lib_xua

USB Audio Shared Components. For use in the XMOS USB Audio Refererence Designs.

This library enables the development of USB Audio devices on the XMOS xCORE architecture.

Features

Key features of the various applications in this repository are as follows

- USB Audio Class 1.0/2.0 Compliant
- Fully Asynchronous operation
- Support for the following sample frequencies: 8, 11.025, 12, 16, 32, 44.1, 48, 88.2, 96, 176.4, 192, 352.8, 384kHz
- Input/output channel and individual volume/mute controls supported
- Support for dynamically selectable output audio formats (e.g. resolution)
- Field firmware upgrade compliant to the USB Device Firmware Upgrade (DFU) Class Specification
- S/PDIF output
- S/PDIF input
- ADAT output
- ADAT input
- MIDI input/output (Compliant to USB Class Specification for MIDI devices)
- DSD output (Native and DoP mode) at DSD64 and DSD128 rates
- Mixer with flexible routing
- Simple playback controls via Human Interface Device (HID)
- Support for operation with Apple devices (requires software module sc_mfi for MFI licensees only - please contact XMOS)

Note, not all features may be supported at all sample frequencies, simultaneously or on all devices. Some features also require specific host driver support.

Software version and dependencies

This document pertains to version 0.1.2 of this library. It is known to work on version 14.3.2 of the xTIMEcomposer tools suite, it may work on other versions.

This library depends on the following other libraries:

- lib_xud (>=0.1.0)
- lib_logging (>=2.1.0)

Related Application Notes

The following application notes use this library:

- AN000246 - Simple USB Audio Device using lib_xua

1 About This Document

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

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

2 Host System Requirements

- Mac OSX version 10.6 or later
- Windows XP, Vista, 7, 8 or 10 with Thesycon Audio Class 2.0 driver for Windows (Tested against version 3.20). Please contact XMOS for details.
- Windows XP, Vista, 7, 8 or 10 with built-in USB Audio Class 1.0 driver.

3 Overview

Functionality	
Provides USB interface to audio I/O.	
Supported Standards	
USB	USB 2.0 (Full-speed and High-speed)
	USB Audio Class 1.0[1]
	USB Audio Class 2.0[2]
	USB Firmware Upgrade (DFU) 1.1[3]
	USB Midi Device Class 1.0[4]
Audio	I2S/TDM
	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	xCORE-200 Series
Requirements	
Development Tools	xTIMEcomposer Development Tools v14 or later
USB	xCORE-200 Series device with integrated USB Phy
Audio	External audio DAC/ADC/CODECs (and required supporting componentry) supporting I2S/TDM
Boot/Storage	Compatible SPI Flash device (or xCORE-200 device with internal flash)
Licensing and Support	
Reference code provided without charge under license from XMOS.	
Please visit http://www.xmos.com/support/contact for support.	
Reference code is maintained by XMOS Limited.	

[1] http://www.usb.org/developers/devclass_docs/audio10.pdf
[2] http://www.usb.org/developers/devclass_docs/Audio2.0_final.zip
[3] http://www.usb.org/developers/devclass_docs/DFU_1.1.pdf
[4] http://www.usb.org/developers/devclass_docs/midi10.pdf

4 Hardware Platforms

A range of hardware platforms for evaluating USB Audio on XMOS devices.

Specific, in depth, details for each platform/board are out of scope of this library documentation however, the features of the most popular platform are described below with the view of providing a worked example.

Please also see application note AN00246.

4.1 xCORE-200 Multi-Channel Audio Board

The XMOS xCORE-200 Multi-channel Audio board[5] (XK-AUDIO-216-MC) is a complete hardware and reference software platform targeted at up to 32-channel USB and networked audio applications, such as DJ decks and mixers.

The Multichannel Audio Platform hardware is based around the XE216-512-TQ128 multicore microcontroller; an dual-tile xCORE-200 device with an integrated High Speed USB 2.0 PHY, RGMII (Gigabit Ethernet) interface and 16 logical cores delivering up to 2000MIPS of deterministic and responsive processing power.

Exploiting the flexible programmability of the xCORE-200 architecture, the Multi-channel Audio Platform supports either USB or network audio source, streaming 8 analogue input and 8 analogue output audio channels simultaneously - at up to 192kHz.

For full details regarding the hardware please refer to xCORE-200 Multichannel Audio Platform Hardware Manual[6].

The reference board has an associated firmware application that uses *lib_xua* to implemented a USB Audio Devicce. Full details of this application can be found in the USB Audio Design Guide.

4.1.1 Analogue Input & Output

A total of eight single-ended analog input channels are provided via 3.5mm stereo jacks. Each is fed into a CirrusLogic CS5368 ADC. Similarly a total of eight single-ended analog output channels are provided. Each is fed into a CirrusLogic CS4384 DAC.

The four digital I2S/TDM input and output channels are mapped to the xCORE input/outputs through a header array. This jumper allows channel selection when the ADC/DAC is used in TDM mode

4.1.2 Digital Input & Output

Optical and coaxial digital audio transmitters are used to provide digital audio input output in formats such as IEC60958 consumer mode (S/PDIF) and ADAT. The output data streams from the xCORE-200 are re-clocked using the external master clock to synchronise the data into the audio clock domain. This is achieved using simple external D-type flip-flops.

4.1.3 MIDI

MIDI I/O is provided on the board via standard 5-pin DIN connectors. The signals are buffered using 5V line drivers and are then connected to 1-bit ports on the xCORE-200, via a 5V to 3.3V buffer.

[5] https://www.xmos.com/support/boards?product=18334
[6] https://www.xmos.com/support/boards?product=18334&component=18687

4.1.4 Audio Clocking

A flexible clocking scheme is provided for both audio and other system services. In order to accommodate a multitude of clocking options, the low-jitter master clock is generated locally using a frequency multiplier PLL chip. The chip used is a Phaselink PL611-01, which is pre-programmed to provide a 24MHz clock from its CLK0 output, and either 24.576 MHz or 22.5792MHz from its CLK1 output.

The 24MHz fixed output is provided to the xCORE-200 device as the main processor clock. It also provides the reference clock to a Cirrus Logic CS2100, which provides a very low jitter audio clock from a synchronisation signal provided from the xCORE-200.

Either the locally generated clock (from the PL611) or the recovered low jitter clock (from the CS2100) may be selected to clock the audio stages; the xCORE-200, the ADC/DAC and Digital output stages. Selection is conntrolled via an additional I/O, bit 5 of PORT 8C.

4.1.5 LEDs, Buttons and Other IO

An array of 4*4 green LEDs, 3 buttons and a switch are provided for general purpose user interfacing. The LED array is driven by eight signals each controlling one of 4 rows and 4 columns.

A standard XMOS xSYS interface is provided to allow host debug of the board via JTAG.

5 Software Overview

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

lib_xua provides the fundamental building blocks for producing USB Audio products on XMOS devices.

Every system is required to have the shared components from *lib_xua* listed in Table 1.

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 utlises two cores
AudioHub	Handles audio I/O over I2S and manages audio data to/from other digital audio I/O components.

Table 1: Required XUA Components

In addition low-level USB I/0 is required and is provided by the external dependency *lib_xud*

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

Table 2: Additional Components Required

In addition Table 3 shows optional components that can be added/enabled from *lib_xua*

Component	**Description**
Mixer	Allows digital mixing of input and output channels. It can also handle volume control instead of the decoupler.
S/PDIF Transmitter	Outputs samples of an S/PDIF digital audio interface.
S/PDIF Receiver	Inputs samples of an S/PDIF digital audio interface (requires the clockgen component).
ADAT Receiver	Inputs samples of an ADAT digital audio interface (requires the clockgen component).
Clockgen	Drives an external frequency generator (PLL) and manages changes between internal clocks and external clocks arising from digital input.
MIDI	Outputs and inputs MIDI over a serial UART interface.
PDM Microphones	Receives PDM data from microphones and performs PDM to PCM conversion

Table 3: Optional Components

Figure 1 shows how the components interact with each other in a typical system. The green circles represent cores with arrows indicating inter-core communications.

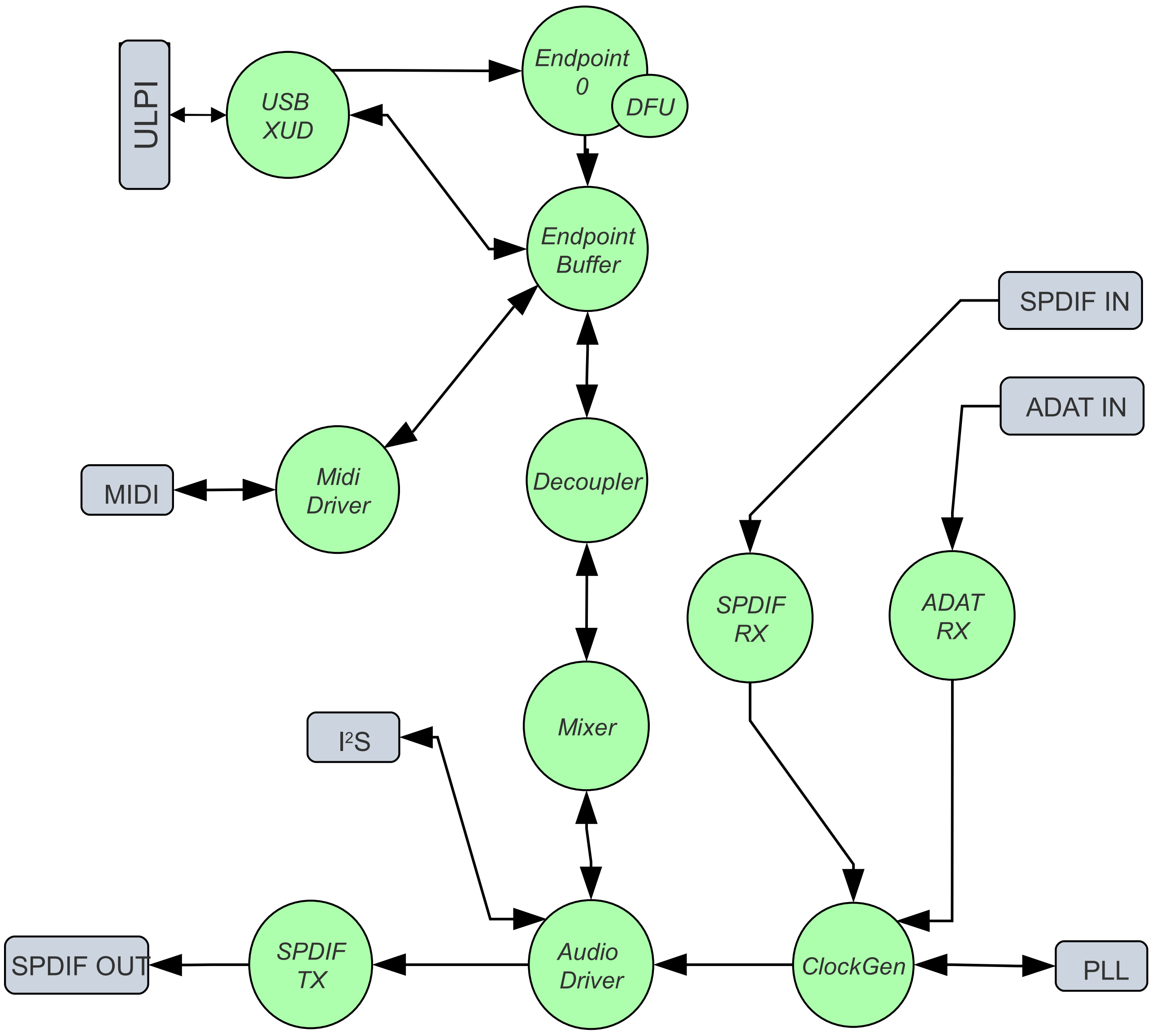

Figure 1: USB Audio Core Diagram

6 Using lib_xua

This sections describes the basic usage of *lib_xud*. It provides a guide on how to program the USB Audio Devices using *lib_xud*.

Reviewing application note AN00246 is highly recommended at this point.

6.1 Library structure

The code is split into several directories.

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

Table 4: lib_xua structure

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

6.2 Including in a project

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

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

It is also required to add `lib_xua` to the USED_MODULES field of your application Makefile:

```
USED_MODULES = .. lib_xua ...
```

6.3 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 n-bit port for internal feedback calculation (typically a free, unused port is used e.g. *16B*)
- A clock-block, which will be clocked from the master clock input port

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 */
```

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

The `XUA_AudioHub()` function requires an audio master clock input to clock the physical audio I/O. Less obvious is the reasoning for the `XUA_Buffer()` task having the same requirement - 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 needs some additional audio I/O, I2S or SPDIF for example.

These should be passed into the various cores as required - see API and Features sections.

6.4 Running the core components

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

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

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

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

`XUA_Buffer()` expects its `p_for_mclk_count` argument to be clocked from the audio master clock before being passed it. The following code satisfies this requirement:

```
{
        /* Connect master-clock clock-block to clock-block pin */
        set_clock_src(clk_audio_mclk_usb, p_mclk_in_usb);           /* Clock clock-block from mclk pin */
        set_port_clock(p_for_mclk_count, clk_audio_mclk_usb);       /* Clock the "count" port from the clock
           ↪ block */
        start_clock(clk_audio_mclk_usb);                            /* Set the clock off running */

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

}
```

Keeping this configuration outside of `XUA_Buffer()` does not preclude the possibllty of sharing `p_mclk_in_usb` port with additional components

To produce a fully operating device a call to `XUD_Main()` (from `lib_xud`) must also be made for USB connectivity:

```
/* Low level USB device layer core */
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 cores */
chan c_sof;

/* Channel for audio data between buffering cores and AudioHub/IO core */
chan c_aud;

/* Channel for communicating control messages from EP0 to the rest of the device (via the buffering cores) */
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.

6.5 Configuring XUA

Configuration of the various build time options of `lib_xua` is done via the optional header *xua_conf.h*. Such build time options include audio class version, sample rates, channel counts etc. Please see the API section for full listings.

The build system will automatically include the *xua_conf.h* header file as appropriate - the user should continue to include *xua.h* as previously directed. A simple example 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
```

6.6 User functions

To enable custom functionality, such as configuring external audio hardware, custom functionality on stream start/stop etc various user overridable functions are provided (see API section for full listings). The default implementations are empty and perform no function.

6.7 Built in main()

Some users maybe not have the software development experiance and simply want to change some settings, whilst others may want to fully customise the implementation - adding additional functionality such as adding DSD or possibly only using a subset of the funcitons provided - just `XUA_AudioHub` for example.

Whilst it is is possible to build up a system from the components

6.8 Enabling Additional Features

This sections describes only the basic feature set of `lib_xua` details on enabling additional features e.g. S/PDIF can be found later in this document.

7 Implementation Detail

This section describes the software architecture of a USB Audio device implemented using *lib_xua*, it's dependancies and other supporting libraries.

This section will now examine these components in further detail.

7.1 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 core and communicates with endpoint cores though a mixture of shared memory and channel communications.

For more details and full XUD API documentation please refer to *lib_xud*.

Figure 1 shows the XUD library communicating with two other cores:

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

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.

7.2.1 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 core and follows a similar format to that of the USB Device examples in *lib_xud* (i.e. Example HID Mouse Demo). 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 `module_usb_device`) 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, please refer to *lib_xud*.

The `USB_StandardRequests()` function takes the devices various descriptors as parameters, these are passed from data structures found in the `descriptors.h` file. These data structures are fully customised based on the how the design is configured using various defines (see XM-005512-PC).

The `USB_StandardRequests()` functions returns a `XUD_Result_t`. `XUD_RESULT_OKAY` indicates 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 parse the `USB_SetupPacket_t` structure to handle device specific requests and then calling `USB_StandardRequests()` as required. This is described next.

7.2.2 Over-riding Standard Requests

The USB Audio design “over-rides” some of the requests handled by `USB_StandardRequests()`, for example it uses the SET_INTERFACE request to indicate it 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 called to handle the response to the host etc.

7.2.3 Class Requests

Before making the call to USB_StandardRequests() the setup packet is parsed for Class requests. These are handled in functions such as AudioClasRequests_2(), AudioClassRequests_2, DFUDeviceRequests() 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 he endpoint 0 request parsing where AudioEndpointRequests_1() is called.

The AudioClassRequests_X() functions parses the request further 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_CNTROL to a Clock Unit in the devices topology (as described in the devices descriptors).

The new sample frequency is extracted and passed via channel to the rest of the design - through the buffering code and eventually to the Audio IO/I2S core. The AudioClassRequests_2() function waits for a handshake to propagate back though 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 core 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 decoupler core 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 core reads the volume values from this array. Note that this array is shared between Endpoint 0 and the decoupler core. This is done in a safe manner, since only Endpoint 0 can write to the array, word update is atomic between cores and the decoupler core only reads from the array (ordering between writes and reads is unimportant in this case). Inline assembly is used by the decoupler core 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 §7.5.

7.3 Audio Endpoints (Endpoint Buffer and Decoupler)

7.3.1 Endpoint Buffer

All endpoints other that Endpoint 0 are handled in one core. This core is implemented in the file `usb_buffer.xc`. This core is communicates directly with the XUD library.

The USB buffer core 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.

7.3.2 Decoupler

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

7.3.3 Audio Buffering Scheme

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

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

1. The decouple core receives samples from the audio core 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 buffer cores needs a buffer to send to the XUD core (after sending the previous buffer), the decouple core is signalled (via a shared memory flag).
3. Upon this signal from the buffering core, the decouple core passes the next packet from the FIFO to the buffer core. It also signals to the XUD library that the buffer core is able to send a packet.
4. When the buffer core has sent this buffer, it signals to the decouple that the buffer has been sent and the decouple core moves the read pointer of the FIFO.

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

1. The decouple core passes a pointer to the buffering core pointing into a FIFO of data and signals to the XUD library that the buffering core is ready to receive.
2. The buffering core then reads a USB packet into the FIFO and signals to the decoupler that the packet has been read.
3. Upon receiving this signal the decoupler core updates the write pointer of the FIFO and provides a new pointer to the buffering core to fill.
4. Upon request from the audio core, the decoupler core sends samples to the audio core by reading samples out of the FIFO.

7.3.4 Decoupler/Audio Core interaction

To meet timing requirements of the audio system, the decoupler core must respond to requests from the audio system to send/receive samples immediately. An interrupt handler is set up in the decoupler core 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 core 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 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 audio subsystem transfers samples destined for the host, then the decouple core sends samples from the host to device. 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 core and 8 received every interrupt.

The complete communication scheme is shown in the table below (for non sample frequency change case):

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

Table 5: Decouple/Audio System Channel Communication

The request and acknowledgement sent to/from Decouple 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

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 asynchronous clocking scheme means that the device is the clocking master than therefore means a high-quality local master clock source can be used.

After each received USB SOF token, the buffering core 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 USB buffering core via an explicit feedback IN endpoint. If both input and output is enabled then the feedback is implicit based on the audio stream sent to the host.

USB Rate Control

The Audio core must consume data from USB and provide data to USB at the correct rate for the selected sample frequency. The *USB 2.0 Specification* states that the maximum variation on USB packets can be +/- 1 sample per USB frame. USB frames are sent at 8kHz, so on average for 48kHz each packet contains six samples per channel. The device uses Asynchronous mode, so 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. Table 6 shows the allowed number of samples per packet for each example audio frequency.

See USB Device Class Definition for Audio Data Formats v2.0 section 2.3.1.1 for full details.

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

Table 6: Allowed samples per packet

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

7.4 Audio Driver

The audio driver receives and transmits samples from/to the decoupler or mixer core over an XC channel. It then drives several in and out I2S/TDM channels. If the firmware is configured with the CODEC as slave, it will also drive the word and bit clocks in this core as well. The word clocks, bit clocks and data are all derived from the incoming master clock (typically the output of the external oscillator or PLL). The audio driver is implemented in the file `audio.xc`.

The audio driver captures and plays audio data over I2S. It also forwards on relevant audio data to the S/PDIF transmit core.

The audio core must be connected to a CODEC that supports I2S (other modes such as “left justified” can be supported with firmware changes). In slave mode, the XMOS device acts as the master generating the Bit Clock (BCLK) and Left-Right Clock (LRCLK, also called Word Clock) signals. Any CODEC or DAC/ADC combination that supports I2S and can be used.

Table 7 shows the signals used to communicate audio between the XMOS device and the CODEC.

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

Table 7: I2S Signals

The bit clock controls the rate at which data is transmitted to and from the CODEC. In the case where the XMOS device 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 CODEC.

Table 8 shows some example clock frequencies and divides for different sample rates (note that this reflects the single tile L-Series reference board configuration):

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

Table 8: Clock Divides used in single tile L-Series Ref Design

The master clock must be supplied by an external source e.g. clock generator, fixed oscillators, PLL etc to generate the two frequencies to support 44.1kHz and 48kHz audio frequencies (e.g. 11.2896/22.5792MHz and 12.288/24.576MHz respectively). This master clock input is then provided to the CODEC and the XMOS device.

7.4.1 Port Configuration (xCORE Master)

The default software configuration is CODEC Slave (xCORE master). That is, the XMOS device provides the BCLK and LRCLK signals to the CODEC.

XS1 ports and XMOS clocks provide many valuable features for implementing I2S. This section describes how these are configured and used to drive the I2S interface.

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

The XMOS device 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`. The preceding diagram shows the connectivity of ports and clock blocks.

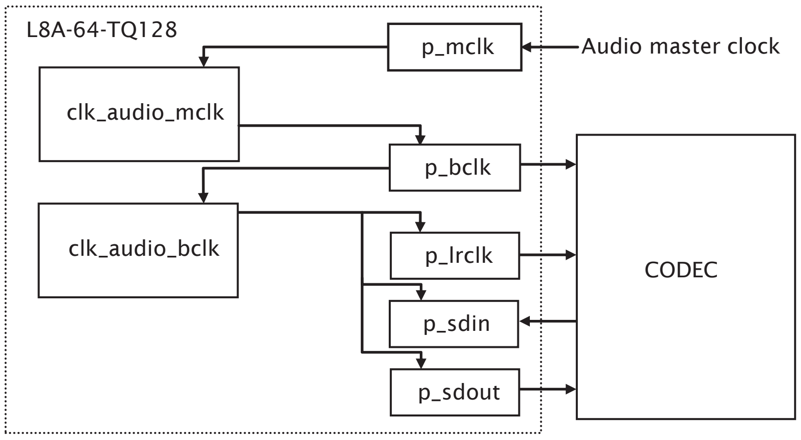

Figure 2: Ports and Clocks (CODEC slave)

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.

xCORE-200 series devices have the ability to divide an extenal clock in a clock-block. However, XS1 based devices do not have this functionality. In order achieve the reqired master-clock to bit-clock/LR-clock divicd on XS1 devices, buffered ports with a transfer width of 32 are also used for p_bclk and p_lrclk. The bit clock is generated by performing outputs of a particular pattern to p_bclk to toggle the output at the desired rate. The pattern depends on the divide between the master-clock and bit-clock. The following table shows the required pattern for different values of this divide:

Divide	**Output pattern**	**Outputs per sample**
2	0xAAAAAAAA	2
4	0xCCCCCCCC	4
8	0xF0F0F0F0	8

Table 9: Output patterns

In any case, 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.

7.4.2 Changing Audio Sample Frequency

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

First, a change of sample frequency is reported by sending the new frequency over an XC channel. The audio core 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 core stops the I2S/TDM interface and calls the CODEC/port configuration functions.

Once this is complete, the I2S/TDM interface is restarted at the new frequency.

7.5 Digital Mixer

The mixer core(s) take outgoing audio from the decoupler core and incoming audio from the audio driver core. It then applies the volume to each channel and passes incoming audio on to the decoupler and outgoing audio to the audio driver. The volume update is achieved using the built-in 32bit to 64bit signed multiply-accumulate function (macs). The mixer is implemented in the file `mixer.xc`.

The mixer takes two cores 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 moves down to two mixes when switching to a higher rate.

The mixer can take inputs from either:

- The USB outputs from the host—these samples come from the decoupler core.
- The inputs from the audio interface on the device—these samples come from the audio driver.

Since the sum of these inputs may be more then 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 output destinations:

- The USB inputs to the host—these samples are sent to the decoupler core.
- The outputs to the audio interface on the device—these samples are sent to the audio driver.

For each possible output, a mapping exists to tell the mixer what its source is. The possible sources are the USB outputs from the host, the inputs for the audio interface or the outputs from the mixer units.

As mentioned in §7.2.3, the mixer can also handle volume setting. If the mixer is configured to handle volume but the number of mixes is set to zero (so the component is solely doing volume setting) then the component will use only one core.

7.5.1 Control

The mixers can receive the following control commands from the Endpoint 0 core via a channel:

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.

Table 10: Mixer Component Commands

7.5.2 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.

For details, consult the README file in the host_usb_mixer_control directory.

The main requirements of this control are to

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

There is enough flexibility within this configuration that there will often be multiple ways of creating the required solution.

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

First consider the inputs to the mixer:

```
./xmos_mixer --display-aud-channel-map 0
```

displays which channels are mapped to which mixer inputs:

```
./xmos_mixer --display-aud-channel-map-sources 0
```

displays which channels could possibly be mapped to mixer inputs. Notice that analogue inputs 1 and 2 are on mixer inputs 10 and 11.

Now examine the audio output mapping:

```
./xmos_mixer --display-aud-channel-map 0
```

displays which channels are mapped to which outputs. By default all of these bypass the mixer. We can also see what all the possible mappings are:

```
./xmos_mixer --display-aud-channel-map-sources 0
```

So now map the first two mixer outputs to physical outputs 1 and 2:

```
./xmos_mixer --set-aud-channel-map 0 26
./xmos_mixer --set-aud-channel-map 1 27
```

You can confirm the effect of this by re-checking the map:

```
./xmos_mixer --display-aud-channel-map 0
```

This now makes analogue outputs 1 and 2 come from the mixer, rather than directly from USB. However the mixer is still mapped to pass the USB channels through to the outputs, so there will still be no functional change yet.

The mixer nodes need to be individually set. They can be displayed with:

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

To get the audio from the analogue inputs to outputs 1 and 2, nodes 80 and 89 need to be set:

```
./xmos_mixer --set-value 0 80 0
./xmos_mixer --set-value 0 89 0
```

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

```
./xmos_mixer --set-value 0 0 -inf
./xmos_mixer --set-value 0 9 -inf
```

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

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 1/2 outputs come from the analogue inputs.

To demonstrate this, firstly undo the changes above:

```
./xmos_mixer --set-value 0 80 -inf
./xmos_mixer --set-value 0 89 -inf
./xmos_mixer --set-value 0 0 0
./xmos_mixer --set-value 0 9 0
```

The mixer should now have the default values. The sources for mixer 1/2 can now be changed:

```
./xmos_mixer --set-mixer-source 0 0 10
./xmos_mixer --set-mixer-source 0 1 11
```

If you rerun:

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

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.

7.6 S/PDIF Transmit

XMOS devices can support S/PDIF transmit up to 192kHz. The XMOS S/SPDIF transmitter component runs in a single core and can be found in `sc_spdif/module_spdif_tx`

The S/PDIF transmitter core 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 core 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.

Note that a minor change to the `SpdifTransmitPortConfig` function would enable *internal* master clock generation (e.g. when clock source is already locked to desired audio clock).

Sample frequencies	44.1, 48, 88.2, 96, 176.4, 192 kHz
Master clock ratios	128x, 256x, 512x
Module	`module_spdif_tx`

Table 11: S/PDIF Capabilities

7.6.1 Clocking

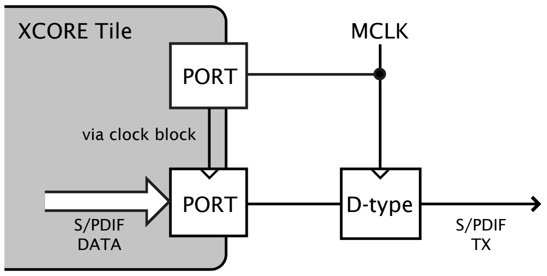

Figure 3: D-Type Jitter Reduction

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 24.576MHz.

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 the preceding diagram).

7.6.2 Usage

The interface to the S/PDIF transmitter core is via a normal channel with streaming built-ins (`outuint`, `inuint`). Data format should be 24-bit left-aligned in a 32-bit word: 0x12345600

The following protocol is used on the channel:

`outuint`	Sample frequency (Hz)
`outuint`	Master clock frequency (Hz)
`outuint`	Left sample
`outuint`	Right sample
`outuint`	Left sample
`outuint`	Right sample
`...`	
`...`	
`outct`	Terminate

Table 12: S/PDIF Component Protocol

7.6.3 Output stream structure

The stream is composed of words with the following structure shown in Table 13. 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 14.

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 13: S/PDIF Stream Structure

Frequency (kHz)	n
44.1	0
48	2
88.2	8
96	A
176.4	C
192	E

Table 14: Channel Status Bits

7.7 S/PDIF Receive

XMOS devices can support S/PDIF receive up to 192kHz.

The S/PDIF receiver module uses a clockblock and a buffered one-bit port. The clock-block is divided of a 100 MHz reference clock. The one bit port is buffered to 4-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.

The S/PDIF receive function never returns. The 32-bit value from the channel input comprises:

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

Table 15: S/PDIF RX Word Structure

The tag has one of three values:

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.

Table 16: S/PDIF RX Tags

See S/PDIF specification for further details on format, user bits etc.

7.7.1 Usage and Integration

Since S/PDIF is a digital steam the devices master clock must be syncronised to it. This is typically done with an external fractional-n multipier. See *Clock Recovery* (§7.9)

The S/PDIF receive function communicates with the `clockGen` component with passes audio data to the audio driver and handles locking to the S/PDIF clock source if required (see External Clock Recovery).

Ideally the parity of each word/sample received should be checked. This is done using the built in `crc32` function (see `xs1.h`):

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 could be used:

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

   if(badParity(data)
     continue;

   tag = data & 0xF;

   /* Extract 24bit audio sample */
   sample = (data << 4) & 0xFFFFFF00;

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

     case FRAME_Z:
       // Store right
       break;
   }
}
```

7.8 ADAT Receive

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 XM-005512-PC.

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

Example of code show 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.

7.8.1 Integration

Since the ADAT is a digital stream the devices master clock must synchronised to it. This is typically achieved with an external fractional-n clock multiplier.

The ADAT receive function communicates with the clockGen component which passes audio data onto the audio driver and handles locking to the ADAT clock source if required.

7.9 External Clock Recovery (ClockGen)

An application can either provide fixed master clock sources via selectable oscillators, clock generation IC, etc, to provide the audio master or use an external PLL/Clock Multiplier to generate a master clock based on reference from the XMOS device.

Using an external PLL/Clock Multiplier allows the design to lock to an external clock source from a digital stream (e.g. S/PDIF or ADAT input).

The clock recovery core (clockGen) is responsible for generating the reference frequency to the Fractional-N Clock Generator. This, in turn, generates the master clock used over the whole design.

When running in *Internal Clock* mode this core simply generates this clock using a local timer, based on the XMOS reference clock.

When running in an external clock mode (i.e. S/PDIF Clock" or "ADAT Clock" mode) digital samples are received from the S/PDIF and/or ADAT receive core.

The external frequency is calculated through counting samples in a given period. The reference clock to the Fractional-N Clock Multiplier is then generated based on this external stream. If this stream becomes invalid, the timer event will fire to ensure that valid master clock generation continues regardless of cable unplugs etc.

This core gets clock selection Get/Set commands from Endpoint 0 via the `c_clk_ctl` channel. This core also records the validity of external clocks, which is also queried through the same channel from Endpoint 0.

This core also can cause the decouple core to request an interrupt packet on change of clock validity. This functionality is based on the Audio Class 2.0 status/interrupt endpoint feature.

7.10 MIDI

The MIDI driver implements a 31250 baud UART input and output. On receiving 32-bit USB MIDI events from the `buffer` core, it parses these and translates them to 8-bit MIDI messages which are sent over UART. Similarly, incoming 8-bit MIDI messages are aggregated into 32-bit USB-MIDI events an passed on to the `buffer` core. The MIDI core is implemented in the file `usb_midi.xc`.

7.11 PDM Microphones

7.12 Overview of PDM implemention

The design is capable of integrating 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 with efficient decimation to user selectable output sample rates.

The `lib_mic_array` library is only available for xCORE-200 series devices.

The following components of the library are used:

- PDM interface
- Four channel decimators

Up to sixteen PDM microphones can be attached to each high channel count PDM interface (`mic_array_pdm_rx()`). One to four processing tasks, `mic_array_decimate_to_pcm_4ch()`, each process up to four channels. For 1-4 channels the library requires two logical cores:

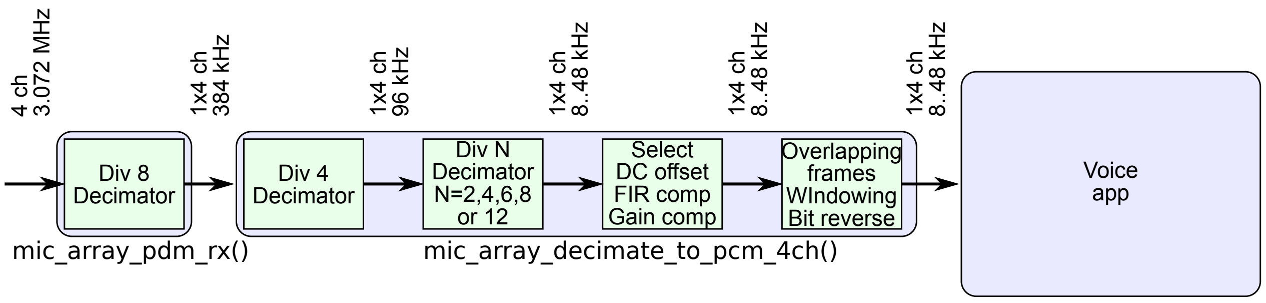

Figure 4: One to four channel count PDM interface

for 5-8 channels three logical cores are required, as shown below:

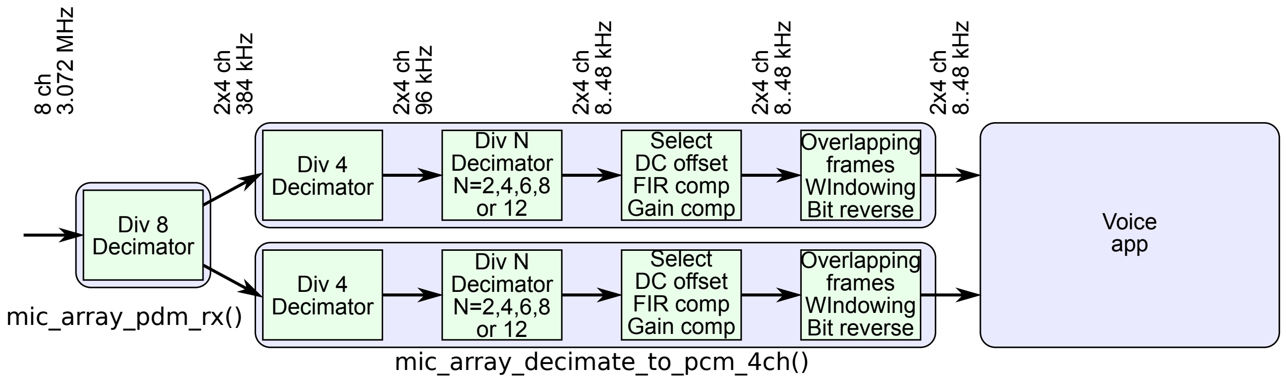
8 ch 3.072 MHz
2x4 ch 384 kHz
2x4 ch 96 kHz
2x4 ch 8..48 kHz
2x4 ch 8..48 kHz
2x4 ch 8..48 kHz
Div 8 Decimator
Div 4 Decimator
Div N Decimator N=2,4,6,8 or 12
Select DC offset FIR comp Gain comp
Overlapping frames WIndowing Bit reverse
Div 4 Decimator
Div N Decimator N=2,4,6,8 or 12
Select DC offset FIR comp Gain comp
Overlapping frames WIndowing Bit reverse
Voice app
mic_array_pdm_rx()
mic_array_decimate_to_pcm_4ch()

Figure 5: Five to eight count PDM interface

The left most task, `mic_array_pdm_rx()`, samples up to 8 microphones and filters the data to provide up to eight 384 KHz data streams, split in two streams of four channels. The processing thread decimates the signal to a user chosen sample rate (one of 48, 24, 16, 12 or 8 KHz).

More channels can be supported by increasing the number of cores dedicated to the PDM tasks. However, the current PDM mic integration into USB Audio limits itself to 8.

After the decimation to the output sample-rate various other steps take place e.g. DC offset elimination, gain correction and compensation etc. Please refer to `lib_mic_array` documention for further implementation detail and complete feature set.

7.12.1 PDM Microphone Hardware Characteristics

The PDM microphones need a *clock input* and provide the PDM signal on a *data output*. All PDM microphones share the same clock signal (buffered on the PCB as appropriate), and output onto eight data wires that are connected to a single 8-bit port:

CLOCK	Clock line, the PDM clock the used by the microphones to drive the data out.
DQ_PDM	The data from the PDM microphones on an 8 bit port.

Table 17: PDM microphone data and signal wires

The only port that is passed into `lib_mic_array` is the 8-bit data port. The library assumes that the input port is clocked using the PDM clock and requires no knowlege of the PDM clock source.

The input clock for the microphones can be generated in a multitude of ways. For example, a 3.072MHz clock can be generated on the board, or the xCORE can divide down 12.288 MHz master clock. Or, if clock accuracy is not important, the internal 100 MHz reference can be divided down to provide an approximate clock.

7.12.2 Integration of PDM Microphones into USB Audio

A PDM microphone wrapper is called from `main()` and takes one channel argument connecting it to the rest of the system:

```
pcm_pdm_mic(c_pdm_pcm);
```

The implemetation of this function can be found in the file `pcm_pdm_mics.xc`.

The first job of this function is to configure the ports/clocking for the microphones, this divides the external audio master clock input (on port `p_mclk`) and outputs the divided clock to the microphones via the `p_pdm_clk` port:

```
configure_clock_src_divide(pdmclk, p_mclk, MCLK_TO_PDM_CLK_DIV);
configure_port_clock_output(p_pdm_clk, pdmclk);
configure_in_port(p_pdm_mics, pdmclk);
start_clock(pdmclk);
```

It then runs the various cores required for the PDM interface and PDM to PCM conversion as discussed previously:

```
par
{
    mic_array_pdm_rx(p_pdm_mics, c_4x_pdm_mic_0, c_4x_pdm_mic_1);
    mic_array_decimate_to_pcm_4ch(c_4x_pdm_mic_0, c_ds_output[0]);
    mic_array_decimate_to_pcm_4ch(c_4x_pdm_mic_1, c_ds_output[1]);
    pdm_process(c_ds_output, c_pcm_out);
}
```

The `pdm_process()` task includes the main integration code, it takes audio from the `lib_mic_array` cores, buffers it, performs optional local processing and outputs it to the audio driver (TDM/I2S core).

This function simply makes a call to `mic_array_get_next_time_domain_frame()` in order to get a frame of PCM audio from the microphones. It then waits for an request for audio samples from the audio/I2S/TDM core via a channel and sends the frame of audio back over this channel.

Note, it is assumed that the system shares a global master-clock, therefore no additional buffering or rate-matching/conversion is required.

7.13 Resource Usage

The following table details the resource usage of each component of the reference design software.

Component	Cores	Memory (KB)	Ports
XUD library	1	9 (6 code)	ULPI ports
Endpoint 0	1	17.5 (10.5 code)	none
USB Buffering	1	22.5 (1 code)	none
Audio driver	1	8.5 (6 code)	See §7.4
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	2	8.7 (6.5 code)	
ClockGen	1	2.5 (2.4 code)	

Table 18: Resource Usage

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.

The XUD library requires an 80MIPS core to function correctly (i.e. on a 500MHz part only six cores can run).

The ULPI ports are a fixed set of ports on the L-Series device. When using these ports, other ports are unavailable when ULPI is active. See the XS1-L Hardware Design Checklist[7] for further details.

[7] http://www.xmos.com/published/xs1lcheck

8 Known Issues

- Quad-SPI DFU will corrupt the factory image with tools version < 14.0.4 due to an issue with libquad-flash
- (#14762) When in DSD mode with S/PDIF output enabled, DSD samples are transmitted over S/PDIF if the DSD and S/PDIF channels are shared, this may or may not be desired
- (#14173) I2S input is completely disabled when DSD output is active - any input stream to the host will contain 0 samples
- (#14780) Operating the design at a sample rate of less than or equal to the SOF rate (i.e. 8kHz at HS, 1kHz at FS) may expose a corner case relating to 0 length packet handling in both the driver and device and should be considered un-supported at this time.
- (#14883) Before DoP mode is detected a small number of DSD samples will be played out as PCM via I2S
- (#14887) Volume control settings currently affect samples in both DSD and PCM modes. This results in invalid DSD output if volume control not set to 0
- Windows XP volume control very sensitive. The Audio 1.0 driver built into Windows XP (usbaudio.sys) does not properly support master volume AND channel volume controls, leading to a very sensitive control. Descriptors can be easily modified to disable master volume control if required (one byte - bmaControls(0) in Feature Unit descriptors)
- 88.2kHz and 176.4kHz sample frequencies are not exposed in Windows control panels. These are known OS restrictions.

9 lib_xua Change Log

9.1 0.2.0

- ADDED: Documentation
- CHANGE: I2S hardware resources no longer used globally and must be passed to XUA_AudioHub()
- CHANGE: NO_USB define renamed to XUA_USB_EN

9.2 0.1.2

- ADDED: Application note AN00246
- CHANGE: xmosdfu emits warning if empty image read via upload
- CHANGE: Simplified mclk port sharing - no longer uses unsafe pointer
- RESOLVED: Runtime exception issues when incorrect feedback calculated (introduced in sc_usb_audio 6.13)
- RESOLVED: Output sample counter reset on stream start. Caused playback issues on some Linux based hosts

9.3 0.1.1

- RESOLVED: Configurations where I2S_CHANS_DAC and I2S_CHANS_ADC are both 0 now build
- RESOLVED: Deadlock in mixer when MAX_MIX_COUNT > 0 for larger channel counts
- Changes to dependencies:
  - lib_logging: Added dependency 2.1.1
  - lib_xud: Added dependency 0.1.0

9.4 0.1.0

- ADDED: FB_USE_REF_CLOCK to allow feedback generation from xCORE internal reference
- ADDED: Linux Makefile for xmosdfu host application
- ADDED: Raspberry Pi Makefile for xmosdfu host application
- ADDED: Documentation of PID argument to xmosdfu
- ADDED: Optional build time microphone delay line (MIC_BUFFER_DEPTH)
- CHANGE: Removal of audManage_if, users should define their own interfaces as required
- CHANGE: Vendor specific control interface in UAC1 descriptor now has a string descriptor so it shows up with a descriptive name in Windows Device Manager
- CHANGE: DFU_BCD_DEVICE removed (now uses BCD_DEVICE)
- CHANGE: Renaming in descriptors.h to avoid clashes with application
- CHANGE: Make device reboot function no-argument (was one channel end)
- RESOLVED: FIR gain compensation for PDM mics set incorrectly for divide of 8
- RESOLVED: Incorrect xmosdfu DYLD path in test script code
- RESOLVED: xmosdfu cannot find XMOS device on modern MacBook Pro (#17897)
- RESOLVED: Issue when feedback is initially incorrect when two SOF's are not yet received
- RESOLVED: AUDIO_TILE and PDM_TILE may now share the same value/tile
- RESOLVED: Cope with out of order interface numbers in xmosdfu
- RESOLVED: DSD playback not functional on xCORE-200 (introduced in sc_usb_audio 6.14)
- RESOLVED: Improvements made to clock sync code in TDM slave mode

9.5 Legacy release history

(Note: Forked from sc_usb_audio at this point)

9.6 7.4.1

- RESOLVED: Exception due to null chanend when using NO_USB

9.7 7.4.0

- RESOLVED: PID_DFU now based on AUDIO_CLASS. This potentially caused issues with UAC1 DFU

9.8 7.3.0

- CHANGE: Example OSX DFU host app updated to now take PID as runtime argument. This enabled multiple XMOS devices to be attached to the host during DFU process

9.9 7.2.0

- ADDED: DFU to UAC1 descriptors (guarded by DFU and FORCE_UAC1_DFU)
- RESOLVED: Removed 'reinterpretation to type of larger alignment' warnings
- RESOLVED: DFU flash code run on tile[0] even if XUD_TILE and AUDIO_IO_TILE are not 0

9.10 7.1.0

- ADDED: UserBufferManagementInit() to reset any state required in UserBufferManagement()
- ADDED: I2S output up-sampling (enabled when AUD_TO_USB_RATIO is > 1)
- ADDED: PDM Mic decimator output rate can now be controlled independently (via AUD_TO_MICS_RATIO)
- CHANGE: Rename I2S input down-sampling (enabled when AUD_TO_USB_RATIO is > 1, rather than via I2S_DOWNSAMPLE_FACTOR)
- RESOLVED: Crosstalk between input channels when I2S input down-sampling is enabled
- RESOLVED: Mic decimation data tables properly sized when mic sample-rate < USB audio sample-rate

9.11 7.0.1

- RESOLVED: PDM microphone decimation issue at some sample rates caused by integration

9.12 7.0.0

- ADDED: I2S down-sampling (I2S_DOWNSAMPLE_FACTOR)
- ADDED: I2S resynchronisation when in slave mode (CODEC_MASTER=1)
- CHANGE: Various memory optimisations when MAX_FREQ = MIN_FREQ
- CHANGE: Memory optimisations in audio buffering
- CHANGE: Various memory optimisations in UAC1 mode
- CHANGE: user_pdm_process() API change
- CHANGE: PDM Mic decimator table now related to MIN_FREQ (memory optimisation)
- RESOLVED: Audio request interrupt handler properly eliminated

9.13 6.30.0

- **RESOLVED: Number of PDM microphone channels configured now based on NUM_PDM_MICS define** (previously hard-coded)
- RESOLVED: PDM microphone clock divide now based MCLK defines (previously hard-coded)
- CHANGE: Second microphone decimation core only run if NUM_PDM_MICS > 4

9.14 6.20.0

- RESOLVED: Intra-frame sample delays of 1/2 samples on input streaming in TDM mode
- RESOLVED: Build issue with NUM_USB_CHAN_OUT set to 0 and MIXER enabled
- RESOLVED: SPDIF_TX_INDEX not defined build warning only emitted when SPDIF_TX defined
- RESOLVED: Failure to enter DFU mode when configured without input volume control

9.15 6.19.0

- RESOLVED: SPDIF_TX_INDEX not defined build warning only emitted when SPDIF_TX defined
- RESOLVED: Failure to enter DFU mode when configured without input volume control

9.16 6.18.1

- **ADDED: Vendor Specific control interface added to UAC1 descriptors to allow control of** XVSM params from Windows (via lib_usb)

9.17 6.18.0

- ADDED: Call to VendorRequests() and VendorRequests_Init() to Endpoint 0
- ADDED: VENDOR_REQUESTS_PARAMS define to allow for custom parameters to VendorRequest calls
- RESOLVED: FIR gain compensation set appropriately in lib_mic_array usage
- CHANGE: i_dsp interface renamed i_audManage

9.18 6.16.0

- ADDED: Call to UserBufferManagement()
- ADDED: PDM_MIC_INDEX in devicedefines.h and usage
- CHANGE: pdm_buffer() task now combinable
- CHANGE: Audio I/O task now takes i_dsp interface as a parameter
- CHANGE: Removed built-in support for A/U series internal ADC
- CHANGE: User PDM Microphone processing now uses an interface (previously function call)

9.19 6.15.2

- RESOLVED: interrupt.h (used in audio buffering) now compatible with xCORE-200 ABI

9.20 6.15.1

- RESOLVED: DAC data mis-alignment issue in TDM/I2S slave mode
- CHANGE: Updates to support API changes in lib_mic_array version 2.0

9.21 6.15.0

- **RESOLVED: UAC 1.0 descriptors now support multi-channel volume control (previously were** hard-coded as stereo)
- **CHANGE: Removed 32kHz sample-rate support when PDM microphones enabled (lib_mic_array** currently does not support non-integer decimation factors)

9.22 6.14.0

- **ADDED: Support for for master-clock/sample-rate divides that are not a power of 2** (i.e. 32kHz from 24.567MHz)
- **ADDED: Extended available sample-rate/master-clock ratios. Previous restriction was <=** 512x (i.e. could not support 1024x and above e.g. 49.152MHz MCLK for Sample Rates below 96kHz) (#13893)
- **ADDED: Support for various "low" sample rates (i.e. < 44100) into UAC 2.0 sample rate** list and UAC 1.0 descriptors
- **ADDED: Support for the use and integration of PDM microphones (including PDM to PCM** conversion) via lib_mic_array
- **RESOLVED: MIDI data not accepted after "sleep" in OSX 10.11 (El Capitan) - related to sc_xud** issue #17092
- **CHANGE: Asynchronous feedback system re-implemented to allow for the first two ADDED** changelog items
- **CHANGE: Hardware divider used to generate bit-clock from master clock (xCORE-200 only).** Allows easy support for greater number of master-clock to sample-rate ratios.
- CHANGE: module_queue no longer uses any assert module/lib

9.23 6.13.0

- **ADDED: Device now uses implicit feedback when input stream is available (previously explicit** feedback pipe always used). This saves chanend/EP resources and means less processing burden for the host. Previous behaviour available by enabling UAC_FORCE_FEEDBACK_EP
- **RESOLVED: Exception when SPDIF_TX and ADAT_TX both enabled due to clock-block being configured** after already started. Caused by SPDIF_TX define check typo
- **RESOLVED: DFU flag address changed to properly conform to memory address range allocated to** apps by tools
- RESOLVED: Build failure when DFU disabled
- RESOLVED: Build issue when I2S_CHANS_ADC/DAC set to 0 and CODEC_MASTER enabled
- RESOLVED: Typo in MCLK_441 checking for MIN_FREQ define
- CHANGE: Mixer and non-mixer channel comms scheme (decouple <-> audio path) now identical
- **CHANGE: Input stream buffering modified such that during overflow older samples are removed** rather than ignoring most recent samples. Removes any chance of stale input packets being sent to host
- CHANGE: module_queue (in sc_usb_audio) now uses lib_xassert rather than module_xassert

9.24 6.12.6

- RESOLVED: Build error when DFU is disabled
- RESOLVED: Build error when I2S_CHANS_ADC or I2S_CHANS_DAC set to 0 and CODEC_MASTER enabled

9.25 6.12.5

- RESOLVED: Stream issue when NUM_USB_CHAN_IN < I2S_CHANS_ADC

9.26 6.12.4

- RESOLVED: DFU fail when DSD enabled and USB library not running on tile[0]

9.27 6.12.3

- **RESOLVED: Method for storing persistent state over a DFU reboot modified to improve resilience** against code-base and tools changes

9.28 6.12.2

- RESOLVED: Reboot code (used for DFU) failure in tools versions > 14.0.2 (xCORE-200 only)
- RESOLVED: Run-time exception in mixer when MAX_MIX_COUNT > 0 (xCORE-200 only)
- RESOLVED: MAX_MIX_COUNT checked properly for mix strings in string table
- **CHANGE: DFU code re-written to use an XC interface. The flash-part may now be connected** to a separate tile to the tile running USB code
- CHANGE: DFU code can now use quad-SPI flash
- **CHANGE: Example xmos_dfu application now uses a list of PIDs to allow adding PIDs easier.** –listdevices command also added.
- CHANGE: I2S_CHANS_PER_FRAME and I2S_WIRES_xxx defines tidied

9.29 6.12.1

- RESOLVED: Fixes to TDM input timing/sample-alignment when BCLK=MCLK
- RESOLVED: Various minor fixes to allow ADAT_RX to run on xCORE 200 MC AUDIO hardware
- CHANGE: Moved from old SPDIF define to SPDIF_TX

9.30 6.12.0

- ADDED: Checks for XUD_200_SERIES define where required
- **RESOLVED: Run-time exception due to decouple interrupt not entering correct issue mode** (affects XCORE-200 only)
- CHANGE: SPDIF Tx Core may now reside on a different tile from I2S
- CHANGE: I2C ports now in structure to match new module_i2c_singleport/shared API.

- Changes to dependencies:
  - sc_util: 1.0.4rc0 -> 1.0.5alpha0
    - xCORE-200 Compatiblity fixes to module_locks

9.31 6.11.3

- RESOLVED: (Major) Streaming issue when mixer not enabled (introduced in 6.11.2)

9.32 6.11.2

- **RESOLVED: (Major) Enumeration issue when MAX_MIX_COUNT > 0 only. Introduced in mixer** optimisations in 6.11.0. Only affects designs using mixer functionality.
- **RESOLVED: (Normal) Audio buffering request system modified such that the mixer output is**

not silent when in underflow case (i.e. host output stream not active) This issue was introduced with the addition of DSD functionality and only affects designs using mixer functionality.

- **RESOLVED: (Minor) Potential build issue due to duplicate labels in inline asm in** set_interrupt_handler macro
- **RESOLVED: (Minor) BCD_DEVICE define in devicedefines.h now guarded by ifndef (caused issues** with DFU test build configs.
- RESOLVED: (Minor) String descriptor for Clock Selector unit incorrectly reported
- **RESOLVED: (Minor) BCD_DEVICE in devicedefines.h now guarded by #ifndef (Caused issues with** default DFU test build configs.
- CHANGE: HID report descriptor defines added to shared user_hid.h
- CHANGE: Now uses module_adat_rx from sc_adat (local module_usb_audio_adat removed)

9.33 6.11.1

- ADDED: ADAT transmit functionality, including SMUX. See ADAT_TX and ADAT_TX_INDEX.
- RESOLVED: (Normal) Build issue with CODEC_MASTER (xCore is I2S slave) enabled
- RESOLVED: (Minor) Channel ordering issue in when TDM and CODEC_MASTER mode enabled
- **RESOLVED: (Normal) DFU fails when SPDIF_RX enabled due to clock block being shared between SPDIF** core and FlashLib

9.34 6.11.0

- ADDED: Basic TDM I2S functionality added. See I2S_CHANS_PER_FRAME and I2S_MODE_TDM
- **CHANGE: Various optimisations in 'mixer' core to improve performance for higher** channel counts including the use of XC unsafe pointers instead of inline ASM
- CHANGE: Mixer mapping disabled when MAX_MIX_COUNT is 0 since this is wasted processing.
- **CHANGE: Descriptor changes to allow for channel input/output channel count up to 32** (previous limit was 18)

9.35 6.10.0

- **CHANGE: Endpoint management for iAP EA Native Transport now merged into buffer() core.** Previously was separate core (as added in 6.8.0).
- CHANGE: Minor optimisation to I2S port code for inputs from ADC

9.36 6.9.0

- **ADDED: ADAT S-MUX II functionality (i.e. 2 channels at 192kHz) - Previously only S-MUX** supported (4 channels at 96kHz).
- **ADDED: Explicit build warnings if sample rate/depth & channel combination exceeds** available USB bus bandwidth.
- **RESOLVED: (Major) Reinstated ADAT input functionality, including descriptors and clock** generation/control and stream configuration defines/tables.
- **RESOLVED: (Major) S/PDIF/ADAT sample transfer code in audio() (from ClockGen()) moved to** aid timing.
- **CHANGE: Modifying mix map now only affects specified mix, previous was applied to all** mixes. CS_XU_MIXSEL control selector now takes values 0 to MAX_MIX_COUNT + 1 (with 0 affecting all mixes).
- **CHANGE: Channel c_dig_rx is no longer nullable, assists with timing due to removal of** null checks inserted by compiler.
- **CHANGE: ADAT SMUX selection now based on device sample frequency rather than selected** stream format - Endpoint 0 now configures clockgen() on a sample-rate change rather than

stream start.

9.37 6.8.0

- ADDED: Evaluation support for iAP EA Native Transport endpoints
- **RESOLVED: (Minor) Reverted change in 6.5.1 release where sample rate listing in Audio Class** 1.0 descriptors was trimmed (previously 4 rates were always reported). This change appears to highlight a Windows (only) enumeration issue with the Input & Output configs
- **RESOLVED: (Major) Mixer functionality re-instated, including descriptors and various required** updates compatibility with 13 tools
- RESOLVED: (Major) Endpoint 0 was requesting an out of bounds channel whilst requesting level data
- **RESOLVED: (Major) Fast mix code not operates correctly in 13 tools, assembler inserting long jmp** instructions
- RESOLVED: (Minor) LED level meter code now compatible with 13 tools (shared mem access)
- **RESOLVED (Minor) Ordering of level data from the device now matches channel ordering into** mixer (previously the device input data and the stream from host were swapped)
- CHANGE: Level meter buffer naming now resemble functionality

9.38 Legacy release history

Please see changelog in sw_usb_audio for changes prior to 6.8.0 release.