devDesc), cfgDesc, sizeof(cfgDesc),
                        null, 0, null, 0, stringDescriptors, sizeof(stringDescriptors)/
↪sizeof(stringDescriptors[0]),
                        sp, usbBusSpeed);
             }
        }

        /* USB bus reset detected, reset EP and get new bus speed */
        if(result == XUD_RES_RST)
        {
            usbBusSpeed = XUD_ResetEndpoint(ep0_out, ep0_in);
        }
    }
} /* Endpoint0 */
```

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

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

    switch(sp.bRequest)
    {
```

(continues on next page)

(continued from previous page)

```
        case HID_GET_REPORT:

            /* Mandatory. Allows sending of report over control pipe */
            /* Send a hid report - note the use of unsafe due to shared mem */
            unsafe {
              char * unsafe p_reportBuffer = g_reportBuffer;
              buffer[0] = p_reportBuffer[0];
            }

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

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

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

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

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

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

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

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

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

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

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

    return XUD_RES_ERR;
} /* HidInterfaceClassRequests */
```

If the HID request is not handled, the function returns `XUD_RES_ERR`. This results in `USB_StandardRequests()` being called, and eventually the endpoint responding to the host with a *STALL* to indicate an unsupported request.

9 Advanced usage

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

The functions documented in *Basic usage* such as `XUD_SetBuffer()` and `XUD_GetBuffer()` are synchronous in nature - they 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 thread efficiently (or at all, depending on the protocols involved).

To solve this `lib_xud` provides an API that is asynchronous in nature with functions that allow the separation of requesting to send/receive a packet and the notification of a successful transfer. This API utilises *xcore* events by using the *XC* `select` statement language feature.

General operation is as follows:

- A `XUD_SetReady_` function is called to mark an endpoint as ready to send or receive data
- A"select" statement is used, along with a `select handler` to wait for, and capture, send/receive notifications from the `XUD_Main` task

9.1 Function details

The available `XUD_SetReady_` functions for the asynchronous API are listed below.

`XUD_SetReady_Out()`

int **`XUD_SetReady_Out`**(*XUD_ep* ep, unsigned char buffer[])

Marks an OUT endpoint as ready to receive data.

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`.

`XUD_SetReady_In()`

static inline XUD_Result_t **`XUD_SetReady_In`**(*XUD_ep* ep, unsigned char buffer[], int len)

Marks an IN endpoint as ready to transmit data.

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.

XUD_SetReady_OutPtr()

static inline XUD_Result_t **XUD_SetReady_OutPtr**(*XUD_ep* ep, unsigned addr)

Marks an OUT endpoint as ready to receive data.

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`.

XUD_SetReady_InPtr()

static inline XUD_Result_t **XUD_SetReady_InPtr**(*XUD_ep* ep, unsigned addr, int len)

Marks an IN endpoint as ready to transmit data.

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_Main()`. These notifications are communicated via channels.

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

XUD_GetData_Select()

void **XUD_GetData_Select**(chanend c, *XUD_ep* ep, REFERENCE_PARAM(unsigned, length), REFERENCE_PARAM(XUD_Result_t, result))

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

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 (that was passed into XUD_SetReady_Out())
- **result** – XUD_Result_t passed by reference. XUD_RES_OKAY on success, for errors see `Status Reporting`.

XUD_SetData_Select()

void **XUD_SetData_Select**(chanend c, *XUD_ep* ep, REFERENCE_PARAM(XUD_Result_t, result))

Select handler function for transmitting IN endpoint data in a select.

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`.

Warning: It is currently not possible to share control endpoint (i.e. endpoint 0) functionalty with other endpoints/tasks. This is because a control endpoint **must** remain responsive to the host.

9.2 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 received */

                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;

        }
    }
}
```

10 Further Reading

- *XMOS* XTC Tools Installation Guide
  https://xmos.com/xtc-install-guide
- *XMOS* XTC Tools User Guide
  https://www.xmos.com/view/Tools-15-Documentation
- *XMOS* application build and dependency management system; *xcommon-cmake*
  https://www.xmos.com/file/xcommon-cmake-documentation/?version=latest
- USB Made Simple
  https://www.usbmadesimple.co.uk/index.html
- USB Specification 2.0
  https://www.usb.org/sites/default/files/usb_20_20240604.zip
- Human Interface Device (HID) Class Specification 1.11
  https://usb.org/document-library/device-class-definition-hid-111>
- HID Descriptor Tool
  https://usb.org/document-library/hid-descriptor-tool