



lib_xua: USB Audio components library

Publication Date: 2026/3/31

Document Number: XM-012296-UG v5.4.0

IN THIS DOCUMENT

1	Overview	4
2	Software architecture	6
3	Basic Usage	9
3.1	Library Structure	9
3.2	Using in an application	9
3.3	"Codeless" programming model	9
3.4	Configuring lib_xua	10
3.5	User functions	10
3.6	Extending the "Codeless" application	10
4	Options	12
4.1	Strings and IDs	12
4.2	Code Location	13
4.3	Channel counts and sample rates	14
4.4	USB Audio Class version	15
4.5	Synchronisation	18
4.6	I ² S/TDM	20
4.7	S/PDIF transmit	21
4.8	S/PDIF receive	22
4.9	ADAT transmit	23
4.10	ADAT receive	24
4.11	MIDI	25
4.12	PDM microphones	26
4.13	Mixer	27
4.14	Direct Stream Digital (DSD)	28
4.15	DFU	30
4.16	Audio stream formats	31
4.17	Other options	34
5	Advanced usage	35
5.1	Core hardware resources	35
5.2	Running the core components	37
5.3	I ² S/TDM	38
5.4	Mixer	39
6	Additional features	40
6.1	S/PDIF transmit	40
6.2	S/PDIF receive	41
6.3	ADAT transmit	41
6.4	ADAT receive	45
7	Implementation detail	47
7.1	Audio Hub and I ² S	48
7.2	Endpoint 0: Management and control	51
7.3	Audio endpoints (Endpoint Buffer and Decoupler)	53
7.4	XMOS USB Device (XUD) library	57
7.5	External clock recovery (Clock Gen)	58
7.6	Digital mixer	59
7.7	S/PDIF transmit	66
7.8	S/PDIF receive	67
7.9	MIDI	70
7.10	PDM microphones	71
7.11	Audio controls via Human Interface Device (HID)	73
7.12	Device Firmware Upgrade (DFU) over USB	74
7.13	Resource usage	79
8	API reference	80
8.1	Configuration defines	80
8.2	User function definitions	89
8.3	Component API	91





1 Overview

lib_xua contains shared components for use in the *XMOS* USB Audio (XUA) Reference Designs.

These components enable the development of USB Audio devices on the *XMOS* xcore architecture.

This document describes the structure of **lib_xua**, its use and resources required. It also covers some implementation detail.

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.

Functionality

Provides USB interface to audio I/O.

Supported Standards

USB	USB 2.0 (Full-speed and High-speed)
	USB Audio Class 1.0
	USB Audio Class 2.0
	USB Firmware Upgrade (DFU) 1.1
	USB Midi Device Class 1.0
Audio	I2S/TDM (16/32-bit)
	S/PDIF
	ADAT
	Direct Stream Digital (DSD)
	PDM Microphones
	MIDI

Supported Sample Frequencies

44.1kHz, 48kHz, 88.2kHz, 96kHz, 176.4kHz, 192kHz, 352.8kHz, 384kHz

Supported Devices

XMOS Devices	<i>xcore-200</i> Series
	<i>xcore.ai</i> Series



Requirements

Development Tools	XMOS XTC Development Tools (see README for version)
USB	<i>xcore</i> device with integrated USB phy (external phy not supported)
Audio	External audio DAC/ADC/CODECs (and required supporting componentry) supporting I2S/TDM
Boot/Storage	Compatible SPI/QSPI Flash device (or <i>xcore</i> device with internal flash)

Licensing and Support

Reference code provided without charge under license from XMOS.

Please visit <http://www.xmos.com/support> for support.

Reference code is maintained by XMOS Limited.



2 Software architecture

This section describes the required software architecture of a USB Audio device implemented using *lib_xua*, its dependencies and other supporting libraries.

lib_xua provides fundamental building blocks for producing USB Audio products on XMOS devices. Every system is required to have the components from *lib_xua* listed in [Table 1](#).

Table 1: Required XUA Components

Component	Description
Endpoint 0	Provides the logic for Endpoint 0 which handles enumeration and control of the device including DFU related requests.
Endpoint buffer	Buffers endpoint data packets to and from the host. Manages delivery of audio packets between the endpoint buffer component and the audio components. It can also handle volume control processing. Note, this currently utilises two cores
AudioHub	Handles audio I/O over I2S and manages audio data to/from other digital audio I/O components.

In addition low-level USB I/O is required and is provided by the external dependency [lib_xud](#).

Table 2: Required external components

Component	Description
XMOS USB Device Driver (XUD)	Handles the low level USB I/O.

In addition [Table 3](#) shows optional components that can be added/enabled from within *lib_xua*

Table 3: Optional Components

Component	Description
Mixer	Allows digital mixing of input and output channels. It can also handle volume control instead of the decoupler.
Clockgen	Drives an external frequency generator (PLL) and manages changes between internal clocks and external clocks arising from digital input. On xcore.ai Clockgen may also work in conjunction with lib_sw_pll to produce a local clock from the XCORE which is locked to the incoming digital stream.
MIDI	Outputs and inputs MIDI over a serial UART interface.



lib_xua also provides optional support for integrating with the following external dependencies listed in Table 4

Table 4: Optional external components

Component	Description
S/PDIF Transmitter (lib_spdif)	Outputs samples on an S/PDIF digital audio interface.
S/PDIF Receiver (lib_spdif < www.xmos.com/file/lib_spdif >)	Inputs samples off an S/PDIF digital audio interface (requires the clockgen component).
ADAT Transmitter (lib_adat)	Outputs samples on an ADAT digital audio interface.
ADAT Receiver (lib_adat)	Inputs samples off an ADAT digital audio interface (requires the clockgen component).
PDM Microphones (lib_mic_array)	Receives PDM data from microphones and performs PDM to PCM conversion

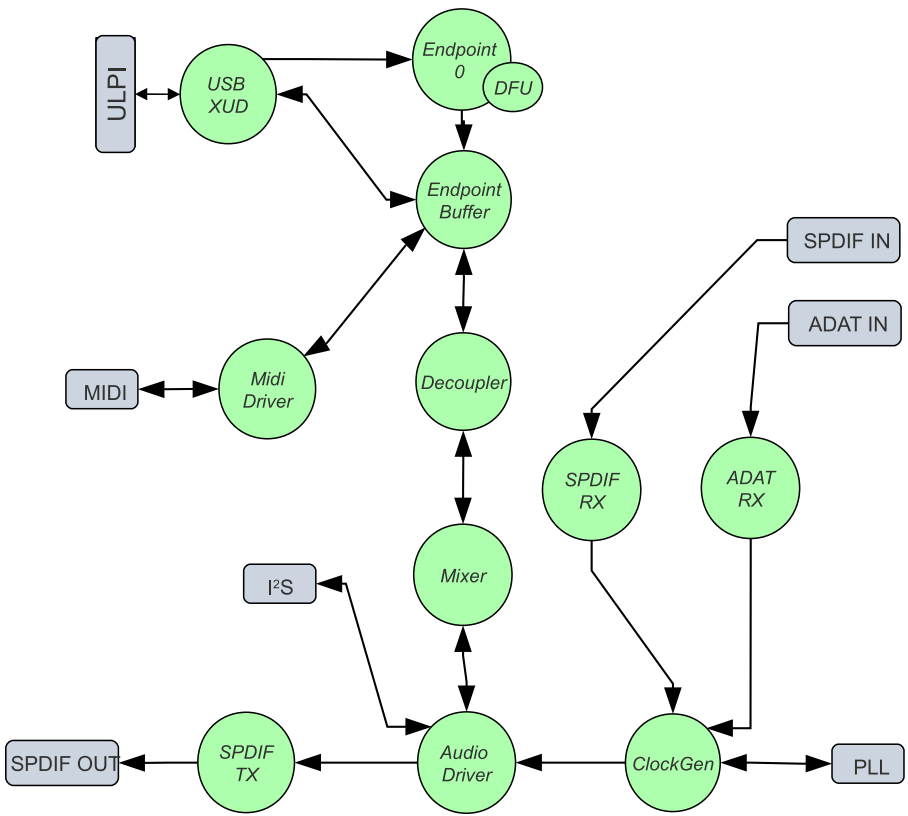


Fig. 1: USB Audio thread diagram

Fig. 1 shows how the components interact with each other in a typical system. The green circles represent threads with arrows indicating inter-thread communications.





3 Basic Usage

This section describes the basic usage of *lib_xua* and provides a guide on how to program USB Audio devices.

3.1 Library Structure

The code is split into several directories.

Table 5: **lib_xua** structure

core	Common code for USB audio applications
midi	MIDI I/O code
dfu	Device Firmware Upgrade code
hid	Human Interface Device code

Note, the **midi** and **dfu** directories are potential candidates for separate libs in their own right.

3.2 Using in an application

lib_xua is intended to be used with [XCommon CMake](#), the XMOS application build and dependency management system.

To use **lib_xua** in an application, add **lib_xua**, to the list of dependent modules in the application's *CMakeLists.txt* file.

```
set(APP_DEPENDENT_MODULES "lib_xua")
```

All *lib_xua* functions can be accessed via the **xua.h** header file:

```
#include <xua.h>
```

3.3 “Codeless” programming model

While it is possible to build a USB Audio device from the low-level components provided by **lib_xua**, this approach may not be suitable for all customers or product types.

For instance, some users may not have a large software development experience and simply want to customise some basic settings such as strings, sample-rates, channel-counts etc. Others may want to fully customise the implementation - adding additional functionality such as integrating DSP or possibly only using a subset of the functions provided - just **XUA_AudioHub**, for example.

In addition, the large number of supported features can lead to a large number of tasks, hardware resources, communication channels etc, requiring quite a lot of code to be authored for each product.

In order to cater for the former class of users, a “codeless” option is provided. Put simply, a file **main.xc** is provided which includes a pre-authored **main()** function along with all of the required hardware resource declarations. Code is generated based on the options provided by the developer in **xua_conf.h**.

Using this development model the user simply authors a **xua_conf.h** with their settings and optional implementations of any ‘user functions’ as desired. This, along with an



XN file for their hardware platform, is all that is required to build a fully featured and functioning product. This XN file should contain definitions of the ports used for the various **lib_xua** functionality, see [Options](#).

This development model also provides the benefit of a full and verified codebase as a basis for a product.

This behaviour described in this section is the default behaviour of **lib_xua**, to disable this set **EXCLUDE_USB_AUDIO_MAIN** to 1 in the application *CMakeLists.txt* or *xua_conf.h* and see [Advanced usage](#).

3.4 Configuring lib_xua

Configuration of the various build time options of **lib_xua** is done via the optional header *xua_conf.h*. To allow the build system to locate this file it should reside somewhere in the application *src* directory.

Such build time options include audio class version, sample rates, channel counts etc. See [API reference](#) for full listings.

The build system will automatically include the *xua_conf.h* header file as appropriate - the developer should continue to include *xua.h* as previously directed. A simple example *xua_conf.h* file is shown below:

```
#ifndef _XUA_CONF_H_
#define _XUA_CONF_H_

/* Output channel count */
#define XUA_NUM_USB_CHAN_OUT (2)

/* Product string */
#define XUA_PRODUCT_STR_A2 "My Product"

#endif
```

3.5 User functions

To enable custom functionality, such as configuring external audio hardware, bespoke behaviour on stream start/stop etc, various functions can be overridden by the user. (see [API reference](#) for full listings). The default implementations of these functions are empty.

3.6 Extending the “Codeless” application

The *main.xc* function allows insertion of extra code using the preprocessor. For example, you may wish to add some control code to control buttons or LEDs or DSP tasks for audio enhancement.

Adding Globals

An optional header file named *xua_conf_globals.h* can be added to the project.

If this file exists in the project’s source tree, its contents will be inserted into *main.xc* at global scope.

Example contents of *xua_conf_globals.h*:

```
unsigned my_global_var = 42;
```

This file may also be used to include additional headers or declare function prototypes, for example:



```
#include "my_header.h"
void my_function(int x);
```

This allows these functions and variables to be available for use in custom code inserted by the following two sections (e.g. user-defined tasks or initialisation).

Adding main function declarations

To add declarations to the application - such as channels or interfaces for communication between tasks — the define `USER_MAIN_DECLARATIONS` can be set in the `xua_conf.h` header file. This inserts code into `main.xc` after the `main()` definition but before the main `par` statement.

Alternatively, an optional header file may be added to the project called `xua_conf_declarations.h`. If this file exists in the project's source tree, its contents will be inserted into `main.xc` before the main `par` statement.

Example contents of `xua_conf_declarations.h`:

```
chan c_usb_to_user_interface;
```

Adding main function tasks

To add extra tasks to the application, the define `USER_MAIN_TASKS` can be set in the `xua_conf.h` header file. This will insert code into `main.xc` after the main `par` statement, allowing the compiler to run these tasks in parallel — either on a dedicated hardware thread or combined with other tasks if marked as `[combinable]`.

Alternatively, an optional header file called `xua_conf_tasks.h` can be added to project. If this file exists anywhere in the project source tree, its contents will be inserted into `main.xc` after the main `par` statement.

Example contents of `xua_conf_tasks.h`:

```
on tile[1]: my_user_interface_task(c_usb_to_user_interface);
```



4 Options

This section describes key options of **lib_xua**. These are typically controlled using build time defines. Where something must be defined, it is recommended this is done in *xua_conf.h* but could also be done in the application *CMakeLists.txt*.

For full details of all options please see [API reference](#).

4.1 Strings and IDs

The codebase includes various strings and IDs that should be customised to match the product requirements. These are listed in [Table 6](#).

The Vendor ID (VID) should be acquired from the USB Implementers Forum (www.usb.org).

Warning

Under no circumstances should the XMOS VID or any other VID be used without express permission.

The VID and Product ID (PID) pair must be unique to each product, otherwise driver incompatibilities may arise.

Table 6: String & ID defines

Define	Description	Default
VENDOR_STR	Name of vendor/manufacturer, note the is appended to various strings.	" XMOS "
PRODUCT_STR_A2	Name of the product when running in Audio Class 2.0 mode	" XMOS xCORE (UAC2.0) "
PRODUCT_STR_A1	Name of the product when running in Audio Class 1.0 mode	" XMOS xCORE (UAC1.0) "
PID_AUDIO_2	Product ID when running in Audio Class 2.0 mode	0x0002
PID_AUDIO_1	Product ID when running in Audio Class 1.0 mode	0x0003



4.2 Code Location

When designing a system there is a choice as to which hardware resources to use for each interface. In a multi-tile system the codebase needs to be informed as to which tiles to use for these hardware resources and associated code.

A series of defines are used to allow the programmer to easily move code between tiles. Arguably the most important of these are `XUA_AUDIO_IO_TILE_NUM` and `XUA_XUD_TILE_NUM`. [Table 7](#) shows a full listing of these `TILE` defines.

Note

If not explicitly defined by user, tile numbers will be derived from the application XN file `PORT` defines. In general this is the recommended approach.

Table 7: Tile defines

Define	Description	Default
<code>XUA_AUDIO_IO_TILE_NUM</code>	Tile on which I2S/TDM, ADAT Rx, S/PDIF Rx & mixer resides	Derived from related port defines in
<code>XUA_XUD_TILE_NUM</code>	Tile on which USB resides, including buffering for all USB interfaces/endppoints	<code>0</code>
<code>XUA_MIDI_TILE_NUM</code>	Tile on which MIDI resides	Derived from MIDI related port defin
<code>XUA_SPDIF_TX_TILE_NUM</code>	Tile on which S/PDIF Tx resides	Derived from <code>PORT_SPDIF_OUT</code> por
<code>XUA_MIC_PDM_TILE_NUM</code>	Tile on which PDM microphones resides	Derived from Mic related port defin
<code>XUA_PLL_REF_TILE_NUM</code>	Tile on which reference signal to CS2100 resides	Derived from <code>PORT_PLL_REF</code> port d

Note

It should be ensured that the relevant port defines in the application XN file match the code location defines



4.3 Channel counts and sample rates

lib_xua is fully configurable in relation to channel counts and sample rates. Practical limitations of these are normally based on USB packet size restrictions and I/O availability.

For example, the maximum packet size for high-speed USB is 1024 bytes, limiting the channel count to 10 channels for a device running at 192kHz with 32bit sample depth.

The defines in [Table 8](#) set the channel counts exposed to the USB host.

Table 8: Channel count defines

Define	Description	Default
NUM_USB_CHAN_OUT	Number of output channels the device advertises to the USB host	N/A (must be defined)
NUM_USB_CHAN_IN	Number of input channels the device advertises to the USB host	N/A (must be defined)

Sample rates ranges are set by the defines in [Table 9](#). The codebase will automatically populate the device sample rate list with popular frequencies between the min and max values. All values are in Hz:

Table 9: Sample rate defines

Define	Description	Default
MAX_FREQ	Maximum supported sample rate (Hz)	192000
MIN_FREQ	Minimum supported sample rate (Hz)	44100
DEFAULT_FREQ	Starting frequency for the device after boot	MIN_FREQ

The codebase requires knowledge of the two master clock frequencies that will be present on the master-clock port(s). One for 44.1kHz, 88.2kHz etc and one for 48kHz, 96kHz etc. These are set using defines in [Table 10](#). All values are in Hz.

Table 10: Master clock rate defines

Define	Description	Default
CLK_441	Master clock defines for 44100 rates (Hz)	(256 * 44100)
MCLK_48	Master clock defines for 48000 rates (Hz)	(256 * 48000)



4.4 USB Audio Class version

The codebase supports USB Audio Class (UAC) versions 1.0 and 2.0.

UAC 2.0 offers many improvements over UAC 1.0, most notable is the complete support for high-speed (HS) operation. This means that Audio Class devices are no longer limited to full-speed (FS) operation allowing greater channel counts, sample frequencies and sample bit-depths. Additional improvements, amongst others, include:

- ▶ Added support for multiple clock domains, clock description and clock control
- ▶ Extensive support for interrupts to inform the host about dynamic changes that occur to different entities such as Clocks etc

Driver support

Audio Class 1.0

- ▶ Supported in Apple macOS.
- ▶ Supported in all modern Microsoft Windows operating systems (i.e. Windows XP and later).

Audio Class 2.0

- ▶ Supported in Apple macOS since version 10.6.4.
- ▶ Supported in Windows since version 10, release 1703.

Third party Windows drivers are also available, however, documentation of these is beyond the scope of this document, please contact XMOS for further details.

Configuring Audio Class version

Configuring the **AUDIO_CLASS** define to **1** or **2** will set the UAC version for the device to 1.0 or 2.0 respectively.

The default value is **2** which causes the device to run as a HS UAC 2.0 device when connected to a HS host/hub and as a FS UAC 2.0 device when connected to a FS host/hub.

Setting **AUDIO_CLASS** to **1** will cause the device to run as a FS Audio Class 1.0 device.

Warning

To ensure specification compliance, Audio Class 1.0 mode is not supported at high-speed.

Defines are also provided to allow a different UAC version for HS and FS:

- ▶ **XUA_AUDIO_CLASS_HS**: UAC version to run at high-speed (0: Disabled, 2: Audio Class 2.0)
- ▶ **XUA_AUDIO_CLASS_FS**: UAC version to run at full-speed (0: Disabled, 1: Audio Class 1.0, 2: Audio Class 2.0)



Warning

Disabling Audio Class support or dynamically switching between Audio Class 1.0 and 2.0 based on USB bus speed may lead to USB compliance issues.

The USB-IF views such behavior as a significant functional change, which may violate compliance requirements. Devices are expected to maintain consistent functionality regardless of bus speed, and altering the USB class or interface descriptors dynamically can result in unpredictable host behavior and test failures during USB-IF certification.

Recommendation: Avoid switching USB Audio Class modes based on bus speed.

Due to bandwidth limitations of FS USB the following restrictions are applied during FS operation for both UAC 1.0 and 2.0 modes:

- ▶ Sample rate is limited to a maximum of 48kHz if both input *and* output is enabled.
- ▶ Sample rate is limited to a maximum of 96kHz if only input *or* output is enabled.
- ▶ Channel count is limited to a maximum of 2 channels for both input and output paths.

Audio Class 1.0 devices Some products may opt to operate in UAC 1.0 mode to enable driver-less compatibility with older Windows versions or certain embedded hosts. This mode was historically preferred for ensuring basic plug-and-play audio functionality without requiring custom drivers.

However, the need for UAC 1.0 support is diminishing as UAC 2.0 becomes more widely supported across modern operating systems and embedded platforms. UAC 2.0 offers better performance, higher sample rates, and more robust feature support, and is now natively supported on Windows 10+, macOS, Linux, and many embedded systems.

Recommendation: Where possible, default to UAC 2.0 for new designs unless specific host compatibility requirements mandate support for UAC 1.0.

The device will operate in FS UAC 1.0 mode if one of the following is true:

- ▶ The code is compiled for USB Audio Class 1.0 *only* i.e. - **AUDIO_CLASS** is set to **1** *or* - **XUA_AUDIO_CLASS_HS** is set to **0** and **XUA_AUDIO_CLASS_FS** is set to **1**.
- ▶ The code is compiled for UAC 2.0 at HS and UAC 1.0 at FS i.e. **XUA_AUDIO_CLASS_HS** is set to **2** and **XUA_AUDIO_CLASS_FS** is set to **1** and the device is connected to a host via a full-speed link

Related defines

Table 11 describes the defines that affect audio class selection:



Table 11: Audio Class defines

Define	Description	Default
AUDIO_CLASS	Audio Class version (1 or 2)	2
XUA_AUDIO_CLASS_HS	Audio Class version to run at high-speed (0: Disabled, 2 UAC 2.0)	2
XUA_AUDIO_CLASS_FS	Audio Class version to run at full-speed (0: Disabled, 1: UAC 1.0, 2: UAC 2.)	2



4.5 Synchronisation

The codebase supports “Synchronous” and “Asynchronous” modes for USB transfer as defined by the USB specification(s).

Asynchronous mode (**XUA_SYNCMODE_ASYNC**) has the advantage that the device is clock-master. This means that a high-quality local master-clock source can be utilised. It also has the benefit that the device may synchronise its master clock to an external digital input stream e.g. S/PDIF and thus avoiding sample-rate conversion.

The drawback of this mode is that it burdens the host with syncing to the device which some hosts may not support. This is especially pertinent to embedded hosts, however, most PCs and mobile devices will indeed support this mode.

Synchronous mode (**XUA_SYNCMODE_SYNC**) is an option if the target host does not support asynchronous mode or if it is desirable to synchronise many devices to a single host. It should be noted, however, that input from digital streams, such as S/PDIF, are not currently supported in this mode.

Note

The selection of synchronisation mode is done at build time and cannot be changed dynamically.

Setting the synchronisation mode of the device is done using the defines in [Table 12](#).

Table 12: Sync mode

Define	Description	Default
XUA_SYNCMODE	USB synchronisation mode	XUA_SYNCMODE_ASYNC

When operating in synchronous mode a local master clock must be generated that is synchronised to the incoming SoF rate from USB. Either an external Cirrus Logic CS2100 device is required for this purpose or, on *xcore.ai* devices, the on-chip application PLL may be used via [lib_sw_pll](#). In the case of using the CS2100, the codebase expects to drive a synchronisation signal to this external device as a reference.

The programmer should ensure the defines in [Table 13](#) are set appropriately.

Table 13: Reference clock location

Define	Description	Default
XUA_PLL_REF_TILE_NUM	Location of reference to CS2100 device	Derived from PORT_PLL_REF in the application XN file.
XUA_USE_SW_PLL	Whether or not to use <i>sw_pll</i> to recover the clock (<i>xcore.ai</i> only)	1 (enabled) for <i>xcore.ai</i> targets.



The codebase expects the CS2100 reference signal port to be defined in the application XN file as **PORT_PLL_REF**. This may be a port of any bit-width, however, connection to bit[0] is assumed:

```
<Port Location="XS1_PORT_1A" Name="PORT_PLL_REF"/>
```

Configuration of the external CS2100 device (typically via I2C) is beyond the scope of this document.



4.6 I²S/TDM

I²S/TDM is typically fundamental to most products and is built into the `XUA_AudioHub()` thread.

Table 14 lists the defines that affect the I²S implementation.

Table 14: I²S defines

Define	Description	Default
<code>I2S_CHANS_DAC</code>	The desired number of output channels via I2S (0 for disabled)	N/A (Must be defined)
<code>I2S_CHANS_ADC</code>	The desired number of input channels via I2S (0 for disabled)	N/A (Must be defined)
<code>XUA_PCM_FORMAT</code>	Enables either TDM or I2S mode	<code>XUA_PCM_FORMAT_I2S</code>
<code>CODEC_MASTER</code>	Sets if <i>xcore</i> is I2S master or slave	0 (<i>xcore</i> is master)
<code>XUA_I2S_N_BITS</code>	I2S/TDM word length (16, 32-bit supported)	32

The I²S code expects that the ports required for I²S (master clock, LR-clock, bit-clock and data lines) are defined in the application XN file on the relevant **Tile**. For example:

```
<Tile Number="0" Reference="tile[0]">
  <Port Location="XS1_PORT_1A" Name="PORT_MCLK_IN"/>
  <Port Location="XS1_PORT_1B" Name="PORT_I2S_LRCLK"/>
  <Port Location="XS1_PORT_1C" Name="PORT_I2S_BCLK"/>
  <Port Location="XS1_PORT_1D" Name="PORT_I2S_DAC0"/>
  <Port Location="XS1_PORT_1E" Name="PORT_I2S_DAC1"/>
  <Port Location="XS1_PORT_1F" Name="PORT_I2S_ADC0"/>
  <Port Location="XS1_PORT_1G" Name="PORT_I2S_ADC1"/>
</Tile>
```

All of the I²S/TDM related ports must be 1-bit ports.

Note

TDM mode allows 8 channels (rather than 2) to be supplied on each data-line.

Note

Data output/input is in "I²S" format, rather than, say "left-justified" or "right-justified" formats. I²S format specifies a single bit-clock delay after the LR-clock transition before sample-data is driven/received. This also applies to TDM mode. TDM support in ADC/DAC hardware is quite varied, an "offset" value may need to be programmed into the external device for compatible operation.



4.7 S/PDIF transmit

The codebase supports a single, stereo, S/PDIF transmitter. This can be output over 75 Ω coaxial or optical fibre. In order to provide S/PDIF transmit functionality **lib_xua** uses **lib_spdif**.

Basic configuration of S/PDIF transmit functionality is achieved with the defines in [Table 15](#).

Table 15: S/PDIF tx defines

Define	Description	Default
XUA_SPDIF_TX_EN	Enable S/PDIF transmit	0 (Disabled)
SPDIF_TX_INDEX	Output channel offset to use for S/PDIF transmit	0

In addition, the developer may choose which tile the S/PDIF transmitter runs on, see [Table 16](#)

Table 16: S/PDIF tile define

Define	Description	Default
XUA_SPDIF_TX_TILE_NUM	Tile that S/PDIF tx is connected to	Derived from PORT_SPDIF_OUT define in the application XN file

The codebase expects the S/PDIF transmit port to be defined in the application XN file as **PORT_SPDIF_OUT**. This must be a 1-bit port, for example:

```
<Port Location="XS1_PORT_1A" Name="PORT_SPDIF_OUT"/>
```



4.8 S/PDIF receive

The codebase supports a single, stereo, S/PDIF receiver. This can be input via 75 Ω coaxial or optical fibre. In order to provide S/PDIF functionality **lib_xua** uses [lib_spdif](#).

Basic configuration of S/PDIF receive functionality is achieved with the defines in [Table 17](#)

Table 17: S/PDIF rx defines

Define	Description	Default
XUA_SPDIF_RX_EN	Enable S/PDIF receive	0 (Disabled)
SPDIF_RX_INDEX	Defines which channels S/PDIF will be input on	N/A (must be defined)

Note

S/PDIF receive always runs on the tile defined by **XUA_AUDIO_IO_TILE_NUM**

The codebase expects the S/PDIF receive port to be defined in the application XN file as **PORT_SPDIF_IN**. This must be a 1-bit port, for example:

```
<Port Location="XS1_PORT_1A" Name="PORT_SPDIF_IN" />
```

When S/PDIF receive is enabled the codebase expects to either drive a synchronisation signal to an external Cirrus Logic CS2100 device or use lib_sw_pll (xcore.ai only) for master-clock generation.

The programmer should ensure the defines in [Table 18](#) are set appropriately

Table 18: Reference clock location

Define	Description	Default
XUA_PLL_REF_TILE_NUM	Location of reference to CS2100 device	Derived from PORT_PLL_REF in XN file

The codebase expects this reference signal port to be defined in the application XN file as **PORT_PLL_REF**. This may be a port of any bit-width, however, connection to bit[0] is assumed:

```
<Port Location="XS1_PORT_1A" Name="PORT_PLL_REF" />
```

Configuration of the external CS2100 device (typically via I2C) is beyond the scope of this document.



4.9 ADAT transmit

The codebase supports a single ADAT transmitter that can transmit eight channels of uncompressed digital audio at sample-rates of 44.1 or 48 kHz over an optical cable. Higher rates are supported with a reduced number of samples via S/MUX ('sample multiplexing'). Using S/MUX, the ADAT transmitter can transmit four channels at 88.2 or 96 kHz or two channels at 176.4 or 192 kHz.

In order to provide ADAT transmit functionality **lib_xua** uses [lib_adat](#).

Basic configuration of ADAT transmit functionality is achieved with the defines in [Table 19](#).

Table 19: ADAT transmit defines

Define	Description	Default
XUA_ADAT_TX_EN	Enable ADAT transmit	0 (Disabled)
ADAT_TX_INDEX	Start channel index of ADAT TX channels	N/A (must be defined by the application)
ADAT_TX_USE_SHARED_BUFF	Use shared memory when transferring samples between AudioHub and the ADAT transmitter task	N/A (must be defined by the application)

The ADAT transmitter runs on the same tile as the Audio IO (**AUDIO_IO_TILE**)

The codebase expects the ADAT transmit port to be defined in the application XN file as **PORT_ADAT_OUT**. This must be a 1-bit port, for example:

```
<Port Location="XS1_PORT_1G" Name="PORT_ADAT_OUT" />
```



4.10 ADAT receive

The codebase supports a single ADAT receiver that can receive up to eight channels of audio at a sample rate of 44.1kHz or 48kHz over an optical interface. Higher rates are supported with a reduced number of samples via S/MUX ('sample multiplexing'). Using S/MUX, the ADAT receiver can receive four channels at 88.2 or 96 kHz or two channels at 176.4 or 192 kHz.

In order to provide ADAT functionality **lib_xua** uses **lib_adat**.

Basic configuration of ADAT receive functionality is achieved with the defines in [Table 20](#).

Table 20: ADAT RX defines

Define	Description	Default
XUA_ADAT_RX_EN	Enable ADAT receive	0 (Disabled)
ADAT_RX_INDEX	Start channel index of ADAT RX channels	N/A (must be defined by the application)

The codebase expects the ADAT receive port to be defined in the application XN file as **PORT_ADAT_IN**. This must be a 1-bit port, for example:

```
<Port Location="XS1_PORT_10" Name="PORT_ADAT_IN"/>
```

When ADAT receive is enabled the codebase expects to either drive a synchronisation signal to an external Cirrus Logic CS2100 device or use **lib_sw_pll** (*xcore.ai* only) for generating a master clock that is synchronised to the ADAT digital stream.

The programmer should ensure the defines in [Table 21](#) are set appropriately:

Table 21: Reference clock location

Define	Description	Default
XUA_PLL_REF_TILE_NUM	Location of reference signal to CS2100 device	Derived from location of PORT_PLL_REF in XN file

The codebase expects this reference signal port to be defined in the application XN file as **PORT_PLL_REF**. This may be a port of any bit-width, however, connection to bit[0] is assumed:

```
<Port Location="XS1_PORT_1A" Name="PORT_PLL_REF"/>
```

Configuration of the external CS2100 device (typically via I²C) is beyond the scope of this document.



4.11 MIDI

lib_xua supports MIDI input/output over USB as per [Universal Serial Bus Device Class Definition for MIDI Devices](#).

MIDI functionality is enabled with the define in [Table 22](#).

Table 22: MIDI enable define

Define	Description	Default
MIDI	Enable MIDI functionality	0 (Disabled)

lib_xua supports MIDI receive on a 4-bit or 1-bit port, defaulting to using a 1-bit port. MIDI transmit is supported over a port of any bit-width. By default **lib_xua** assumes the transmit and receive I/O is connected to bit[0] of the port. This is configurable for the transmit port. [Table 23](#) provides information on configuring these parameters.

Table 23: MIDI port defines

Define	Description	Default
MIDI_RX_PORT_WIDTH	Port width of the MIDI rx port (1 or 4bit)	1 (1-bit port)
MIDI_SHIFT_TX	MIDI tx bit	0 (bit[0])

The MIDI code expects that the ports for receive and transmit are defined in the application XN file on the relevant Tile. The expected names for the ports are **PORT_MIDI_IN** and **PORT_MIDI_OUT**, for example:

```
<Tile Number="0" Reference="tile[0]">
  <!-- MIDI -->
  <Port Location="XS1_PORT_1F" Name="PORT_MIDI_IN"/>
  <Port Location="XS1_PORT_4C" Name="PORT_MIDI_OUT"/>
</Tile>
```



4.12 PDM microphones

lib_xua supports input from up to 8 PDM microphones although this is extensible.

PDM microphone support is provided via [lib_mic_array](#). Settings for PDM microphones are controlled with the defines in [Table 24](#).

Table 24: PDM defines

Define	Description	Default
XUA_NUM_PDM_MICS	The number of mic outputs to enable (0 for disabled). This enables compilation of the PDM to PCM code also.	0 (disabled)
XUA_NUM_PDM_MICS_IN	The number of mic input lines. This defines the width of the PDM data port and must be at least XUA_NUM_PDM_MICS .	XUA_NUM_PDM_MICS
XUA_PDM_MIC_INDEX	Defines which starting input channel the mics map to	0
XUA_PDM_MIC_USE_PDM_ISR	Define as 1 to enable merging of the PDM receive task and decimation task into a single thread using an ISR	1 (Run PDM RX service as an ISR)
XUA_PDM_MIC_USE_DDR	Define as 1 to enable two microphones sharing a single data line (DDR mode)	1 (DDR mode)

Note

mic array task always runs on the tile defined by **XUA_MIC_PDM_TILE_NUM**, the value of which is inferred from the **PORT_PDM_CLK** port define in the application XN file.

Note

Currently the only supported sampling rates for the PDM microphones are 16kHz, 32kHz and 48kHz.

Note

Setting **XUA_PDM_MIC_USE_PDM_ISR** is only recommended for PDM mic counts below 8.

Please see the [PDM Microphones](#) section for further details.



4.13 Mixer

lib_xua supports audio mixing functionality with highly flexible routing options.

Essentially the mixer is capable of performing 8 separate mixes with up to 18 inputs at sample rates up to 96kHz and 2 mixes with up to 18 inputs at higher sample rates.

Inputs to the mixer can be selected from any device input (USB, S/PDIF, I2S etc) and outputs from the mixer can be routed to any device output (USB, S/PDIF, I2S etc).

See [Digital mixer](#) for full details of the mixer including control.

Basic configuration of mixer functionality is achieved with the the defines [Table 25](#).

Table 25: Mixer defines

Define	Description	Default
MIXER	Enable mixer	0 (Disabled)
MAX_MIX_COUNT	Number of separate mix outputs to perform	8
MIX_INPUTS	Number of channels input into the mixer	18

Note

The mixer threads always run on the tile defined by **XUA_AUDIO_IO_TILE_NUM**



4.14 Direct Stream Digital (DSD)

Direct Stream Digital (DSD) is used for digitally encoding audio signals on Super Audio CDs (SACD). It uses pulse-density modulation (PDM) encoding.

lib_xua supports DSD playback from the host via “DSD over PCM” (DoP) and a “Native” implementation which is, while USB specification based, proprietary to XMOS.

DSD is enabled with by setting the following [define](#) to a non-zero value:

Table 26: DSD defines

Define	Description	Default
DSD_CHANS_DAC	Number of DSD channels	0 (Disabled)

Typically this would be set to **2** for stereo output.

By default both “Native” and DoP functionality are enabled when DSD is enabled. The Native DSD implementation uses an alternative streaming interface such that the host can inform the device that DSD data is being streamed. See: [Audio stream formats](#) for details.

If only DoP functionality is desired the Native implementation can be disabled with the define in [Table 27](#).

Table 27: Native DSD defines

Define	Description	Default
NATIVE_DSD	Enable/Disable “Native” DSD implementation	1 (Enabled)

DSD over PCM (DoP)

DoP support follows the method described in the [DoP Open Standard 1.1](#).

While Native DSD support is available in Windows through a driver, macOS incorporates a USB driver that only supports PCM, this is also true of the central audio engine, Core-Audio. It is therefore not possible to use the “Native” scheme defined above using the built in driver of macOS.

Since the macOS only allows a PCM path a method of transporting DSD audio data over PCM frames has been developed.

Standard DSD has a sample size of 1 bit and a sample rate of 2.8224MHz - this is 64x the speed of a compact disc (CD). This equates to the same data-rate as a 16 bit PCM stream at 176.4kHz.

In order to clearly identify when this PCM stream contains DSD and when it contains PCM some header bits are added to the sample. A 24-bit PCM stream is therefore used, with the most significant byte being used for a DSD marker (alternating 0x05 and 0xFA values).

When enabled, if USB audio design detects a un-interrupted run of these samples (above a defined threshold) it switches to DSD mode, using the lower 16-bits as DSD sample data. When this check for DSD headers fails the design falls back to PCM mode. DoP



detection and switching is done completely in the Audio/I2S thread (*xua_audiohub.xc*). All other code handles the audio samples as PCM.

The design supports higher DSD/DoP rates (i.e. DSD128) by simply raising the underlying PCM sample rate e.g. from 176.4kHz to 352.8kHz. The marker byte scheme remains exactly the same regardless of rate.

Note

DoP requires bit-perfect transmission - therefore any audio/volume processing will corrupt the stream.

“Native” vs DoP

Since the DoP specification requires header bytes this eats into the data bandwidth. The “Native” implementation has no such overhead and can therefore transfer the same DSD rate at half the effective PCM rate of DoP. Such a property may be desired when supporting DSD128 without exposing a 352.8kHz PCM rate, for example.

Ports

The codebase expects 1-bit ports to be defined in the application XN file for the DSD data and clock lines for example:

```
<Port Location="XS1_PORT_1M" Name="PORT_DSD_DAC0"/>
<port Location="XS1_PORT_1N" Name="PORT_DSD_DAC1"/>
<Port Location="XS1_PORT_1G" Name="PORT_DSD_CLK"/>
```

Note

The DSD ports may or may not overlap the I2S ports - the codebase will reconfigure the ports as appropriate when switching between PCM and DSD modes.



4.15 DFU

The codebase supports DFU over USB implementation compliant with version 1.1 of [Universal Serial Bus Device Class Specification for Device Firmware Upgrade](#).

[Table 28](#) lists the DFU related configuration options.

Table 28: DFU defines

Define	Description	Default
XUA_DFU_EN	Enable DFU functionality	1 (Enabled)
DFU_PID	Product ID when enumerating in DFU mode. This is recommended to be different from the runtime device PID	PID_AUDIO_2 or PID_AUDIO_1 depending on whether the device is running Audio Class 2.0 or 1.0



4.16 Audio stream formats

The design currently supports up to three different stream formats for playback, selectable at run time. This is implemented using standard Alternate Settings to the Audio Streaming interfaces.

An Audio Streaming interface can have Alternate Settings that can be used to change certain characteristics of the interface and underlying endpoint. A typical use of Alternate Settings is to provide a way to change the subframe size and/or number of channels on an active Audio Streaming interface. Whenever an Audio Streaming interface requires an isochronous data endpoint, it must at least provide the default Alternate Setting (Alternate Setting 0) with zero bandwidth requirements (no isochronous data endpoint defined) and an additional Alternate Setting that contains the actual isochronous data endpoint. This zero bandwidth alternative setting 0 is always implemented by the design.

For further information refer to 3.16.2 of [USB Audio Device Class Definition for Audio Devices, release 2.0](#)

Customisable parameters for the Alternate Settings provided by the design are as follows.:

- ▶ Audio sample resolution
- ▶ Audio sample subslot size
- ▶ Audio data format

Note

Currently only a single format is supported for the recording stream

By default the design exposes two sets of Alternative Settings for the playback Audio Streaming interface, one for 16-bit and another for 24-bit playback. When DSD is enabled an additional (32-bit) alternative is exposed.

Audio subslot

An audio subslot holds a single audio sample. See [USB Device Class Definition for Audio Data Formats](#) for full details. This is represented by *bSubslotSize* in the devices descriptor set.

An audio subslot always contains an integer number of bytes. The specification limits the possible audio subslot size to 1, 2, 3 or 4 bytes per audio subslot.

Since the *xcore* is a 32-bit machine the value 4 is typically used for *bSubSlot* - this means that packing/unpacking samples to/from packets is computationally trivial. Other values can, however, be used and the values 4, 3 and 2 are supported.

Values other than 4 may be used for the following reasons:

- ▶ Bus-bandwidth needs to be efficiently utilised. For example maximising channel-count/sample-rates in full-speed operation.
- ▶ To support restrictions with certain hosts. For example, historically many Android based hosts support only 16 bit samples in a 2-byte subslot.



bSubSlot size is set using the following defines:

- ▶ When running in high-speed:
 - ▶ *HS_STREAM_FORMAT_OUTPUT_1_SUBSLOT_BYTES*
 - ▶ *HS_STREAM_FORMAT_OUTPUT_2_SUBSLOT_BYTES*
 - ▶ *HS_STREAM_FORMAT_OUTPUT_3_SUBSLOT_BYTES*
- ▶ When running in full-speed:
 - ▶ *FS_STREAM_FORMAT_OUTPUT_1_SUBSLOT_BYTES*
 - ▶ *FS_STREAM_FORMAT_OUTPUT_2_SUBSLOT_BYTES*
 - ▶ *FS_STREAM_FORMAT_OUTPUT_3_SUBSLOT_BYTES*

Audio sample resolution

An audio sample is represented using a number of bits (*bBitResolution*) less than or equal to the number of total bits available in the audio subslot i.e. $bBitResolution \leq bSubslot-Size * 8$). The design supports values 16, 24 and 32.

bBitResolution is set using the following defines:

- ▶ When operating at high-speed:
 - ▶ *HS_STREAM_FORMAT_OUTPUT_1_RESOLUTION_BITS*
 - ▶ *HS_STREAM_FORMAT_OUTPUT_2_RESOLUTION_BITS*
 - ▶ *HS_STREAM_FORMAT_OUTPUT_3_RESOLUTION_BITS*
- ▶ When operating at full-speed:
 - ▶ *FS_STREAM_FORMAT_OUTPUT_1_RESOLUTION_BITS*
 - ▶ *FS_STREAM_FORMAT_OUTPUT_2_RESOLUTION_BITS*
 - ▶ *FS_STREAM_FORMAT_OUTPUT_3_RESOLUTION_BITS*

Audio format

The design supports two audio formats, PCM and, when “Native” DSD is enabled, Direct Stream Digital (DSD). A DSD capable DAC is required for the latter.

The USB Audio *Raw Data* format is used to indicate DSD data (2.3.1.7.5 of [USB Device Class Definition for Audio Data Formats](#)). This use of a RAW/DSD format in an alternative setting is termed by XMOS as *Native DSD*

The following defines affect both full-speed and high-speed operation:

- ▶ *STREAM_FORMAT_OUTPUT_1_DATAFORMAT*
- ▶ *STREAM_FORMAT_OUTPUT_2_DATAFORMAT*
- ▶ *STREAM_FORMAT_OUTPUT_3_DATAFORMAT*

The following options are supported:

- ▶ *UAC_FORMAT_TYPE1_RAW_DATA*
- ▶ *UAC_FORMAT_TYPE1_PCM*



Note

Currently DSD is only supported on the output/playback stream

Note

4 byte slot size with a 32 bit resolution is required for RAW/DSD format

Native DSD requires driver support and is available in the Thesycon Windows driver via ASIO.



4.17 Other options

There are a few other, lesser used, options available. These are shown in [Table 29](#).

Table 29: Other defines

Define	Description	Default
XUA_USB_EN	Allows the use of the audio subsystem without USB	1 (enabled)
INPUT_VOLUME_CONTROL	Enables volume control on input channels, both descriptors and processing	1 (enabled)
OUTPUT_VOLUME_CONTROL	Enables volume control on output channels, both descriptors and processing	1 (enabled)
XUA_CHAN_BUFF_CTRL	Enables event based communication between XUA_Buffer_Ep() and XUA_Buffer_Decouple() which significantly reduces power consumption (approx 40 mW on xcore.ai) at the cost of consuming two extra channel-ends. Consequently this option may not be viable on some high end configurations which feature multiple digital interfaces such as SPDIF or ADAT.	0 (disabled)
XUA_USER_IN_ENDPOINTS	Allows additional input USB endpoints to be declared. Endpoints must be initialised using code from the xua_user_endpoint_init.h include file.	Undefined
XUA_USER_OUT_ENDPOINTS	Allows additional output USB endpoints to be declared. Endpoints must be initialised using code from the xua_user_endpoint_init.h include file.	Undefined
XUA_USER_INTERFACES	Used for inserting interfaces into descriptor for composite devices. Note - This define must be able to be compiled under C.	Undefined

Note

When extending the descriptors to add user interfaces, the descriptors must also be extended to tell the host what kind of device to expect. There are three optional include files which may be used to insert code into user descriptors. The expected files in your project are **xua_user_descriptors_incl.h** to add any additional include files, **xua_user_descriptors_decl.h** to make the descriptor declarations and **xua_user_descriptors_content.h** to populate them. The include files must be able to be compiled under C.

Note

In addition to adding user interfaces and descriptors, endpoint 0 handlers and endpoint initialisation must be supplied. The expected files in your project are **xua_user_ep0_decl.h** to make declarations (for example external function prototype), **xua_user_ep0_handler.h** to provide handling code for the extensions to endpoint 0 and **xua_user_endpoint_init.h** where the additionally declared endpoints are initialised. The include files for endpoint 0 must be able to be compiled under C.



5 Advanced usage

Whilst it is possible to program USB Audio devices using **lib_xua** by only setting defines (see “[Codeless programming model](#)”) some developers may want to code a USB Audio device from scratch using the building blocks provided by **lib_xua**.

This could be for a number of reasons, adding complex DSP, merging with some other functionality etc. This section describes these building blocks and their use.

Reviewing application note AN00246 is highly recommended at this point.

5.1 Core hardware resources

The user must declare and initialise relevant hardware resources (globally) and pass them to the relevant function of **lib_xua**.

As an absolute minimum the following resources are required:

- ▶ A 1-bit port for audio master clock input
- ▶ A clock-block, which will be clocked from the master clock input port

When using the default asynchronous mode of operation an additional port is required:

- ▶ A n-bit port for internal feedback calculation (typically a free, unused port is used e.g. **XS1_PORT_16B**)

Example declaration of these resources might look as follows:

```
in port p_mclk_in      = PORT_MCLK_IN;
in port p_for_mclk_count = PORT_MCLK_COUNT; /* Extra port for counting master clock ticks */
clock clk_audio_mclk   = on tile[0]: XS1_CLKBLK_5; /* Master clock */
```

Note

The **PORT_MCLK_IN** and **PORT_MCLK_COUNT** definitions are derived from the projects XN file

The **XUA_AudioHub()** function typically requires an audio master clock input to clock the physical audio I/O such as S/PDIF transmit and I²S¹. Less obvious is the reasoning for the **XUA_Buffer()** task having the same requirement when running in asynchronous mode - it is used for the USB feedback system and packet sizing.

Due to the above, if the **XUD_AudioHub()** and **XUA_Buffer()** cores must reside on separate tiles a separate master clock input port must be provided to each, for example:

```
/* Master clock for the audio IO tile */
in port p_mclk_in      = PORT_MCLK_IN;

/* Resources for USB feedback */
in port p_mclk_in_usb  = PORT_MCLK_IN_USB; /* Extra master clock input for the USB tile */
```

Whilst the hardware resources described in this section satisfy the basic requirements for the operation (or build) of **lib_xua**, projects typically also need some additional audio I/O, I²S or S/PDIF for example.

These should be passed into the various task functions as required - see [API reference](#).

¹ This is not the case when *only* I²S *slave* is used.





5.2 Running the core components

In their most basic form the core components can be run as follows:

```
par
{
    /* Endpoint 0 thread from lib_xua */
    XUA_Endpoint0(c_ep_out[0], c_ep_in[0], c_aud_ctl, ...);

    /* Buffering threads - handles audio data to/from EP's and gives/gets data to/from the audio I/O thread */
    /* Note, this spawns two threads */
    XUA_Buffer(c_ep_out[1], c_ep_in[1], c_sof, c_aud_ctl, p_for_mclk_count, c_aud);

    /* AudioHub/I/O thread does most of the audio IO i.e. I2S (also serves as a hub for all audio) */
    XUA_AudioHub(c_aud, ...) ;
}
```

`XUA_Buffer()` expects the `p_for_mclk_count` argument to be clocked from the audio master clock before receiving it as a parameter. The following code satisfies this requirement:

```
{
    /* Connect master-clock clock-block to clock-block pin */

    /* Clock clock-block from mclk pin */
    set_clock_src(clk_audio_mclk_usb, p_mclk_in_usb);

    /* Clock the "count" port from the clock block */
    set_port_clock(p_for_mclk_count, clk_audio_mclk_usb);

    /* Set the clock off running */
    start_clock(clk_audio_mclk_usb);

    XUA_Buffer(c_ep_out[1], c_ep_in[1], c_sof, c_aud_ctl, p_for_mclk_count, c_aud);
}
```

Note

Keeping this configuration outside of `XUA_Buffer()` means the possibility of sharing the `p_mclk_in_usb` port with additional tasks is not precluded.

For USB connectivity a call to `XUD_Main()` (from `lib_xud`) must also be made:

```
/* Low level USB device layer thread */
on tile[1]: XUD_Main(c_ep_out, 2, c_ep_in, 2, c_sof, epTypeTableOut, epTypeTableIn, null, null, -1, XUD_SPEED_HS,
↳ XUD_PWR_SELF);
```

Additionally, the required communication channels must also be declared:

```
/* Channel arrays for lib_xud */
chan c_ep_out[2];
chan c_ep_in[2];

/* Channel for communicating SOF notifications from XUD to the Buffering threads */
chan c_sof;

/* Channel for audio data between buffering threads and AudioHub/I/O thread */
chan c_aud;

/* Channel for communicating control messages from EP0 to the rest of the device (via the buffering threads) */
chan c_aud_ctl;
```

This section provides enough information to implement a skeleton program for a USB Audio device. When running the `xcore` device will present itself as a USB Audio Class device on the bus. Audio streaming will be impaired since tasks relating to physical audio interfaces are yet to be instantiated.



5.3 I²S/TDM

I²S/TDM is typically fundamental to most products and is built into the `XUA_AudioHub()` thread.

In order to enable I²S/TDM one must declare an array of ports for the data-lines (one for each direction):

```
/* Port declarations. Note, the defines come from the XN file */
buffered out port:32 p_i2s_dac[] = {PORT_I2S_DAC0}; /* I2S Data-line(s) */
buffered in port:32 p_i2s_adc[] = {PORT_I2S_ADC0}; /* I2S Data-line(s) */
```

Ports for the sample and bit clocks are also required:

```
buffered out port:32 p_lrclk = PORT_I2S_LRCLK; /* I2S Bit-clock */
buffered out port:32 p_bclk = PORT_I2S_BCLK; /* I2S L/R-clock */
```

Note

All of these ports must be 1-bit ports, 32-bit buffered. Based on whether the *xcore* is bus slave/master the ports must be declared as input/output respectively.

These ports must then be passed to the `XUA_AudioHub()` task appropriately.

I²S/TDM functionality also requires two clock-blocks, one for bit-clock and another for the master clock e.g.:

```
/* Clock-block declarations */
clock clk_audio_bclk = on tile[0]: XS1_CLKBLK_4; /* Bit clock */
clock clk_audio_mclk = on tile[0]: XS1_CLKBLK_5; /* Master clock */
```

These hardware resources must be passed into the call to `XUA_AudioHub()`:

```
/* AudioHub/I/O thread does most of the audio I/O i.e. I2S (also serves
 * as a hub for all audio) */
on tile[0]: XUA_AudioHub(c_aud, clk_audio_mclk, clk_audio_bclk, p_mclk_in,
    p_lrclk, p_bclk, p_i2s_dac, p_i2s_adc);
```



5.4 Mixer

Since the mixer has no I/O the instantiation is straight forward. Communication wise, the mixer threads are inserted between the *AudioHub* and Buffering thread(s)

It takes three channel ends as parameters, one for audio to/from the buffering thread(s), one for audio to/from the *AudioHub* thread and another one for control requests from the *Endpoint0* thread.

The mixer task will automatically handle the change in mix count based on the current sample frequency (communicated via the data channels from the buffering task).

An example of how the mixer task might be called is shown below (some parameter lists are abbreviated)

```
chan c_aud_0, c_aud_1, c_mix_ctl1;

par
{
    XUA_Buffer(..., c_aud_0, ...);

    mixer(c_aud0, c_aud_1, c_mix_ctl1);

    XUA_AudioHub(c_aud_1, ...);

    XUA_Endpoint0(..., c_mix_ctl1, ...);
}
```



6 Additional features

The previous chapter describes the use of core functionality contained within **lib_xua**. This section details enabling additional features with supported external dependencies, for example, **lib_xua** can provide S/PDIF output through the use of **lib_spdif**.

Where something must be defined, it is recommended this is done in *xua_conf.h* but could also be done in the application *CMakeLists.txt*.

6.1 S/PDIF transmit

lib_xua supports the development of devices with S/PDIF transmit functionality through the use of **lib_spdif**. The XMOS S/PDIF transmitter runs on a single thread and supports rates up to 192 kHz.

The S/PDIF transmitter thread takes PCM audio samples via a channel and outputs them in S/PDIF format to a port. Samples are provided to the S/PDIF transmitter task from the **XUA_AudioHub()** task.

The channel should be declared as normal:

```
chan c_spdif_tx
```

In order to use the S/PDIF transmitter with **lib_xua** a 1-bit port must be declared e.g:

```
buffered out port:32 p_spdif_tx = PORT_SPDIF_OUT;    /* SPDIF transmit port */
```

This port should be clocked from the master-clock, **lib_spdif** provides a helper function for setting up the port:

```
spdif_tx_port_config(p_spdif_tx, clk_audio_mclk, p_mclk_in, delay);
```

Note

If sharing the master-clock port and clockblock with **XUA_AudioHub()** (or any other task) then this setup should be done before running the tasks in a **par** statement.

Finally the S/PDIF transmitter task must be run - passing in the port and channel for communication with **XUA_AudioHub**. For example:

```
par
{
    while(1)
    {
        /* Run the S/PDIF transmitter task */
        spdif_tx(p_spdif_tx, c_spdif_tx);
    }

    /* AudioHub/IO thread does most of the audio IO i.e. I2S (also serves as
    * a hub for all audio).
    * Note, since we are not using I2S we pass in null for LR and Bit
    * clock ports and the I2S dataline ports */
    XUA_AudioHub(c_aud, clk_audio_mclk, null, p_mclk_in, null, null,
        null, null, c_spdif_tx);
}
```

For further details please see the documentation, application notes and examples provided for **lib_spdif**.



6.2 S/PDIF receive

lib_xua supports the development of devices with S/PDIF receive functionality through the use of [lib_spdif](#). The XMOS S/PDIF receiver runs on a single thread and supports rates up to 192 kHz.

The S/PDIF receiver inputs data via a port and outputs samples via a channel. It requires a 1-bit port. For example:

```
in port p_spdif_rx = PORT_SPDIF_IN;
```

It also requires a clock-block, for example:

```
clock clk_spd_rx = XS1_CLKBLK_1;
```

Finally, a channel for the output samples must be declared, note, this should be a streaming channel:

```
streaming chan c_spdif_rx;
```

The S/PDIF receiver should be called on the appropriate tile:

```
spdif_rx(c_spdif_rx, p_spdif_rx, clk_spd_rx, 192000);
```

Note

It is recommended to use the value 192000 for the **sample_freq_estimate** parameter

With the steps above an S/PDIF stream can be captured by the *xcore*. To be functionally useful the audio master clock must be able to synchronise to this external digital stream. Additionally, the host can be notified regarding changes in the validity of this stream, it's frequency etc.

To synchronise to external streams the codebase assumes the use of an external Cirrus Logic CS2100 device or [lib_sw_pll](#) on *xcore.ai* designs.

The **ClockGen()** task from **lib_xua** provides the reference signal to the CS2100 device or timing information to [lib_sw_pll](#) and also handles recording of clock validity etc. See [External clock recovery \(Clock Gen\)](#) for full details regarding **ClockGen()**.

It also provides a small FIFO for S/PDIF samples before they are forwarded to the **AudioHub** thread. As such it is required to be inserted in the communication path between the S/PDIF receiver and the **AudioHub** thread. For example:

```
chan c_dig_rx;
streaming chan c_spdif_rx;

par
{
    SpdifReceive(..., c_spdif_rx, ...);

    clockGen(c_spdif_rx, ..., c_dig_rx, ...);

    XUA_AudioHub(..., c_dig_rx, ...);
}
```

6.3 ADAT transmit

lib_xua supports the development of devices with ADAT transmit functionality through the use of [lib_adat](#). The XMOS ADAT transmitter runs on a single thread and supports



transmitting 8 channels of digital audio at 44.1 or 48 kHz. Higher rates are supported with a reduced number of samples via S/MUX ('sample multiplexing'). Using S/MUX, the ADAT transmitter can transmit four channels at 88.2 or 96 kHz (SMUX II) or two channels at 176.4 or 192 kHz (SMUX IV).

ADAT transmitter requires a thread to run on. Blocks of audio samples are transmitted from the `XUA_AudioHub()` to the ADAT transmitter over either a dedicated channel or a combination of a channel and shared memory.

Each block of audio samples is made of 8 samples. At sampling rates 44.1/48 kHz (SMUX I), this consists of a single sample of each of the eight ADAT channels:

- ▶ Channel 0 sample
- ▶ Channel 1 sample
- ▶ Channel 2 sample
- ▶ Channel 3 sample
- ▶ Channel 4 sample
- ▶ Channel 5 sample
- ▶ Channel 6 sample
- ▶ Channel 7 sample

At 88.2/96 kHz (SMUX II), the audio sample block consists of two samples per four ADAT channels:

- ▶ Channel 0 sample 0
- ▶ Channel 0 sample 1
- ▶ Channel 1 sample 0
- ▶ Channel 1 sample 1
- ▶ Channel 2 sample 0
- ▶ Channel 2 sample 1
- ▶ Channel 3 sample 0
- ▶ Channel 3 sample 1

At 176.4/192 kHz (SMUX IV), the audio sample block consists of four samples per two ADAT channels:

- ▶ Channel 0 sample 0
- ▶ Channel 0 sample 1
- ▶ Channel 0 sample 2
- ▶ Channel 0 sample 3
- ▶ Channel 1 sample 0
- ▶ Channel 1 sample 1



- Channel 1 sample 2
- Channel 1 sample 3

The configuration option `ADAT_TX_USE_SHARED_BUFF` determines whether the audio samples block is transmitted using only a channel (`ADAT_TX_USE_SHARED_BUFF` is not defined) or a channel + shared memory (`ADAT_TX_USE_SHARED_BUFF` is defined). When using a channel + shared memory for samples transfer, it is required that the ADAT transmitter and the `XUA_AudioHub()` tasks run on the same tile.

The USB Audio reference applications with ADAT interface enabled in `sw_usb_audio` are only tested with `ADAT_TX_USE_SHARED_BUFF` defined. The sample transfer sequence described in [Sample communication](#) assumes that `ADAT_TX_USE_SHARED_BUFF` is defined.

Declarations

The channel used for communicating between `XUA_AudioHub` and ADAT transmitter should be declared:

```
chan c_adat_out
```

In order to use the ADAT transmitter with `lib_xua` a 1-bit port must be declared e.g:

```
on tile[XUA_AUDIO_IO_TILE_NUM] : buffered out port:32 p_adat_tx = PORT_ADAT_OUT;
```

This port should be clocked from the master-clock:

```
configure_out_port_no_ready(p_adat_tx, clk_audio_mclk, 0);
set_clock_fall_delay(clk_audio_mclk, 7);
```

Finally the ADAT transmitter task is run - passing in the port and channel for communication with `XUA_AudioHub`:

```
adat_tx_port(c_adat_out, p_adat_tx);
```

Sample communication

Samples are communicated between `XUA_AudioHub()` and the `adat_tx_port()` task.

The interface to the ADAT transmitter task is via a normal channel with streaming builtins (`outuint`, `inuint`).

To begin with, `XUA_AudioHub` sends two values on the channel - the master clock multiplier and the S/MUX setting.

The master clock multiplier is the ratio between the mclk frequency and the sampling frequency. The S/MUX setting is 1, 2 or 4, depending on the sampling frequency:

```
/* Calculate what master clock we should be using */
if (((MCLK_441) % curSamFreq) == 0)
{
    mClk = MCLK_441;
#ifdef (XUA_ADAT_TX_EN)
    /* Calculate ADAT SMUX mode (1, 2, 4) */
    adatSmuxMode = curSamFreq / 44100;
    adatMultiple = mClk / 44100;
#endif
}
else if (((MCLK_48) % curSamFreq) == 0)
{
    mClk = MCLK_48;
#ifdef (XUA_ADAT_TX_EN)
    /* Calculate ADAT SMUX mode (1, 2, 4) */
    adatSmuxMode = curSamFreq / 48000;
```

(continues on next page)



(continued from previous page)

```

        adatMultiple = mClk / 48000;
#ifdef
    }

```

This is followed by communicating the address of a block of memory holding the audio samples block. The **XUA_AudioHub** “runs ahead” of the ADAT transmitter task, assembling the next sample block while the ADAT transmitter converts the current block into an ADAT stream to transmit over the optical interface.

The ADAT transmitter, once done processing the current block, acknowledges this by sending a data token over the channel to **XUA_AudioHub** as a handshake mechanism. On receiving this handshake, **XUA_AudioHub** sends the address of the next block of samples over the channel.

Note that a **XS1_CT_END** control token is not sent between blocks of data, leading to the channel remaining open and getting used as a streaming channel.

XUA_AudioHub only terminates the connection by sending a **XS1_CT_END** token when there's a sampling frequency change that requires the **XUA_AudioHub** task to re-communicate the master clock multiplier and the S/MUX setting.

In case of a sampling frequency change, the **XUA_AudioHub()** task receives the pending handshake from the ADAT transmitter, followed by sending the **XS1_CT_END** token indicating the end of data streaming to the ADAT task. A fresh transmission is then started by sending the new master clock multiplier and the S/MUX setting to the ADAT transmitter, followed by audio blocks transfer as described above.

[Fig. 2](#) describes the communication between **XUA_AudioHub** and the ADAT transmitter:

For further details please see the documentation and examples provided with [lib_adat](#).

Channel count changes

When ADAT transmit is enabled, the number of USB playback channels vary depending on the sampling freq (S/MUX mode). This is exposed to the USB host as alternative interfaces, each supporting different channel counts, for the streaming output interface. The number of alternative interfaces exposed depends on the **MIN_FREQ** and **MAX_FREQ** supported over the USB interface. In the most generic case, where the device supports all sampling rates from 44.1 to 192 kHz, 3 alternative interfaces on the streaming output interface are exposed, each supporting a different channel count.



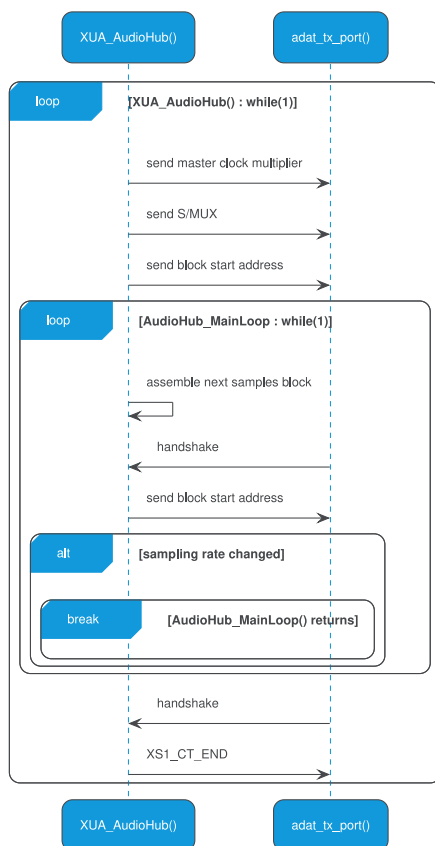


Fig. 2: Communication between XUA_AudioHub and ADAT transmit

6.4 ADAT receive

lib_xua supports the development of devices with ADAT receive functionality through the use of [lib_adat](#). The XMOS ADAT receiver runs on a single thread.

The ADAT receive component receives up to eight channels of audio at a sample rate of 44.1kHz or 48kHz. The API for calling the receiver functions is described in the [ADAT receive example in lib_adat](#).

The component outputs 32 bits words split into nine word frames. The frames are laid out in the following manner:

- Control byte
- Channel 0 sample
- Channel 1 sample
- Channel 2 sample
- Channel 3 sample



- ▶ Channel 4 sample
- ▶ Channel 5 sample
- ▶ Channel 6 sample
- ▶ Channel 7 sample

An example of how to read the output of the ADAT component is shown below:

```
control = inuint(oChan);  
for(int i = 0; i < 8; i++)  
{  
    sample[i] = inuint(oChan);  
}
```

Samples are 24-bit values contained in the lower 24 bits of the word.

The control word comprises four control bits in bits [11..8] and the value 0b00000001 in bits [7..0]. This control word enables synchronization at a higher level, in that, on the channel, a single odd word is always read followed by eight words of data.

Usage and integration

Since the ADAT is a digital stream the device's master clock must be synchronised to it. The integration of ADAT receive is much the same as S/PDIF receive in that the ADAT receive function communicates with the *Clock Gen* thread. This *Clock Gen* thread then passes audio data onto the Audio Hub thread. It also handles locking to the ADAT clock source.

There are some small differences with the S/PDIF integration accounting for the fact that ADAT typically has 8 channels compared to S/PDIF's two.

The *Clock Gen* thread also handles SMUX II (e.g. 4 channels at 96kHz) and SMUX IV (e.g. 2 channels at 192 kHz), populating the sample FIFO as appropriate. SMUX modes are communicated to the *Clock Gen* thread from Endpoint 0 via the `c_clk_ctl` channel. SMUX modes are exposed to the USB host using Alternative Interfaces, with appropriate channel counts, for the streaming input Endpoint.



7 Implementation detail

This chapter examines the implementation of the various components that make up **lib_xua**. It also examines the integration of dependencies and supporting libraries.



7.1 Audio Hub and I²S

The **AudioHub** task performs many functions. It receives and transmits samples from/to the Decoupler or Mixer thread over a channel.

It also drives several in and out I²S/TDM channels to/from a CODEC, DAC, ADC etc. From now on these external devices will be termed "audio hardware".

If the firmware is configured with the *xcore* as I²S master the required clock lines will also be driven from this task. It also has the task of forwarding on and receiving samples to/from other audio related tasks/threads such as S/PDIF tasks, ADAT etc.

In master mode, the *xcore* generates the I²S "Continuous Serial Clock" (SCK), or "Bit-Clock" (BCLK) and the "Word Select" (WS) or "Left-Right Clock" (LRCLK) signals. Any CODEC or DAC/ADC combination that supports I²S and can be used.

The LR-clock, bit-clock and data are all derived from the incoming master clock (typically the output of the external oscillator or PLL). This is not part of the I²S standard but is commonly included for synchronizing the internal operation of the analog/digital converters.

The Audio Hub task is implemented in the file `xua_audiohub.xc`.

[Table 30](#) shows the signals used to communicate audio between the XMOS device and the external audio hardware.

Table 30: I²S Signals

Signal	Description
LRCLK	The word clock, transition at the start of a sample
BCLK	The bit clock, clocks data in and out
SDIN	Sample data in (from CODEC/ADC to the XMOS device)
SDOUT	Sample data out (from the XMOS device to CODEC/DAC)
MCLK	The master clock running the CODEC/DAC/ADC

The bit clock controls the rate at which data is transmitted to and from the external audio hardware.

In the case where the *xcore* is the master, it divides the MCLK to generate the required signals for both BCLK and LRCLK, with BCLK then being used to clock data in (SDIN) and data out (SDOUT) of the external audio hardware.

[Table 31](#) shows some example clock frequencies and divides for different sample rates:

Table 31: Clock Divide examples

Sample Rate (kHz)	MCLK (MHz)	BCLK (MHz)	Divide
44.1	11.2896	2.819	4
88.2	11.2896	5.638	2
176.4	11.2896	11.2896	1
48	24.576	3.072	8
96	24.576	6.144	4
192	24.576	12.288	2



For *xcore-200* devices the master clock must be supplied by an external source e.g. clock generator, fixed oscillators, PLL etc. *xcore.ai* devices may use the integrated secondary PLL.

Two master clock frequencies are required to support 44.1 kHz and 48 kHz audio frequencies (e.g. 11.2896/22.5792MHz and 12.288/24.576MHz respectively). This master clock input is then provided to the external audio hardware and the *xcore* device.

Port configuration (xcore master)

The default software configuration is of *xcore* being the I²S master. That is, the *xcore* device provides the BCLK and LRCLK signals to the external audio hardware

xcore ports and clocks provide many valuable features for implementing I²S. This section describes how these are configured and used to drive the I²S interface.

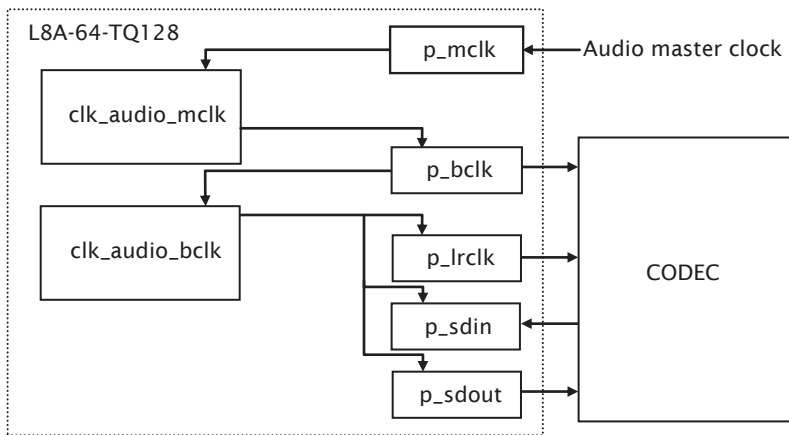


Fig. 3: Ports and clocks (xcore master)

The code to configure the ports and clocks is in the `ConfigAudioPorts()` function. Developers should not need to modify this.

The *xcore* inputs MCLK and divides it down to generate BCLK and LRCLK.

To achieve this MCLK is input into the device using the 1-bit port **p_mclk**. This is attached to the clock block **clk_audio_mclk**, which is in turn used to clock the BCLK port, **p_bclk**. BCLK is used to clock the LRCLK (**p_lrclk**) and data signals SDIN (**p_sdin**) and SDOUT (**p_sdout**).

Again, a clock block is used (**clk_audio_bclk**) which has **p_bclk** as its input and is used to clock the ports **p_lrclk**, **p_sdin** and **p_sdout**. Fig. 3 shows the connectivity of ports and clock blocks.

p_sdin and **p_sdout** are configured as buffered ports with a transfer width of 32, so all 32 bits are input in one input statement. This allows the software to input, process and output 32-bit words, whilst the ports serialize and deserialize to the single I/O pin connected to each port.

Unlike previous *xcore* devices, *xcore-200* (XS2 architecture) and *xcore.ai* (XS3 architecture) series devices have the ability to divide an external clock in a clock-block.



The bit clock outputs 32 clock cycles per sample. In the special case where the divide is 1 (i.e. the bit clock frequency equals the master clock frequency), the `p_bclk` port is set to a special mode where it simply outputs its clock input (i.e. `p_mclk`). See `configure_port_clock_output()` in `xs1.h` for details.

`p_lrclk` is clocked by `p_bclk`. In I²S mode the port outputs the pattern `0x7fffffff` followed by `0x80000000` repeatedly. This gives a signal that has a transition one bit-clock before the data (as required by the I²S standard) and alternates between high and low for the left and right channels of audio.

Changing audio sample frequency

When the host changes sample frequency, a new frequency is sent to the audio driver thread by Endpoint 0 (via the buffering threads and mixer).

First, a change of sample frequency is reported by sending the new frequency over an XC channel. The audio thread detects this by checking for the presence of a control token on the channel channel

Upon receiving the change of sample frequency request, the audio thread stops the I²S/TDM interface and calls the CODEC/port configuration functions.

Once this is complete, the I²S/TDM interface (i.e. the main loop in `AudioHub`) is restarted at the new frequency.



7.2 Endpoint 0: Management and control

All USB devices must support a mandatory control endpoint, Endpoint 0. This controls the management tasks of the USB device.

These tasks can be generally split into enumeration, audio configuration and firmware upgrade requests.

Enumeration

When the device is first attached to a host, enumeration occurs. This process involves the host interrogating the device as to its functionality. The device does this by presenting several interfaces to the host via a set of descriptors.

During the enumeration process the host will issue various commands to the device including assigning the device a unique address on the bus.

The endpoint 0 code runs in its own thread and follows a similar format to that of the USB HID Mouse Device examples in [lib_xud](#). That is, a call is made to `USB_GetSetupPacket()` to receive a command from the host. This populates a `USB_SetupPacket_t` structure, which is then parsed.

There are many mandatory requests that a USB Device must support as required by the USB Specification. Since these are required for all devices in order to function a `USB_StandardRequests()` function is provided (see `xud_device.xc`) which implements all of these requests. This includes the following items:

- ▶ Requests for standard descriptors (Device descriptor, configuration descriptor etc) and string descriptors
- ▶ USB GET/SET INTERFACE requests
- ▶ USB GET/SET_CONFIGURATION requests
- ▶ USB SET_ADDRESS requests

For more information and full documentation, including full worked examples of simple devices, refer to [lib_xud](#).

The `USB_StandardRequests()` function takes the device's various descriptors as parameters. These are passed from data structures found in the `xud_ep0_descriptors.h` file. These data structures are fully customised based on how the design is configured using various defines.

The `USB_StandardRequests()` function returns a `XUD_Result_t`. `XUD_RESULT_OKAY` to indicate that the request was fully handled without error and no further action is required - The device should move to receiving the next request from the host (via `USB_GetSetupPacket()`).

The function returns `XUD_RES_ERR` if the request was not recognised by the `USB_StandardRequests()` function and a `STALL` has been issued.

The function may also return `XUD_RES_RST` if a bus-reset has been issued onto the bus by the host and communicated from XUD to Endpoint 0.

Since the `USB_StandardRequests()` function STALLs an unknown request, the endpoint 0 code must first parse the `USB_SetupPacket_t` structure to handle device specific requests and then call `USB_StandardRequests()` as required.



Overriding standard requests

The USB Audio design “overrides” some of the requests handled by `USB_StandardRequests()`, for example it uses the `SET_INTERFACE` request to indicate if the host is streaming audio to the device. In this case the setup packet is parsed, the relevant action taken, the `USB_StandardRequests()` is still called to handle the response to the host.

Class requests

Before making the call to `USB_StandardRequests()` the setup packet is parsed for Class requests. These are handled in functions such as `AudioClassRequests_1()`, `AudioClassRequests_2`, `dfu_usb_class_int_requests()` etc depending on the type of request.

Any device specific requests are handled - in this case Audio Class, MIDI class, DFU requests etc.

Some of the common Audio Class requests and their associated behaviour will now be examined.

Audio requests When the host issues an audio request (e.g. sample rate or volume change), it sends a command to Endpoint 0. Like all requests this is returned from `USB_GetSetupPacket()`. After some parsing (namely as Class Request to an Audio Interface) the request is handled by either the `AudioClassRequests_1()` or `AudioClassRequests_2()` function (based on whether the device is running in Audio Class 1.0 or 2.0 mode).

Note, Audio Class 1.0 Sample rate changes are send to the relevant endpoint, rather than the interface - this is handled as a special case in the endpoint 0 request parsing where `AudioEndpointRequests_1()` is called.

The `AudioClassRequests_X()` functions further parses the request in order to ascertain the correct audio operation to execute.

Audio request: Set sample rate The `AudioClassRequests_2()` function parses the passed `USB_SetupPacket_t` structure for a CUR request of type `SAM_FREQ_CONTROL` to a Clock Unit in the device's topology (as described in the device descriptors).

The new sample frequency is extracted and passed via a channel to the rest of the design - through the buffering code and eventually to the Audio Hub (I²S) thread. The `AudioClassRequests_2()` function waits for a handshake to propagate back through the system before signalling to the host that the request has completed successfully. Note, during this time the USB library is NAKing the host, essentially holding off further traffic/requests until the sample-rate change is fully complete.

Audio Request: Volume control When the host requests a volume change, it sends an audio interface request to Endpoint 0. An array is maintained in the Endpoint 0 thread that is updated with such a request.

When changing the volume, Endpoint 0 applies the master volume and channel volume, producing a single volume value for each channel. These are stored in the array.

The volume will either be handled by the `decouple` thread or the `mixer` component (if the mixer component is used). Handling the volume in the mixer gives the decoupler more performance to handle more channels.



If the effect of the volume control array on the audio input and output is implemented by the decoupler, the **decoupler** thread reads the volume values from this array. Note that this array is shared between Endpoint 0 and the decoupler thread. This is done in a safe manner, since only Endpoint 0 can write to the array, word update is atomic between threads and the decoupler thread only reads from the array (ordering between writes and reads is unimportant in this case). Inline assembly is used by the decoupler thread to access the array, avoiding the parallel usage checks of XC.

If volume control is implemented in the mixer, Endpoint 0 sends a mixer command to the mixer to change the volume. Mixer commands are described in [Digital mixer](#).

7.3 Audio endpoints (Endpoint Buffer and Decoupler)

Endpoint Buffer

All endpoints other than Endpoint 0 are handled in one thread. This thread is implemented in the file **ep_buffer.xc**. This thread communicates directly with the XUD library.

The USB buffer thread is also responsible for feedback calculation based on USB Start Of Frame (SOF) notification and reads from the port counter of a port connected to the master clock.

Decouple

The decoupler supplies the USB buffering thread with buffers to transmit/receive audio data to/from the host. It marshals these buffers into FIFOs. The data from the FIFOs is then sent over XC channels to other parts of the system as they need it. In asynchronous mode this thread also determines the size of each packet of audio to send to the host (thus matching the audio rate to the USB packet rate). The decoupler is implemented in the file **decouple.xc**.

Audio buffering scheme

This scheme is executed by co-operation between the buffering thread, the decouple thread and the XUD library.

For data going from the device to the host the following scheme is used:

1. The Decouple thread receives samples from the Audio Hub thread and puts them into a FIFO. This FIFO is split into packets when data is entered into it. Packets are stored in a format consisting of their length in bytes followed by the data.
2. When the Endpoint Buffer thread needs a buffer to send to the XUD thread (after sending the previous buffer), the Decouple thread is signalled (via a shared memory flag).
3. Upon this signal from the Endpoint Buffer thread, the Decouple thread passes the next packet from the FIFO to the Endpoint Buffer thread. It also signals to the XUD library that the Endpoint Buffer thread is able to send a packet.
4. When the Endpoint Buffer thread has sent this buffer, it signals to the Decouple thread that the buffer has been sent and the Decouple thread moves the read pointer of the FIFO.

For data going from the host to the device the following scheme is used:

1. The Decouple thread passes a pointer to the Endpoint Buffer thread pointing into a FIFO of data and signals to the XUD library that the Endpoint Buffer thread is ready to receive.



2. The Endpoint Buffer thread then reads a USB packet into the FIFO and signals to the Decouple thread that the packet has been read.
3. Upon receiving this signal the Decouple thread updates the write pointer of the FIFO and provides a new pointer to the Endpoint Buffer thread to fill.
4. Upon request from the Audio Hub thread, the Decouple thread sends samples to the Audio Hub thread by reading samples out of the FIFO.

Decoupler/Audio Hub interaction

To meet timing requirements of the audio system (i.e Audio Hub/Mixer), the Decoupler thread must respond to requests from the audio system to send/receive samples immediately. An interrupt handler is set up in the decoupler thread to do this. The interrupt handler is implemented in the function `handle_audio_request`.

The audio system sends a word over a channel to the decouple thread to request sample transfer (using the build in `outuint()` function). The receipt of this word in the channel causes the `handle_audio_request` interrupt to fire.

The first operation the interrupt handler does (once it inputs the word that triggered the interrupt) is to send back a word acknowledging the request (if there was a change of sample frequency a control token would instead be sent—the audio system uses a `testct()` to inspect for this case).

Sample transfer may now take place. First the Decouple thread sends samples from host to device then the audio subsystem transfers samples destined for the host. These transfers always take place in channel count sized chunks (i.e. `NUM_USB_CHAN_OUT` and `NUM_USB_CHAN_IN`). That is, if the device has 10 output channels and 8 input channels, 10 samples are sent from the decouple thread and 8 received every interrupt.

The complete communication scheme is shown in [Table 32](#) (for non sample frequency change case).

Table 32: Decouple/Audio system channel communication

Decouple	Audio System	Note
	<code>outuint()</code>	Audio system requests sample exchange
<code>inuint()</code>		Interrupt fires and <code>inuint</code> performed
<code>outuint()</code>		Decouple sends ack
	<code>testct()</code>	Checks for CT indicating SF change
	<code>inuint()</code>	Word indication ACK input (No SF change)
<code>inuint()</code>	<code>outuint()</code>	Sample transfer (Device to Host)
<code>inuint()</code>	<code>outuint()</code>	
<code>inuint()</code>	<code>outuint()</code>	
...		
<code>outuint()</code>	<code>inuint()</code>	Sample transfer (Host to Device)
<code>outuint()</code>	<code>inuint()</code>	
<code>outuint()</code>	<code>inuint()</code>	
<code>outuint()</code>	<code>inuint()</code>	
...		



Note

The request and acknowledgement sent to/from the Decouple thread to the Audio System is an “output underflow” sample value. If in PCM mode it will be 0, in DSD mode it will be DSD silence. This allows the buffering system to output a suitable underflow value without knowing the format of the stream (this is especially advantageous in the DSD over PCM (DoP) case)

Asynchronous feedback

When built to operate in Asynchronous mode the device uses a feedback endpoint to report the rate at which audio is output/input to/from external audio interfaces/devices. This feedback is in accordance with the *USB 2.0 Specification*. This calculated feedback value is also used to size packets to the host.

This asynchronous clocking scheme means that the device is the clock master and therefore a high-quality local master clock or a digital input stream can be used as the clock source.

After each received USB Start Of Frame (SOF) token, the buffering thread takes a time-stamp from a port clocked off the master clock. By subtracting the time-stamp taken at the previous SOF, the number of master clock ticks since the last SOF is calculated. From this the number of samples (as a fixed point number) between SOFs can be calculated. This count is aggregated over 128 SOFs and used as a basis for the feedback value.

The sending of feedback to the host is also handled in the Endpoint Buffer thread via an explicit feedback IN endpoint.

If both input and output is enabled then the feedback can be implicit based on the audio stream sent to the host. In practice though an explicit feedback endpoint is normally used due to restrictions in Microsoft Windows operating systems (see `UAC_FORCE_FEEDBACK_EP`).

USB rate control

The device must consume data from USB host and provide data to USB host at the correct rate for the selected sample frequency. When running in asynchronous mode the *USB 2.0 Specification* states that the maximum variation on USB packets can be +/- 1 sample per USB frame (Synchronous mode mandates no variation other than that required to match a sample rate that doesn't cleanly divide the USB SOF period e.g. 44.1kHz)

High-speed USB frames are sent at 8kHz, so on average for 48kHz each packet contains six samples per channel.

When running in Asynchronous mode, the audio clock may drift and run faster or slower than the host. Hence, if the audio clock is slightly fast, the device may occasionally input/output seven samples rather than six. Alternatively, it may be slightly slow and input/output five samples rather than six. [Allowed samples per packet in Async mode](#) shows the allowed number of samples per packet for each example audio frequency in Asynchronous mode.

When running in Synchronous mode the audio clock is synchronised to the USB host SOF clock. Hence, at 48kHz the device always expects six samples from, and always sends six samples to the host.

See [USB Device Class Definition for Audio Data Formats v2.0](#) section 2.3.1.1 for full details.



Table 33: Allowed samples per packet in Async mode

Frequency (kHz)	Min Packet	Max Packet
44.1	5	6
48	5	7
88.2	10	11
96	11	13
176.4	20	21
192	23	25

To implement this control, the Decoupler thread uses the feedback value calculated in the EP Buffering thread. This value is used to work out the size of the next packet it will insert into the audio FIFO.

Note

In Synchronous mode the same system is used, but the feedback value simply uses a fixed value rather than one derived from the master clock port.



7.4 XMOS USB Device (XUD) library

All low level communication with the USB host is handled by the XMOS USB Device (XUD) library - [lib_xud](#)

The **XUD_Main()** function runs in its own thread and communicates with endpoint threads through a mixture of shared memory and channel communications.

For more details and full XUD API documentation please refer to [lib_xud](#).

[Fig. 1](#) shows the XUD library communicating with two other threads:

- ▶ Endpoint 0: This thread controls the enumeration/configuration tasks of the USB device.
- ▶ Endpoint Buffer: This thread sends/receives data packets from the XUD library. The thread receives audio data from the AudioHub, MIDI data from the MIDI thread etc.



7.5 External clock recovery (Clock Gen)

To provide an audio master clock an application may use selectable oscillators, clock generation IC or, in the case of *xcore.ai* devices, an integrated secondary PLL, to generate fixed master clock frequencies.

It may also use an external PLL/Clock Multiplier to generate a master clock based on a reference from the *xcore*.

Using the internal secondary PLL on an external PLL/Clock Multiplier allows an Asynchronous mode design to lock to an external clock source from a digital stream (e.g. S/PDIF or ADAT input). **lib_xua** supports the Cirrus Logic CS2100 device or use of [lib_sw_pll](#) (*xcore.ai* only) for this purpose. Other devices may be supported via code modification.

The Clock Recovery thread (Clock Gen) is responsible for either generating the reference frequency to the CS2100 device or driving **lib_sw_pll** from time measurements based on the local master clock and the time of received samples. Clock Gen (via CS2100 or **lib_sw_pll**) generates the master clock used over the whole design. This thread also serves as a smaller buffer between ADAT and S/PDIF receiving threads and the Audio Hub thread.

When using **lib_sw_pll** (*xcore.ai* only) a further thread is instantiated which performs the sigma-delta modulation of the *xcore* PLL to ensure the lowest jitter over the audio band. See [lib_sw_pll](#) documentation for further details.

When running in *Internal Clock* mode this thread simply generates this clock using a local timer, based on the *xcore*'s internal 100 MHz reference clock.

When running in an external clock mode (i.e. S/PDIF Clock" or "ADAT Clock" mode) samples are received from the S/PDIF and/or ADAT receive thread. The external frequency is calculated through counting samples in a given period. Either the reference clock to the CS2100 is then generated based on the reception of these samples or the timing information is provided to **lib_sw_pll** to generate the phase-locked clock on-chip (*xcore.ai* only).

If an external stream becomes invalid, the *Internal Clock* timer event will fire to ensure that valid master clock generation continues regardless of cable unplugs etc. Efforts are made to ensure that the transitions between these clocks are relatively seamless. Additionally efforts are also made to try and keep the jitter on the reference clock as low as possible, regardless of activity level of the Clock Gen thread. This is achieved through the use of port times to schedule pin toggling rather than directly outputting to the port in the case of using the CS2100. For **lib_sw_pll** cases the last setting is kept for the sigma-delta modulator ensuring clock continuity.

The Clock Gen thread gets clock selection Get/Set commands from Endpoint 0 via the **c_clk_ctl** channel. This thread also records the validity of external clocks, which is also queried through the same channel from Endpoint 0. Note, the *Internal Clock* is always reported as being valid. It should be noted that the device always reports the current device sample rate regardless of the clock being interrogated. This results in improved user experience for most driver/operating system combinations

To inform the host of any status change, the Clock Gen thread can also cause the Decouple thread to request an interrupt packet on change of clock validity. This functionality is based on the Audio Class 2.0 status/interrupt endpoint feature.



Note

When running in Synchronous mode external digital input streams are currently not supported. Such a feature would require sample-rate conversion to convert from the S/PDIF or ADAT clock domain to the USB host clock domain. As such this thread is not used in a Synchronous mode device.

7.6 Digital mixer

The Mixer thread(s) take outgoing audio from the Decouple thread and incoming audio from the Audio Hub thread. It then applies the volume to each channel and passes incoming audio on to Decouple and outgoing audio to Audio Hub. The volume update is achieved using the built-in 32bit to 64bit signed multiply-accumulate function (**maccs**). The mixer is implemented in the file **mixer.xc**.

The mixer takes (up to) two threads and can perform eight mixes with up to 18 inputs at sample rates up to 96kHz and two mixes with up to 18 inputs at higher sample rates. The component automatically reverts to generating two mixes when running at the higher rate.

The mixer can take inputs from either:

- ▶ The USB outputs from the host—these samples come from the Decouple thread.
- ▶ The inputs from the audio interfaces on the device—these samples come from the Audio Hub thread and includes samples from digital input streams.

Since the sum of these inputs may be more than the 18 possible mix inputs to each mixer, there is a mapping from all the possible inputs to the mixer inputs.

After the mix occurs, the final outputs are created. There are two possible output destinations for each mix.

- ▶ The USB inputs to the host—these samples are sent to the Decouple thread.
- ▶ The outputs to the audio interface on the device—these samples are sent to the Audio Hub thread

For each possible output from the device, a mapping exists to inform the mixer what its source is. The possible sources are the output from the USB host, the inputs from the Audio Hub thread or the outputs from the mixes.

Essentially the mixer/router can be configured such that any device input can be used as an input to any mix or routed directly to any device output. Additionally, any device output can be derived from any mixer output or any device input.

As mentioned in [Audio Request: Volume control](#), the mixer can also handle processing of volume controls. If the mixer is configured to handle volume but the number of mixes is set to zero (such that the thread is solely doing volume setting) then the component will use only one thread. This is sometimes a useful configuration for large channel count devices since it offloads volume processing from the buffering sub-system.

A sequence diagram showing the communication between Audio Hub, Decouple and mixer threads is shown in [Fig. 4](#). **mixer1** thread exchanges data with Decouple and Audio Hub along with any volume control operations and performs the mixing operations for the even output channel numbers. The mixing for the odd channels is offloaded to the **mixer2** thread.



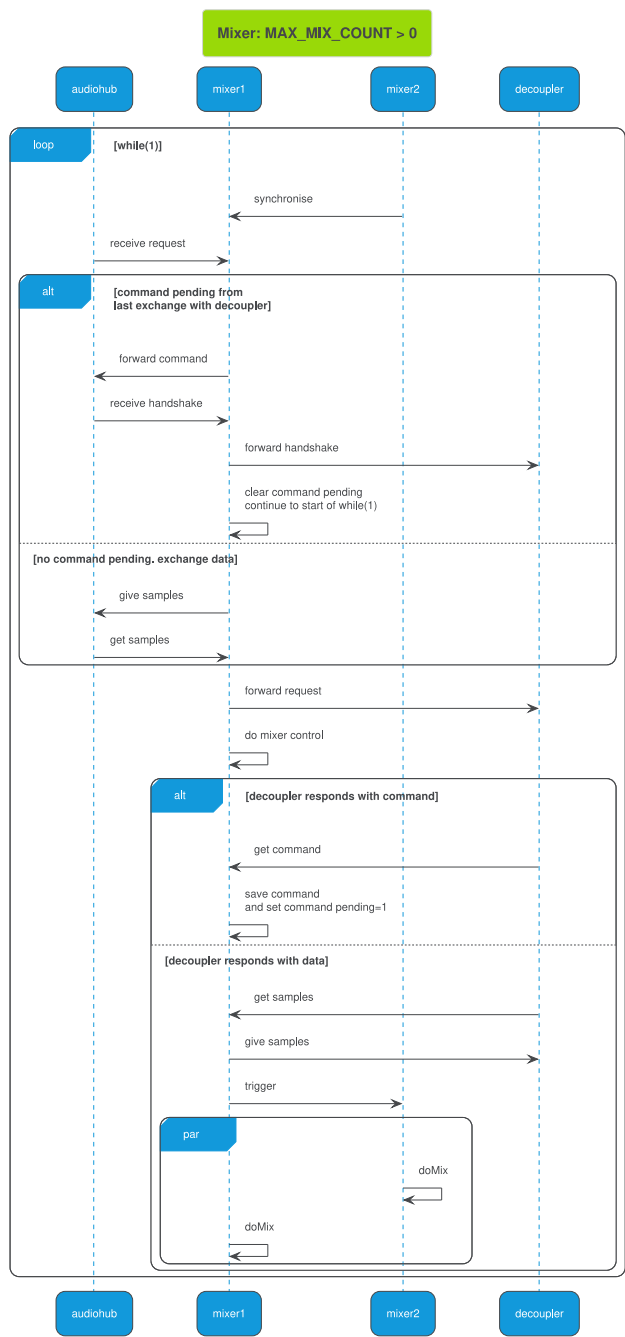


Fig. 4: Mixer communication sequence diagram



The mixer can also be configured in passthrough mode (**MAX_MIX_COUNT** = 0), as shown in Fig. 5. In this mode, the **mixer2** thread is not present and the **mixer1** exchanges data with Audio Hub and Decouple along with any volume control operations without doing any actual mixing.

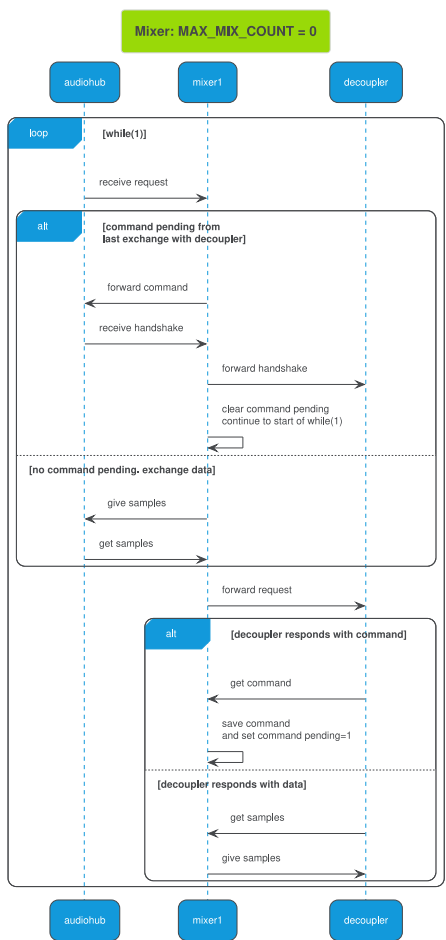


Fig. 5: Mixer in passthrough mode

Control

The mixer tasks can receive the control commands from the host via USB Control Requests to Endpoint 0. The Endpoint 0 thread relays these to the Mixer threads(s) via a channel (**c_mix_ctl**). These commands are described in Table 34.



Table 34: Mixer control commands

Command	Description
SET_SAMPLES_TO_HOST_MAP	Sets the source of one of the audio streams going to the host.
SET_SAMPLES_TO_DEVICE_MAP	Sets the source of one of the audio streams going to the audio driver.
SET_MIX_MULT	Sets the multiplier for one of the inputs to a mixer.
SET_MIX_MAP	Sets the source of one of the inputs to a mixer.
SET_MIX_IN_VOL	If volume adjustment is being done in the mixer, this command sets the volume multiplier of one of the USB audio inputs.
SET_MIX_OUT_VOL	If volume adjustment is being done in the mixer, this command sets the volume multiplier of one of the USB audio outputs.

Host control

The mixer can be controlled from a host PC by sending requests to Endpoint 0. *XMOS* provides a simple command line based sample application demonstrating how the mixer can be controlled. This is intended as an example of how you might add mixer control to your own control application. It is not intended to be exposed to end users.

For details, consult the *README* file in the *host_usb_mixer_control* directory. A list of arguments can also be seen with:

```
$ ./xmos_mixer --help
```

The main requirements of this control utility are to

- Set the mapping of input channels into the mixer
- Set the coefficients for each mixer output for each input
- Set the mapping for physical outputs which can either come directly from the inputs or via the mixer.

Note

The flexibility within this configuration space is such that there are often multiple ways of producing the desired result. Product developers may only want to expose a subset of this functionality to their end users.

Whilst using the *XMOS* host control example application, consider the example of setting the mixer to perform a loop-back from analogue inputs 1 & 2 to analogue outputs 1 & 2.

Note

The command outputs shown are examples; the actual output will depend on the mixer configuration.

The following will show the index for each device output along with which channel is currently mapped to it. In this example the analogue outputs 1 & 2 are 0 & 1 respectively:



```
$ ./xmos_mixer --display-aud-channel-map

Audio Output Channel Map
-----
0 (DEVICE OUT - Analogue 1) source is 0 (DAW OUT - Analogue 1)
1 (DEVICE OUT - Analogue 2) source is 1 (DAW OUT - Analogue 2)
2 (DEVICE OUT - SPDIF 1) source is 2 (DAW OUT - SPDIF 1)
3 (DEVICE OUT - SPDIF 2) source is 3 (DAW OUT - SPDIF 2)
$ _
```

The DAW Output Map can be seen with:

```
$ ./xmos_mixer --display-daw-channel-map

DAW Output To Host Channel Map
-----
0 (DEVICE IN - Analogue 1) source is 4 (DEVICE IN - Analogue 1)
1 (DEVICE IN - Analogue 2) source is 5 (DEVICE IN - Analogue 2)
$ _
```

Note

In both cases, by default, these bypass the mixer.

The following command will list the channels which can be mapped to the device outputs from the Audio Output Channel Map. Note that, in this example, analogue inputs 1 & 2 are source 4 & 5 and Mix 1 & 2 are source 6 & 7:

```
$ ./xmos_mixer --display-aud-channel-map-sources

Audio Output Channel Map Source List
-----
0 (DAW OUT - Analogue 1)
1 (DAW OUT - Analogue 2)
2 (DAW OUT - SPDIF 1)
3 (DAW OUT - SPDIF 2)
4 (DEVICE IN - Analogue 1)
5 (DEVICE IN - Analogue 2)
6 (MIX - Mix 1)
7 (MIX - Mix 2)
$ _
```

Using the indices from the previous commands, we will now re-map the first two mixer channels (Mix 1 & Mix 2) to device outputs 1 & 2:

```
$ ./xmos_mixer --set-aud-channel-map 0 6
$ ./xmos_mixer --set-aud-channel-map 1 7
$ _
```

The effect of this can be confirmed by re-checking the map:

```
$ ./xmos_mixer --display-aud-channel-map

Audio Output Channel Map
-----
0 (DEVICE OUT - Analogue 1) source is 6 (MIX - Mix 1)
1 (DEVICE OUT - Analogue 2) source is 7 (MIX - Mix 2)
2 (DEVICE OUT - SPDIF 1) source is 2 (DAW OUT - SPDIF 1)
3 (DEVICE OUT - SPDIF 2) source is 3 (DAW OUT - SPDIF 2)
$ _
```

Analogue outputs 1 & 2 are now derived from the mixer, rather than directly from USB. However, since the mixer is mapped, by default, to just pass the USB channels through to the outputs no functional change will be observed.

The mixer nodes need to be individually set. The nodes in `mixer_id 0` can be displayed with the following command:



```
$ ./xmos_mixer --display-mixer-nodes 0

Mixer Values (0)
-----

Mixer outputs
      1      2
DAW - Analogue 1  0:[0000.000] 1:[ -inf ]
DAW - Analogue 2  2:[ -inf ] 3:[0000.000]
DAW - SPDIF 1     4:[ -inf ] 5:[ -inf ]
DAW - SPDIF 2     6:[ -inf ] 7:[ -inf ]
AUD - Analogue 1  8:[ -inf ] 9:[ -inf ]
AUD - Analogue 2 10:[ -inf ] 11:[ -inf ]
$ -
```

Note

The USB audio reference design has only one unit so the `mixer_id` argument should always be 0.

With mixer outputs 1 & 2 mapped to device outputs analogue 1 & 2; to get the audio from the analogue inputs to device outputs mixer_id 0 node 8 and node 11 need to be set to 0db:

```
$ ./xmos_mixer --set-value 0 8 0
$ ./xmos_mixer --set-value 0 11 0
$ -
```

At the same time, the original mixer outputs can be muted:

```
$ ./xmos_mixer --set-value 0 0 -inf
$ ./xmos_mixer --set-value 0 3 -inf
$ -
```

Now audio inputs on analogue 1 and 2 should be heard on outputs 1 and 2 respectively.

As mentioned above, the flexibility of the mixer is such that there will be multiple ways to create a particular mix. Another option to create the same routing would be to change the mixer sources such that mixer outputs 1 and 2 come from the analogue inputs 1 and 2.

To demonstrate this, the changes documented above should be undone (resetting the device will yield the same result):

```
$ ./xmos_mixer --set-value 0 8 -inf
$ ./xmos_mixer --set-value 0 11 -inf
$ ./xmos_mixer --set-value 0 0 0
$ ./xmos_mixer --set-value 0 3 0
$ -
```

The mixer should now have the default values. The sources for mixer 0 output 1 and 2 can now be changed using indices from the *Audio Output Channel Map Source* list:

```
$ ./xmos_mixer --set-mixer-source 0 0 4
Set mixer(0) input 0 to device input 4 (AUD - Analogue 1)
$ ./xmos_mixer --set-mixer-source 0 1 5
Set mixer(0) input 1 to device input 5 (AUD - Analogue 2)
$ -
```

Re-running the following command will show that the first column now has "AUD - Analogue 1 and 2" rather than "DAW (Digital Audio Workstation i.e. the host) - Analogue 1 and 2" confirming the new mapping. Again, by playing audio into analogue inputs 1/2 this can be heard looped through to analogue outputs 1/2:




```
$ ./xmos_mixer --display-mixer-nodes 0
```



7.7 S/PDIF transmit

lib_xua supports the development of devices with S/PDIF transmit through the use of [lib_spdif](#). The *XMOS* S/SPDIF transmitter component runs in a single thread and supports sample-rates upto 192 kHz.

The S/PDIF transmitter thread takes PCM audio samples via a channel and outputs them in S/PDIF format to a port. A lookup table is used to encode the audio data into the required format.

It receives samples from the Audio I/O thread two at a time (for left and right). For each sample, it performs a lookup on each byte, generating 16 bits of encoded data which it outputs to a port.

S/PDIF sends data in frames, each containing 192 samples of the left and right channels.

Audio samples are encapsulated into S/PDIF words (adding preamble, parity, channel status and validity bits) and transmitted in biphase-mark encoding (BMC) with respect to an *external* master clock.

Table 35: S/PDIF capabilities

Sample frequencies	44.1, 48, 88.2, 96, 176.4, 192 kHz
Master clock ratios	128x, 256x, 512x
Library	lib_spdif

Clocking

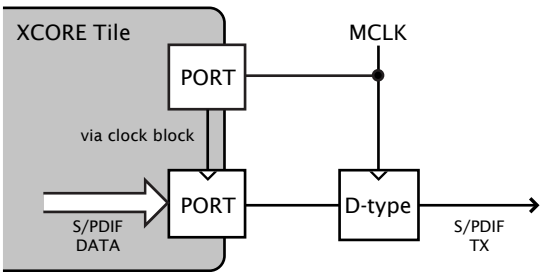


Fig. 6: D-Type Jitter Reduction

The S/PDIF signal is output at a rate dictated by the external master clock. The master clock must be 1x 2x or 4x the BMC bit rate (that is 128x 256x or 512x audio sample rate, respectively). For example, the minimum master clock frequency for 192kHz is therefore 24.576 MHz.

This resamples the master clock to its clock domain (oscillator), which introduces jitter of 2.5-5 ns on the S/PDIF signal. A typical jitter-reduction scheme is an external D-type flip-flop clocked from the master clock (as shown in [Fig. 6](#)).

Usage

The interface to the S/PDIF transmitter thread is via a normal channel using streaming builtin functions (**outuint**, **inuint**). Data format should be 24-bit left-aligned in a 32-bit word: **0x12345600**

The following protocol is used on the channel:



Table 36: S/PDIF Component Protocol

outct	New sample rate command
outuint	Sample frequency (Hz)
outuint	Master clock frequency (Hz)
outuint	Left sample
outuint	Right sample
outuint	Left sample
outuint	Right sample
...	
...	

This communication is wrapped up in the API functions provided by **lib_spdif**.

Output stream structure

The stream is composed of words with the structure shown in [Table 37](#). The channel status bits are 0x0nc07A4, where c=1 for left channel, c=2 for right channel and n indicates sampling frequency as shown in [Table 38](#).

Table 37: S/PDIF stream structure

Bits		
0:3	Preamble	Correct B M W order, starting at sample 0
4:27	Audio sample	Top 24 bits of given word
28	Validity bit	Always 0
29	Subcode data (user bits)	Unused, set to 0
30	Channel status	See below
31	Parity	Correct parity across bits 4:30

Table 38: Channel status bits

Frequency (kHz)	n
44.1	0x0
48	0x2
88.2	0x8
96	0xA
176.4	0xC
192	0xE

7.8 S/PDIF receive

xcore devices can support S/PDIF receive up to 192 kHz - see [lib_spdif](#) for full specifications.

The S/PDIF receiver module uses a clock-block and a buffered one-bit port. The clock-block is divided off a 100 MHz reference clock. The one bit port is buffered to 32 bits. The receiver code uses this clock to over sample the input data.



The receiver outputs audio samples over a *streaming channel end* where data can be input using the built-in input operator. **lib_spdif** also provides API functions that wrap up this communication.

The S/PDIF receive function never returns. The 32-bit value from the channel input comprises of fields shown in [Table 39](#).

Table 39: S/PDIF receive word structure

Bits	
0:3	A tag (see below)
4:28	PCM encoded sample value
29:31	User bits (parity, etc)

The tag has one of three values, as shown in [Table 40](#).

Table 40: S/PDIF receive tags

Tag	Meaning
FRAME_X	Sample on channel 0 (Left for stereo)
FRAME_Y	Sample on another channel (Right if for stereo)
FRAME_Z	Sample on channel 0 (Left), and the first sample of a frame; can be used if the user bits need to be reconstructed.

See S/PDIF, IEC 60958-3:2006, specification for further details on format, user bits etc.

Usage and integration

Since S/PDIF is a digital stream, the device's master clock must be synchronised to it. This is typically done with an external device or the *xcvra* secondary PLL. See [External clock recovery \(Clock Gen\)](#).

Note

Due to the requirement for this clock recovery S/PDIF receive can only be used in Asynchronous mode.

The S/PDIF receive function communicates with the Clock Gen thread, which in turn passes audio data to the Audio Hub thread. The Clock Gen thread also handles locking to the S/PDIF clock source. Again, see [External clock recovery \(Clock Gen\)](#).

The parity of each word/sample received is checked. This is done using the **spdif_rx_check_parity()** function provided by **lib_spdif**:

```
/* Check parity and ignore if bad */
if(spdif_rx_check_parity(spdifRxData))
    continue;
```

If bad parity is detected the word/sample is ignored, otherwise the tag is inspected for channel (i.e. left or right) and the sample stored.



The following code snippet illustrates how the output of the S/PDIF receive component is fundamentally used. Note the use of helper defines/macros for frame identification and sample data extraction, provided by **lib_spdif**:

```
while(1)
{
    c_spdif_rx := data;

    if(spdif_check_parity(data)
        continue;

    tag = data & SPDIF_RX_PREAMBLE_MASK;

    /* Extract 24bit audio sample */
    sample = SPDIF_RX_EXTRACT_SAMPLE(data);

    switch(tag)
    {
        case SPDIF_FRAME_X:
        case SPDIF_FRAME_X:
            // Store left sample
            break;

        case SPDIF_FRAME_Z:
            // Store right sample
            break;
    }
}
```

The Clock Gen thread stores samples in a small FIFO before they are communicated to the Audio Hub thread.



7.9 MIDI

The MIDI thread implements a 31250 baud UART (8-N-1) for both input and output. It uses a single dedicated thread which performs multiple functions:

- ▶ UART transmit (tx) peripheral.
- ▶ UART transmit FIFO of 1024 bytes (may be configured by the user).
- ▶ Decoding of USB MIDI message to bytes.
- ▶ UART receive (rx) peripheral.
- ▶ Packing of received MIDI bytes into USB MIDI messages/events.

It is connected via a channel to the Endpoint Buffer thread meaning that it can be placed on any *xcore* tile in the system subject to resource availability.

The Endpoint Buffer thread implements the two Bulk endpoints (one In and one Out) as well as interacting with small, shared-memory, FIFOs for each endpoint.

On receiving 32-bit USB MIDI events from the Endpoint Buffer thread over the channel, the MIDI thread parses these and translates them to 8-bit MIDI messages which are sent out over the UART. Up to 1024 bytes may be buffered by the MIDI task for outgoing messages in the default configuration. If the outgoing buffer is full then it will cause the USB endpoint to be NACKed which provides flow control in the case that the host application sends messages faster than the UART can transmit them. This is important because the USB bandwidth far exceeds the MIDI UART bandwidth by many orders of magnitude. The combination of buffering and flow control ensures outgoing messages are not dropped during normal operation.

Incoming 8-bit MIDI messages from the UART receiver are packed into 32-bit USB MIDI events and passed on to the Endpoint Buffer thread. Since the rate of ingress to the MIDI port is tiny in comparison to the host USB bandwidth, no buffering is required in the MIDI thread and the MIDI events are always forwarded on directly to USB immediately.

All MIDI message types are supported including Sysex (MIDI System Exclusive) strings allowing custom function such as bank updates and patches, backup and device firmware upgrade (DFU) where supported by the MIDI device.

The MIDI thread is implemented in the file `usb_midi.xc` and the USB buffering is handled in the file `ep_buffer.xc`.



7.10 PDM microphones

lib_xua is capable of integrating with PDM microphones. The PDM stream from the microphones is converted to PCM and output to the host via USB.

Interfacing to the PDM microphones is done using the *XMOS* microphone array library ([lib_mic_array](#)). **lib_mic_array** is designed to allow interfacing to PDM microphones coupled to efficient decimation filters at a user configurable output sample rate.

Note

The **lib_mic_array** library is only available for *xcore.ai* series devices since it uses the Vector Processing Unit of the XS3 architecture.

Up to eight PDM microphones can be attached to the PDM interface (**mic_array_task()**) but it is possible to extend this.

After PDM capture and decimation to the output sample-rate various other steps take place e.g. DC offset elimination etc. Please refer to the documentation provided with [lib_mic_array](#) for further implementation detail and a complete feature set.

By default the sample rates supported are 16 kHz, 32 kHz and 48 kHz although other rates are supportable with some modifications.

Please see [AN00248 Using lib_xua with lib_mic_array](#) for a practical example of this feature.

Hardware characteristics

The PDM microphones require a *clock input* and provide the PDM signal on a *data output*. All of the PDM microphones must share the same clock signal (buffered on the PCB as appropriate), and output onto the data wire(s) that are connected to the capture port. The lines required to interface with PDM microphones are listed in [Table 41](#).

Table 41: PDM microphone data and signal wires

Signal	Description
CLOCK	The PDM clock the used by the microphones to drive the data out.
DQ-PDM	The data from the PDM microphones on the capture port.

Note

The clocking for PDM microphones may be single data rate (one microphone per pin) or double data rate (two microphones per pin clocking on alternate edges). By default **lib_xua** assumes double data rate which provides more efficient port usage.

When initialising the mic array, a **pdm_rx_resources_t** structure is passed to **mic_array_init()**. This contains the hardware resource IDs required for PDM capture:



```

unsigned mClk = MCLK_48, pdmClk = 3072000;
#if (XUA_PDM_MIC_USE_DDR)
    pdm_rx_resources_t pdm_res = PDM_RX_RESOURCES_DDR(
        PORT_MCLK_IN,
        PORT_PDM_CLK,
        PORT_PDM_DATA,
        mClk,
        pdmClk,
        XST_CLKBLK_1,
        XST_CLKBLK_2);
#else
    pdm_rx_resources_t pdm_res = PDM_RX_RESOURCES_SDR(
        PORT_MCLK_IN,
        PORT_PDM_CLK,
        PORT_PDM_DATA,
        mClk,
        pdmClk,
        XST_CLKBLK_1);
#endif
mic_array_init(&pdm_res, NULL, mic_samp_rate);

```

For full details of the `pdm_rx_resources_t` structure and mic array initialisation, please refer to the [lib_mic_array documentation](#).

Usage & integration

Mic array task A PDM microphone wrapper, `mic_array_task()`, is called from `main()`. It takes one channel argument connecting it to the rest of the system, and a `channel_map` specifying the mapping from PDM input pins to microphone output channels:

```

on stdcore[XUA_MIC_PDM_TILE_NUM]:
{
    for(int i=0; i<XUA_NUM_PDM_MICS; i++) {
        mic_array_channel_map[i] = i;
    }
    mic_array_task(c_pdm_pcm, mic_array_channel_map);
}

```

The implementation of `mic_array_task()` can be found in the file `mic_array_task.c`. It typically takes one hardware thread.

`mic_array_task()` runs a `while(1)` loop. At the start of each iteration, it calls `xua_user_pdm_init()`, which is an optional user-provided function that may be used to update `channel_map` if a non-1:1 mapping between PDM input pins and microphone output channels is required.

It then waits to receive the current sampling rate over the `c_pdm_pcm` channel. Once received, it initialises the mic array for the requested sampling rate (using `mic_array_init()`) and then starts the mic array thread(s) via `mic_array_start()`. `mic_array_start()` launches either one or two hardware threads, depending on the value of `XUA_PDM_MIC_USE_PDM_ISR`.

The `c_pdm_pcm` channel is passed to the mic array decimator thread, over which it sends decimated PCM frames to `XUA_AudioHub()`.

Note, it is assumed that the system shares a global master-clock, therefore no additional buffering or rate-matching/conversion is required. This ensures the PDM subsystem and XUA Audiohub are synchronous.

Receiving PCM samples `XUA_AudioHub`'s main IO loop (`AudioHub_MainLoop()`) calls the mic array function `ma_frame_rx()` using the other end of the channel passed to the mic array thread to receive PCM frames from the mic array:

```

ma_frame_rx(mic_samps_base_addr, c_m2a, MIC_ARRAY_CONFIG_SAMPLES_PER_FRAME, MIC_ARRAY_CONFIG_MIC_COUNT);

```

Note that the interface between mic array and xua is sample based - so the only supported value of `MIC_ARRAY_CONFIG_SAMPLES_PER_FRAME` is 1.



Restarting the mic array If `AudioHub_MainLoop()` function exits, the mic array is shut down by calling `ma_shutdown()` from `XUA_AudioHub()`:

```
ma_shutdown((chanend_t)c_pdm_in); // shutdown mics
```

Note, that the channel passed to `ma_shutdown()` is the same one used by `ma_frame_rx()` to receive PCM frames, so care must be taken to ensure that `ma_frame_rx()` is not being called concurrently when `ma_shutdown()` is invoked.

Calling `ma_shutdown()` causes the mic array thread(s) to terminate. When this happens, the `mic_array_start()` call inside `mic_array_task()` returns, after which `mic_array_task()` waits again to receive the next sampling-rate value on the channel before restarting the mic-array thread(s).

Weak callback APIs Two weak callback APIs - `xua_user_pdm_init()` and `xua_user_pdm_process()`, are provided, which optionally allow user code to be executed at startup (post PDM microphone initialisation) and after each sample frame is formed.

7.11 Audio controls via Human Interface Device (HID)

The design supports simple audio controls such as play/pause, volume up/down etc via the [USB Human Interface Device Class Specification](#).

This functionality is enabled by setting the `HID_CONTROLS` define to `1`. Setting to `0` disables this feature.

When turned on the following items are enabled:

1. HID descriptors are enabled in the Configuration Descriptor informing the host that the device has a HID interface.
2. A Get Report Descriptor request is enabled in `endpoint0`.
3. Endpoint data handling is enabled in the `buffer` thread.

The Get Descriptor Request enabled in endpoint 0 returns the report descriptor for the HID device. This details the format of the HID reports returned from the device to the host. It maps a bit in the report to a function such as play/pause.

The USB Audio Framework implements a report descriptor that should fit most basic audio device controls. If further controls are necessary the HID Report Descriptor in `hid_report_descriptor.h` should be modified. The default report size is 1 byte with the format as follows:

Table 42: Default HID Report Format

Bit	Function
0	Play/Pause
1	Scan Next Track
2	Scan Prev Track
3	Volume Up
4	Volume Down
5	Mute
6-7	Unused



On each HID report request from the host the function `UserHIDGetData()` is called from `XUA_Buffer_Ep()`. This function is passed an array `hidData[]` by reference. The programmer should report the state of the buttons into this array. For example, if a volume up command is desired, bit 3 should be set to 1, else 0.

Since the `UserHIDGetData()` function is called from the `XUA_Buffer_Ep()` thread, care should be taken not to add too much execution time to this function since this could cause issues with servicing other endpoints.

7.12 Device Firmware Upgrade (DFU) over USB

The DFU implementation in `lib_xua` is compliant with version 1.1 of [Universal Serial Bus Device Class Specification for Device Firmware Upgrade](#).

This section describes the DFU implementation in `lib_xua`, which is supported by `lib_dfu`. For information about using a DFU loader to send DFU commands to the USB Audio device, refer to appnote [AN02019: Using Device Firmware Upgrade \(DFU\) in USB Audio](#).

The USB device descriptors expose a DFU interface that handles updates to the boot image of the device over USB.

The host sends DFU requests as Host to Device Class requests to the DFU interface. On receiving DFU commands from the host, the `dfu_usb_class_int_requests` function is called from the Endpoint 0 thread. This function calls the DFU handler functions over the `dfuInterface` XC interface. The DFU handler thread, `DFUHandler` that implements the server side of the `dfuInterface` has to be scheduled on the same tile as the flash so it can access the flash memory. The `dfuInterface` interface essentially links USB to the [XMOS flash user library](#).

The DFU interface is enabled by default (See `XUA_DFU_EN` define in `xua_conf_default.h`). When DFU is enabled, there are two sets of descriptors that the device can export, depending on the mode in which it operates. There is a descriptor set for the runtime mode, which is the mode the device normally operates in and a set of descriptors for the DFU mode.

In the runtime mode, the DFU interface is one of potentially multiple interfaces that the device exposes. [Fig. 7](#) shows the DFU interface descriptors when enumerating in runtime mode as seen in a [Beagle USB analyser](#) trace.

Interface Descriptor		Radix: auto
bLength	9	
bDescriptorType	INTERFACE (0x04)	
bInterfaceNumber	3	
bAlternateSetting	0	
bNumEndpoints	0	
bInterfaceClass	Application Specific (0xfe)	
bInterfaceSubClass	Device Firmware Upgrade (0x01)	
bInterfaceProtocol	Runtime (0x01)	
iInterface	XMOS DFU (10)	

DFU Functional Descriptor		Radix: auto
bLength	9	
bDescriptorType	DFU FUNCTIONAL (0x21)	
bmAttributes.bitCanDnload	Yes (0b1)	
bmAttributes.bitCanUpload	Yes (0b1)	
bmAttributes.bitManifestationTolerant	Yes (0b1)	
bmAttributes.bitWillDetach	Yes (0b1)	
wDetachTimeOut	250ms (250)	
wTransferSize	64	
bcdDFUVersion	1.10 (0x0110)	

Fig. 7: DFU interface when part of runtime mode descriptor set



Note the **bInterfaceProtocol** field set to **Runtime**.

In DFU mode, the device exports the DFU descriptor set. The DFU mode descriptors specify only one interface, the DFU interface. Fig. 8 shows the DFU interface descriptors when enumerating in DFU mode as seen in a Beagle USB analyser trace.

Interface Descriptor		Radix: auto
bLength	9	
bDescriptorType	INTERFACE (0x04)	
bInterfaceNumber	0	
bAlternateSetting	0	
bNumEndpoints	0	
bInterfaceClass	Application Specific (0xfe)	
bInterfaceSubClass	Device Firmware Upgrade (0x01)	
bInterfaceProtocol	DFU mode (0x02)	
iInterface	XMOS DFU (10)	

DFU Functional Descriptor		Radix: auto
bLength	9	
bDescriptorType	DFU FUNCTIONAL (0x21)	
bmAttributes.bitCanDnload	Yes (0b1)	
bmAttributes.bitCanUpload	Yes (0b1)	
bmAttributes.bitManifestationTolerant	Yes (0b1)	
bmAttributes.bitWillDetach	Yes (0b1)	
wDetachTimeOut	250ms (250)	
wTransferSize	64	
bcdDFUVersion	1.10 (0x0110)	

Fig. 8: DFU interface when part of DFU mode descriptor set

Note the **bInterfaceProtocol** field set to **DFU mode**.

Before starting the DFU upload or download process, the host sends a **DFU_DETACH** command to detach the device from runtime to DFU mode. In response to the **DFU_DETACH** command, the device reboots itself into DFU mode and enumerates using the DFU mode descriptors. Once the device is in DFU mode, the DFU interface can accept commands defined by the [DFU 1.1 class specification](#).

After detaching the device, the host proceeds with the DFU download/upload commands to write/read the firmware upgrade image to/from the device. Once the DFU download or upload process is complete, the host sends a **DETACH** command, and the device reboots itself back in runtime mode.

Note

It is recommended that the runtime mode and DFU mode descriptors have different product IDs. This is to ensure that the host operating system loads the correct driver as the device switches between runtime and DFU modes. The runtime and DFU PID are defined as overridable defines **PID_AUDIO_2** and **DFU_PID** respectively in **xua_conf_default.h**. Users can define custom PIDs in their application by overriding these defines.

During the DFU download process, on receiving the first **DFU_DNLOAD** command (**wBlockNum** = 0), the device erases **FLASH_MAX_UPGRADE_SIZE** bytes of the upgrade section of the flash. This is done by repeatedly calling the function to erase a flash sector until the entire upgrade section is erased, and can take several seconds. To avoid the **DFU_DNLOAD** request timing out, the flash erase is instead done in the **DFU_GETSTATUS**



handling code for block 0. So for block 0, the device ends up returning the status as **dfuDNBUSY** several times while the flash erase is in progress. Fig. 9 describes the DFU download process.

Note

Once a valid upgrade image is loaded in flash, on subsequent reboots, the device will boot from the upgrade image. If the upgrade image is invalid e.g. invalid CRC, the factory image will be loaded. To revert back to the factory image, there is a custom request available **XMOS_DFU_REVERTFACTORY**.

For further details of DFU the API and the implementation, please see * **lib_dfu** (https://www.xmos.com/libraries/lib_dfu) documentation.



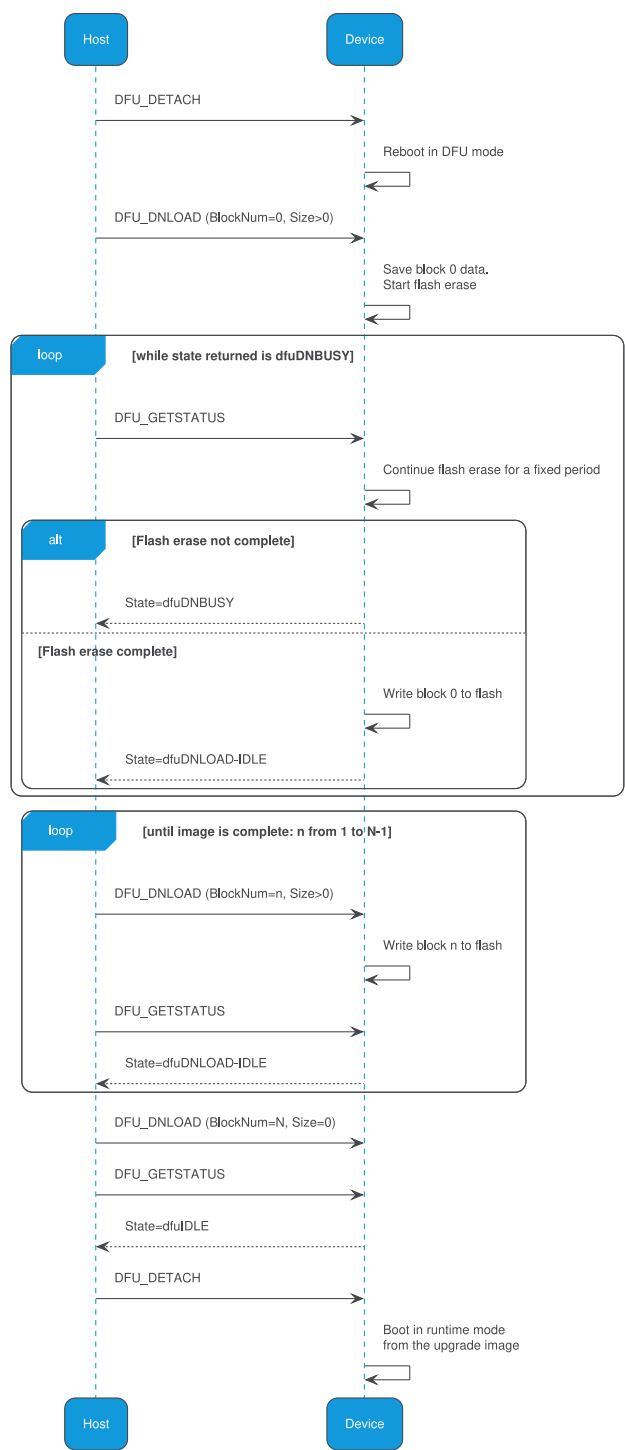


Fig. 9: Message sequence chart for the DFU download operation



Enumerating as a WinUSB device on Windows

The Endpoint 0 code supports extra descriptors called the [Microsoft Operating System \(MSOS\) descriptors](#) that allow the device to enumerate as a WinUSB device on Windows. The MSOS descriptors report the compatible ID as *WINUSB* which enables Windows to load *Winusb.sys* as the device's function driver without a custom INF file. This means that when the device is connected, the DFU interface shows up as WinUSB compatible automatically, without requiring the user to manually load a driver for it using a utility like Zadig.

The MSOS descriptors are present in the file `xua_ep0_msos_descriptors.c` and also in `lib_xud` in the file `simple_ep0_msos_descriptors.c`. In order to enumerate as a device capable of supplying MSOS descriptors, the device's `bcdUSB` version in the device descriptor has to be **0x0201**. On seeing the `bcdUSB` version as 0x0201 when the device enumerates, the host requests for a descriptor called the Binary Device Object Store (BOS) descriptor. This descriptor contains information about the capability of the device. It specifies the device to be MSOS 2.0 capable and contains information about the vendor request code (**bRequest**) and the request length (**wLength**) that the host needs to use to when making a vendor request to query for the MSOS descriptor.

The host then makes a vendor request with the **bRequest** and **wLength** as specified in the BOS platform descriptor querying for the MSOS descriptor.

Warning

If writing a host application that also sends vendor requests to the device, users should ensure that they do not use the **bRequest** that is reserved for the MSOS descriptor. The MSOS descriptor vendor request's **bRequest** is defined as the `XUA_REQUEST_GET_MSOS_DESCRIPTOR` define in `xua_conf_default.h`.

```
#define XUA_REQUEST_GET_MSOS_DESCRIPTOR 0x20
```

The MSOS descriptor reports the compatible ID as *WINUSB* for the DFU interface. It also specifies the device interface GUID in its registry property. The GUID is required to access the DFU interface from a user application running on the host (for example the Thesycon DFU driver or the dfu-util DFU application)

Note

The default device interface GUID for the DFU interfaces is specified in the `XUA_WINUSB_DEVICE_INTERFACE_GUID_DFU` define in `xua_conf_default.h`. Users can override this by redefining `XUA_WINUSB_DEVICE_INTERFACE_GUID_DFU` in the application. A utility such as [guidgenerator](#) can be used for generating a GUID.

Tip

The MSOS descriptors for reporting WinUSB compatibility are only relevant for Windows.



7.13 Resource usage

The following table details the resource usage of each component of the reference design software. Note, memory usage is approximate and varies based on device used, compiler settings etc.

Table 43: Resource Usage

Component	Cores	Memory (KB)	Ports
XUD library	1	9 (6 code)	USB ports
Endpoint 0	1	17.5 (10.5 code)	none
USB Buffering	2	22.5 (1 code)	1 x n bit port
Audio Hub	1	8.5 (6 code)	See Audio Hub and I²S
S/PDIF Tx	1	3.5 (2 code)	1 x 1 bit port
S/PDIF Rx	1	3.7 (3.7 code)	1 x 1 bit port
ADAT Rx	1	3.2 (3.2 code)	1 x 1 bit port
MIDI	1	6.5 (1.5 code)	2 x 1 bit ports
Mixer	(up to) 2	8.7 (6.5 code)	
ClockGen	1	2.5 (2.4 code)	

These resource estimates are based on the multichannel reference design with all options of that design enabled. For fewer channels, the resource usage is likely to decrease.

Note

lib_xud requires an 85 MIPS core to function correctly (i.e. on a 600MHz part only seven cores can run).

Note

Unlike other interfaces, since the USB PHY is internal, the USB ports are a fixed set of ports and cannot be modified. See **lib_xud** documentation for full details.



8 API reference

8.1 Configuration defines

An application using the USB audio framework needs to have defines set for configuration. Defaults for these defines are found in `xua_conf_default.h`.

These defines should be overridden in an optional header file `xua_conf.h` file or in the application's `CMakeLists.txt` for the relevant build configuration.

This section fully documents all of the settable defines and their default values (where appropriate).

Code location (tile)

XUA_AUDIO_IO_TILE_NUM

Location (tile) of audio I/O.

The tile number to run the AudioHub() task. The location if automatically detected from ports in the XN file.

XUA_XUD_TILE_NUM

Location (tile) of audio I/O.

The tile number to run lib_xud tasks on.

XUA_MIDI_TILE_NUM

Location (tile) of MIDI I/O.

The tile number to run MIDI tasks on. The location if automatically detected from ports in the XN file.

XUA_SPDIF_TX_TILE_NUM

Location (tile) of SPDIF Tx.

The tile number to run the S/PDIF tx task on. The location if automatically detected from ports in the XN file.

XUA_MIC_PDM_TILE_NUM

Location (tile) of PDM Rx.

The tile number to run the PDM Mics tasks on. The location if automatically detected from ports in the XN file.

XUA_PLL_REF_TILE_NUM

Location (tile) of reference signal to CS2100.

The tile number to run the PLL reference clock task on. The location if automatically detected from ports in the XN file.

Channel counts

NUM_USB_CHAN_OUT

Number of output channels (host to device). Default: NONE (Must be defined by app)



NUM_USB_CHAN_IN

Number of input channels (device to host). Default: NONE (Must be defined by app)

I2S_CHANS_DAC

Number of I2S channels to DAC/CODEC. Must be a multiple of 2.
Default: NONE (Must be defined by app)

I2S_CHANS_ADC

Number of I2S channels from ADC/CODEC. Must be a multiple of 2.
Default: NONE (Must be defined by app)

Frequencies and clocks

MAX_FREQ

Max supported sample frequency for device (Hz).
Default: 192000Hz

MIN_FREQ

Min supported sample frequency for device (Hz).
Default: 44100Hz

DEFAULT_FREQ

Default device sample frequency. A safe default should be used.
Default: MIN_FREQ

MCLK_441

Master clock defines for 44100 rates (in Hz).
Default: NONE (Must be defined by app)

MCLK_48

Master clock defines for 48000 rates (in Hz).
Default: NONE (Must be defined by app)

XUA_USE_SW_PLL

Enable/disable the use of the secondary/application PLL for generating and recovering master-clocks. Only available on xcore.ai devices.
Default: Enabled (for xcore.ai devices)

Audio Class

AUDIO_CLASS

Legacy USB Audio Class version.
Default: 2 (Audio Class version 2.0)
Note: XUA_USB_AUDIO_CLASS_HS and XUA_USB_AUDIO_CLASS_FS are derived from this value. Setting these defines directly will override this value.



XUA_AUDIO_CLASS_HS

Audio class version to run at HS.

Default: AUDIO_CLASS

Note: Set to 0 for no operation at HS.

XUA_AUDIO_CLASS_FS

Audio class version to run at FS.

Default: AUDIO_CLASS

Note: Set to 0 for no operation at FS.

Feature configuration

I²S/TDM

CODEC_MASTER

Defines whether XMOS device runs as master (i.e. drives LR and Bit clocks)

0: XMOS is I2S master. 1: CODEC is I2S master.

Default: 0 (XMOS is master)

XUA_I2S_N_BITS

Number of bits per channel for I2S/TDM. Supported values: 16/32-bit.

Default: 32 bits

XUA_PCM_FORMAT

Format of PCM audio interface. Should be set to XUA_PCM_FORMAT_I2S or XUA_PCM_FORMAT_TDM.

Default: XUA_PCM_FORMAT_I2S

MIDI

MIDI

Enable MIDI functionality including buffering, descriptors etc. Default: DISABLED.

MIDI_RX_PORT_WIDTH

MIDI Rx port width (1 or 4bit). Default: 1.

S/PDIF

XUA_SPDIF_TX_EN

Enables SPDIF Tx. Default: 0 (Disabled)

SPDIF_TX_INDEX

Defines which output channels (stereo) should be output on S/PDIF. Note, Output channels indexed from 0.

Default: 0 (i.e. channels 0 & 1)

XUA_SPDIF_RX_EN

Enables SPDIF Rx. Default: 0 (Disabled)



SPDIF_RX_INDEX

S/PDIF Rx first channel index, defines which channels S/PDIF will be input on. Note, indexed from 0.

Default: NONE (Must be defined by app when SPDIF_RX enabled)

ADAT

XUA_ADAT_RX_EN

Enables ADAT Rx. Default: 0 (Disabled)

ADAT_RX_INDEX

ADAT Rx first channel index. defines which channels ADAT will be input on. Note, indexed from 0.

Default: NONE (Must be defined by app when XUA_ADAT_RX_EN is true)

PDM microphones

XUA_NUM_PDM_MICS

Number of PDM microphone outputs in the design.

Default: 0

XUA_PDM_MIC_USE_PDM_ISR

Merge the PDM receive task into the decimation task using an ISR. Note: this works well with lower PDM mic counts but 8 and above may require separation into dedicated tasks.

DEFAULT: 1, Enable PDM RX ISR mode

XUA_PDM_MIC_USE_DDR

Indicates whether two microphones share a single data line (DDR mode).

When set to 1, the system uses Double Data Rate (DDR) signaling to read two microphones from a single data line. When set to 0, each microphone is read on a separate data line.

Default: 1

XUA_PDM_MIC_INDEX

Disable USB functionality just leaving AudioHub.

Default: Enabled

PDM Microphone first channel index, defines which channels microphones will be input on. Note, indexed from 0.

Default: 0 (i.e. channels [0:XUA_NUM_PDM_MICS-1])

DFU

XUA_DFU_EN

Enable DFU functionality.

Default: 1 (Enabled)



HID

HID_CONTROLS

Enable HID playback controls functionality.

1 for enabled, 0 for disabled.

Default 0 (Disabled)

USB device configuration

VENDOR_STR

Vendor String used by the device. This is also pre-pended to various strings used by the design.

Default: "XMOS"

VENDOR_ID

USB Vendor ID (or VID) as assigned by the USB-IF.

Default: 0x20B1 (XMOS)

PRODUCT_STR

USB Product String for the device. If defined will be used for both PRODUCT_STR_A2 and PRODUCT_STR_A1.

Default: Undefined

PRODUCT_STR_A2

Product string for Audio Class 2.0 mode.

Default: "XMOS xCORE (UAC2.0)"

PRODUCT_STR_A1

Product string for Audio Class 1.0 mode.

Default: "XMOS xCORE (UAC1.0)"

PID_AUDIO_1

USB Product ID (PID) for Audio Class 1.0 mode. Only required if XUA_AUDIO_CLASS_FS == 1.

Default: 0x0003

PID_AUDIO_2

USB Product ID (PID) for Audio Class 2.0 mode.

Default: 0x0002

BCD_DEVICE

Device firmware version number in Binary Coded Decimal format: 0xJJMN where JJ: major, M: minor, N: sub-minor version number.

NOTE: User code should not modify this but should modify BCD_DEVICE_J, BCD_DEVICE_M, BCD_DEVICE_N instead

Default: XMOS USB Audio Release version (e.g. 0x0651 for 6.5.1).



Stream Formats

Output/playback

OUTPUT_FORMAT_COUNT

Number of supported output stream formats.

Values 1,2,3 supported

Default: 2

STREAM_FORMAT_OUTPUT_1_RESOLUTION_BITS

Sample resolution (bits) of output stream Alternate 1.

Default: 24 if Alternate 1 is PCM, else 32 if DSD/RAW

Note, 24 on the lowest alt in case of OUTPUT_FORMAT_COUNT = 1 leaving 24bit as the designs default resolution.

STREAM_FORMAT_OUTPUT_2_RESOLUTION_BITS

Sample resolution (bits) of output stream Alternate 2.

Default: 16 if Alternate 2 is PCM, else 32 if DSD/RAW

STREAM_FORMAT_OUTPUT_3_RESOLUTION_BITS

Sample resolution (bits) of output stream Alternate 3.

Default: 32 if Alternate 2 is PCM, else 32 if DSD/RAW

HS_STREAM_FORMAT_OUTPUT_1_SUBSLOT_BYTES

Sample sub-slot size (bytes) of output stream Alternate 1 when running in high-speed.

Default: 4 if resolution for Alternate 1 is 24bits, else resolution / 8

Note, the default catches the 24bit special case where 4-byte subslot is nicer for our 32-bit machine. Typically do not care about this extra bus overhead at High-speed

HS_STREAM_FORMAT_OUTPUT_2_SUBSLOT_BYTES

Sample sub-slot size (bytes) of output stream Alternate 2 when running in high-speed.

Default: 4 if resolution for Alternate 2 is 24bits, else resolution / 8

Note, the default catches the 24bit special case where 4-byte subslot is nicer for our 32-bit machine. Typically do not care about this extra bus overhead at High-speed

HS_STREAM_FORMAT_OUTPUT_3_SUBSLOT_BYTES

Sample sub-slot size (bytes) of output stream Alternate 3 when running in high-speed.

Default: 4 if resolution for Alternate 3 is 24bits, else resolution / 8

Note, the default catches the 24bit special case where 4-byte subslot is nicer for our 32-bit machine. Typically do not care about this extra bus overhead at High-speed

FS_STREAM_FORMAT_OUTPUT_1_SUBSLOT_BYTES

Sample sub-slot size (bytes) of output stream Alternate 1 when running in full-speed.



Note, in full-speed mode bus bandwidth is at a premium, therefore pack samples into smallest possible sub-slot.

Default: `STREAM_FORMAT_OUTPUT_1_RESOLUTION_BITS / 8`

FS_STREAM_FORMAT_OUTPUT_2_SUBSLOT_BYTES

Sample sub-slot size (bytes) of output stream Alternate 2 when running in full-speed.

Note, in full-speed mode bus bandwidth is at a premium, therefore pack samples into smallest possible sub-slot.

Default: `STREAM_FORMAT_OUTPUT_2_RESOLUTION_BITS / 8`

FS_STREAM_FORMAT_OUTPUT_3_SUBSLOT_BYTES

Sample sub-slot size (bytes) of output stream Alternate 3 when running in full-speed.

Note, in full-speed mode bus bandwidth is at a premium, therefore pack samples into smallest possible sub-slot.

Default: `STREAM_FORMAT_OUTPUT_3_RESOLUTION_BITS / 8`

STREAM_FORMAT_OUTPUT_1_DATAFORMAT

Sample audio data-format if output stream Alternate 1.

Default: `UAC_FORMAT_TYPE1_RAW_DATA` when Alternate 1 is RAW/DSD else `UAC_FORMAT_TYPE1_PCM`

STREAM_FORMAT_OUTPUT_2_DATAFORMAT

Sample audio data-format if output stream Alternate 2.

Default: `UAC_FORMAT_TYPE1_RAW_DATA` when Alternate 2 is RAW/DSD else `UAC_FORMAT_TYPE1_PCM`

STREAM_FORMAT_OUTPUT_3_DATAFORMAT

Sample audio data-format if output stream Alternate 3.

Default: `UAC_FORMAT_TYPE1_RAW_DATA` when Alternate 3 is RAW/DSD else `UAC_FORMAT_TYPE1_PCM`

Input/recording

INPUT_FORMAT_COUNT

Number of supported input stream formats. Default: 1.

STREAM_FORMAT_INPUT_1_RESOLUTION_BITS

Sample resolution (bits) of input stream Alternate 1.

Default: 24

HS_STREAM_FORMAT_INPUT_1_SUBSLOT_BYTES

Sample sub-slot size (bytes) of input stream Alternate 1 when running in high-speed.

Default: 4 if resolution for Alternate 1 is 24bits, else resolution / 8

Note, the default catches the 24bit special case where 4-byte subslot is nicer for our 32-bit machine. Typically do not care about this extra bus overhead at High-speed



FS_STREAM_FORMAT_INPUT_1_SUBSLOT_BYTES

Sample sub-slot size (bytes) of input stream Alternate 1 when running in full-speed.

Note, in full-speed mode bus bandwidth is at a premium, therefore pack samples into smallest possible sub-slot.

Default: $\text{STREAM_FORMAT_INPUT_1_RESOLUTION_BITS} / 8$

STREAM_FORMAT_INPUT_1_DATAFORMAT

Sample audio data-format for input stream Alternate 1.

Default: `UAC_FORMAT_TYPE1_PCM`

Volume control

OUTPUT_VOLUME_CONTROL

Enable/disable output volume control including all processing and descriptor support.

Default: 1 (Enabled)

INPUT_VOLUME_CONTROL

Enable/disable input volume control including all processing and descriptor support.

Default: 1 (Enabled)

MIN_VOLUME

The minimum volume setting above -inf. This is a signed 8.8 fixed point number that must be strictly greater than -128 (0x8000)

Default: 0x8100 (-127db)

MAX_VOLUME

The maximum volume setting. This is a signed 8.8 fixed point number.

Default: 0x0000 (0db)

VOLUME_RES

The resolution of the volume control in db as a 8.8 fixed point number.

Default: 0x100 (1db)

Mixing

MIXER

Enable "mixer" core.

Default: 0 (Disabled)

MAX_MIX_COUNT

Number of separate mixes to perform.

Default: 8 if MIXER enabled, else 0



MIX_INPUTS

Number of channels input into the mixer.

Note, total number of mixer nodes is $MIX_INPUTS * MAX_MIX_COUNT$

Default: 18

MIN_MIXER_VOLUME

The minimum volume setting for the mixer unit above -inf. This is a signed 8.8 fixed point number that must be strictly greater than -128 (0x8000)

Default: 0x8100 (-127db)

MAX_MIXER_VOLUME

The maximum volume setting for the mixer. This is a signed 8.8 fixed point number.

Default: 0x0000 (0db)

VOLUME_RES_MIXER

The resolution of the volume control in db as a 8.8 fixed point number.

Default: 0x100 (1db)

Power

XUA_POWERMODE

Report as self or bus powered device. This affects descriptors and XUD usage and is important for USB compliance.

Default: XUA_POWERMODE_BUS

XUA_CHAN_BUFF_CTRL

Enable power saving feature in XUA_Buffer_Decouple()

If set to 1 then a channel is instantiated between the XUA_Buffer_Ep() and XUA_Buffer_Decouple() tasks (which together form the buffer between XUD and Audio) that limits shared memory polling in XUA_Buffer_Ep() to occur only when a change has been made by XUA_Buffer_Decouple(). This significantly reduces core power at the cost of two channel ends on the USB_TILE.



8.2 User function definitions

The following functions can be defined by an application using **lib_xua**.

Note

Default, empty, implementations of these functions are provided in **lib_xua**. These are marked as weak symbols so the application can simply define its own version of them.

External audio hardware configuration functions

The following functions can be optionally used by the design to configure external audio hardware. As a minimum, in most applications, it is expected that a implementation of *AudioHwConfig()* will need to be provided.

void **AudioHwInit**(void)

User audio hardware initialisation code.

This function is called when the device starts up and should contain user code to perform any required audio hardware initialisation

void **AudioHwConfig**(
 unsigned samFreq, unsigned mClk, unsigned dsdMode, unsigned sam-
 pRes_DAC, unsigned sampRes_ADC,
)

User audio hardware configuration code.

This function is called when on sample rate change and should contain user code to configure audio hardware (clocking, CODECs etc) for a specific mClk/Sample frequency. It is called from audiohub on XUA_AUDIO_IO_TILE_NUM.

Parameters

- ▶ **samFreq** – The new sample frequency (in Hz)
- ▶ **mClk** – The new master clock frequency (in Hz)
- ▶ **dsdMode** – DSD mode, DSD_MODE_NATIVE, DSD_MODE_DOP or DSD_MODE_OFF
- ▶ **sampRes_DAC** – Playback sample resolution (in bits)
- ▶ **sampRes_ADC** – Record sample resolution (in bits)

void **AudioHwConfig_Mute**(void)

User code mute audio hardware.

This function is called before *AudioHwConfig()* and should contain user code to mute audio hardware before a sample rate change in order to reduced audible pops/clicks It is called from audiohub on XUA_AUDIO_IO_TILE_NUM.

Note, if using the application PLL of a xcore.ai device this function will be called before the master-clock is changed

void **AudioHwConfig_UnMute**(void)

User code to un-mute audio hardware.

This function is called after *AudioHwConfig()* and should contain user code to un-mute audio hardware after a sample rate change. It is called from audiohub on XUA_AUDIO_IO_TILE_NUM.



void **AudioHwShutdown**(void)

User audio hardware de-initialisation code.

This function is called when streaming stops (device enumerated but audio is idle) and should contain user code to perform any required audio hardware de-initialisation. This can be useful for saving power in the audio sub-system. It is called from audiohub on XUA_AUDIO_IO_TILE_NUM.

Note this callback will only be called if the XUA_LOW_POWER_NON_STREAMING define is set, otherwise lib_xua assumes that I2S is always looping.

Audio stream start/stop functions

The following functions can be optionally used by the design. They can be useful for mute lines etc.

void **UserAudioStreamState**(int inputActive, int outputActive)

User stream start code.

User code to perform any actions required at every stream start - either input or output.

/param inputActive An input stream is active if 1, else inactive if 0 /param OutputActive An output stream is active if 1, else inactive if 0

Host active functions

The following function can be used to signal that the device is connected to a valid host.

void **UserHostActive**(int active)

User host active code.

This function can be used to perform user defined actions based on host present/not-present events. This function is called on a change in state.

Parameters

- **active** – Indicates if the host is active or not. 1 for active, else 0

HID controls

The following function is called when the device wishes to read physical user input (buttons etc). The function should write relevant HID bits into this array. The bit ordering and functionality is defined by the HID report descriptor used.

size_t **UserHIDGetData**(
const unsigned id, unsigned char hidData[HID_MAX_DATA_BYTES],
)

Get the data for the next HID Report.

Parameters

- **id** – [in] The HID Report ID (see 5.6, 6.2.2.7, 8.1 and 8.2 of the USB Device Class Definition for HID 1.1) Set to zero if the application provides only one HID Report which does not include a Report ID
- **hidData** – [out] The HID data If using Report IDs, this function places the Report ID in the first element; otherwise the first element holds the first byte of HID event data.

Return values

Zero – means no new HID event data has been recorded for the given *id*



Returns

The length of the HID Report in the *hidData* argument

PDM Mics user callback functions

Two weak callback APIs are provided which optionally allow user code to be executed at startup (before PDM microphone initialisation) and after each PCM sample is received. These can be useful for custom hardware initialisation required by the PDM microphone or post processing such as gain control before samples are forwarded to XUA

```
void xua_user_pdm_init(unsigned
                        channel_map[MIC_ARRAY_CONFIG_MIC_COUNT])
```

User pre-PDM mic function callback (optional).

This function is called before mic-array initialisation—both the first time and every time the mic array restarts. It can be used to update the **channel_map** in cases where the mapping between PDM input port pins and microphone output channels is not 1:1. It may also be used to initialise any PDM related hardware.

Note

This function is called on the same tile as the mic-array task (XUA_MIC_PDM_TILE_NUM). **channel_map** is a global array whose default initialisation occurs on that tile before **mic_array_task()** runs. **xua_user_pdm_init()** is invoked by **mic_array_task()** itself, before starting the mic-array hardware thread.

```
void xua_user_pdm_process(int32_t
                          mic_audio[MIC_ARRAY_CONFIG_MIC_COUNT])
```

USB PDM Mic PCM sample post processing callback (optional).

This is called after a PCM sample is received from mic_array. It can be used to modify the samples (gain, filter etc.) before sending to XUA audiohub.

Warning

This function is called from Audiohub (I2S) and so any processing must take significantly less than half of a sample period else I2S will break timing.

Parameters

- **mic_audio** – Array of samples for in-place processing

8.3 Component API

The following functions can be called from the top level main of an application and implement the various components described in [Software architecture](#).

When using the USB audio framework the **c_ep_in** array is always composed in the following order:

- Endpoint 0 (in)
- Audio Feedback endpoint (if output enabled)
- Audio IN endpoint (if input enabled)



- ▶ MIDI IN endpoint (if MIDI enabled)
- ▶ Clock Interrupt endpoint

The array **c_ep_out** is always composed in the following order:

- ▶ Endpoint 0 (out)
- ▶ Audio OUT endpoint (if output enabled)
- ▶ MIDI OUT endpoint (if MIDI enabled)

```
void XUA_Endpoint0(
    chanend c_ep0_out, chanend c_ep0_in, NULLABLE_RESOURCE(chanend,
    c_aud_ctl), NULLABLE_RESOURCE(chanend, c_mix_ctl), NULLABLE_RESOURCE(chanend,
    c_clk_ctl), NULLABLE_CLIENT_INTERFACE(i_dfu, dfuInterface),
)
```

Endpoint 0 task for USB Audio devices

Function implementing Endpoint 0 for enumeration, control and configuration of USB audio devices. It uses the descriptors defined in **xua_ep0_descriptors.h**.

Parameters

- ▶ **c_ep0_out** – Chanend connected to the XUD_Main() out endpoint array
- ▶ **c_ep0_in** – Chanend connected to the XUD_Main() in endpoint array
- ▶ **c_aud_ctl** – Chanend connected to the decouple thread for control audio (sample rate changes etc.). Note when null, the audio device only supports single sample rate/format and DFU is not supported either since this channel is used to carry messages about format, rate and DFU state
- ▶ **c_mix_ctl** – Optional chanend to be connected to the mixer core(s) if present
- ▶ **c_clk_ctl** – Optional chanend to be connected to the clockgen core if present
- ▶ **dfuInterface** – Interface to DFU task (this task must be run on a tile connected to boot flash).

```
void XUA_Buffer(
    chanend c_aud_out, chanend c_aud_in, chanend c_aud_fb, chanend
    c_midi_from_host, chanend c_midi_to_host, chanend
    c_midi, NULLABLE_RESOURCE(chanend, c_int), NULLABLE_RESOURCE(chanend,
    c_clk_int), chanend c_sof, chanend c_aud_ctl, NULLABLE_RESOURCE(in_port_t,
    p_off_mclk), chanend c_hid, chanend c_aud, chanend
    c_audio_rate_change, CLIENT_INTERFACE pll_ref_if, i_pll_ref, chanend
    c_swpll_update,
)
```

USB Audio Buffering Core(s).

This function buffers USB audio data between the XUD and the audio subsystem. Most of the chanend parameters to the function should be connected to XUD_Manager(). The uses two cores.

Parameters

- ▶ **c_aud_out** – Audio OUT endpoint channel connected to the XUD
- ▶ **c_aud_in** – Audio IN endpoint channel connected to the XUD



- ▶ **c_aud_fb** – Audio feedback endpoint channel connected to the XUD
- ▶ **c_midi_from_host** – MIDI OUT endpoint channel connected to the XUD
- ▶ **c_midi_to_host** – MIDI IN endpoint channel connected to the XUD
- ▶ **c_midi** – Channel connected to MIDI thread
- ▶ **c_int** – Audio clocking interrupt endpoint channel connected to the XUD
- ▶ **c_clk_int** – Optional chanend connected to the clockGen() thread if present
- ▶ **c_sof** – Start of frame channel connected to the XUD
- ▶ **c_aud_ctl** – Audio control channel connected to Endpoint0()
- ▶ **p_off_mclk** – A port that is clocked of the MCLK input (not the MCLK input itself)
- ▶ **c_hid** – Channel connected to the HID handler thread
- ▶ **c_aud** – Channel connected to XUA_AudioHub() thread
- ▶ **c_audio_rate_change** – Channel to notify and synchronise on audio rate change
- ▶ **i_pll_ref** – Interface to task that toggles reference pin to CS2100
- ▶ **c_swpll_update** – Channel connected to software PLL task. Expects master clock counts based on USB frames.

```
void XUA_AudioHub(
    NULLABLE_RESOURCE(chanend, c_aud), NULLABLE_RESOURCE(clock,
    clk_audio_mclk), NULLABLE_RESOURCE(clock, clk_audio_bclk), NULLABLE_RESOURCE(in_port_t,
    p_mclk_in), NULLABLE_RESOURCE(i2s_clk_port_type,
    p_lrclk), NULLABLE_RESOURCE(i2s_clk_port_type, p_bclk), NULLABLE_ARRAY_OF_SIZE(out_buffered_p,
    p_i2s_dac, I2S_WIRES_DAC), NULLABLE_ARRAY_OF_SIZE(in_buffered_port_32_t,
    p_i2s_adc, I2S_WIRES_ADC), chanend c_spdif_tx, chanend c_dig, chanend
    c_audio_rate_change, NULLABLE_SERVER_INTERFACE(i_dfu, dfuInter-
    face), chanend c_pdm_in,
)
```

The audio driver thread.

This function drives I2S ports and handles samples to/from other digital I/O threads.

Parameters

- ▶ **c_aud** – Audio sample channel connected to the mixer() thread or the decouple() thread
- ▶ **clk_audio_mclk** – Nullable clockblock to be clocked from master clock
- ▶ **clk_audio_bclk** – Nullable clockblock to be clocked from i2s bit clock
- ▶ **p_mclk_in** – Master clock inport port (must be 1-bit). Use null when xcore is slave
- ▶ **p_lrclk** – Nullable port for I2S sample clock
- ▶ **p_bclk** – Nullable port for I2S bit clock
- ▶ **p_i2s_dac** – Nullable array of ports for I2S data output lines
- ▶ **p_i2s_adc** – Nullable array of ports for I2S data input lines
- ▶ **i_SoftPll** – Interface to software PLL task
- ▶ **c_spdif_tx** – Channel connected to S/PDIF transmitter core from lib_spdif
- ▶ **c_dig** – Channel connected to the clockGen() thread for receiving/transmitting samples



- ▶ **c_audio_rate_change** – Channel notifying ep_buffer of an mclk frequency change and sync for stable clock
- ▶ **dfuInterface** – Interface supporting DFU methods
- ▶ **c_pdm_in** – Channel for receiving decimated PDM samples

void **mixer**(chanend c_to_host, chanend c_to_audio, chanend c_mix_ctl)

Digital sample mixer.

This thread mixes audio streams between the decouple() thread and the audio() thread.

Parameters

- ▶ **c_to_host** – a chanend connected to the decouple() thread for receiving/transmitting samples
- ▶ **c_to_audio** – a chanend connected to the audio() thread for receiving/transmitting samples
- ▶ **c_mix_ctl** – a chanend connected to the Endpoint0() thread for receiving control commands

void **clockGen**(

NULLABLE_RESOURCE(streaming_chanend_t, c_spdif_rx), NULLABLE_RESOURCE(streaming_chanend_t, c_adat_rx), CLIENT_INTERFACE(pll_ref_if, i_pll_ref), chanend c_audio, chanend c_clk_ctl, chanend c_clk_int, chanend c_audio_rate_change, port p_for_mclk_count_aud, chanend c_sw_pll,

)

Clock generation and digital audio I/O handling.

Parameters

- ▶ **c_spdif_rx** – channel connected to S/PDIF receive thread
- ▶ **c_adat_rx** – channel connect to ADAT receive thread
- ▶ **i_pll_ref** – interface to task that outputs clock signal to drive external frequency synthesizer
- ▶ **c_audio** – channel connected to the audio() thread
- ▶ **c_clk_ctl** – channel connected to Endpoint0() for configuration of the clock
- ▶ **c_clk_int** – channel connected to the decouple() thread for clock interrupts
- ▶ **c_audio_rate_change** – channel to notify of master clock change
- ▶ **p_for_mclk_count_aud** – port used for counting mclk and providing a timestamp
- ▶ **c_sw_pll** – channel used to communicate with software PLL task

void **usb_midi**(

NULLABLE_RESOURCE(in_buffered_port_1_t, p_midi_in), NULLABLE_RESOURCE(port, p_midi_out), NULLABLE_RESOURCE(clock, clk_midi), NULLABLE_RESOURCE(chanend, c_midi), unsigned cable_number,

)

USB MIDI I/O task.

This function passes MIDI data between XUA_Buffer and MIDI UART I/O.

Parameters

- ▶ **p_midi_in** – 1-bit input port for MIDI
- ▶ **p_midi_out** – 1-bit output port for MIDI



- ▶ **clk_midi** – Clock block used for clockin the UART; should have a rate of 100MHz
- ▶ **c_midi** – Chanend connected to the decouple() thread
- ▶ **cable_number** – The cable number of the MIDI implementation. This should be set to 0.

```
void mic_array_task(
    chanend c_mic_to_audio, unsigned channel_map[MIC_ARRAY_CONFIG_MIC_COUNT],
)
```

USB PDM microphone task.

Starts the mic-array processing thread(s)

Supported sample rates: 16 kHz, 32 kHz, and 48 kHz.

The task runs in a continuous while(1) loop until **ma_shutdown()** is invoked. When **ma_shutdown()** is called, the internal mic thread terminates (**mic_array_start()** returns). After that, a new sampling-rate value may be received on the same channel, and the mic-array thread is then started again at the new rate.

c_mic_to_audio channel usage:

- ▶ While the mic thread is running, decimated PCM frames are sent from the mic array to the application over this channel.
- ▶ Before the mic thread is started, the PCM sampling rate is received over this channel.

Parameters

- ▶ **c_mic_to_audio** – Channel over which decimated PCM frames are produced by the mic array and delivered to the application.
- ▶ **channel_map** – Array mapping the logical microphone indices to PDM input channels. The i^{th} entry is the pdm-data port pin that is routed to microphone output channel i



Copyright © 2026, All Rights Reserved.

Xmos Ltd. is the owner or licensee of this design, code, or Information (collectively, the "Information") and is providing it to you "AS IS" with no warranty of any kind, express or implied and shall have no liability in relation to its use. Xmos Ltd makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.

Xmos, XCORE, VocalFusion and the Xmos logo are registered trademarks of Xmos Ltd. in the United Kingdom and other countries and may not be used without written permission. Company and product names mentioned in this document are the trademarks or registered trademarks of their respective owners.

