USB Audio Design Guide

Document Number: XM0088546.1

Publication Date: 2016/5/12

XMOS

SYNOPSIS

The XMOS USB Audio solution provides *USB Audio Class* compliant devices over USB 2.0 (high-speed or full-speed). Based on the XMOS XS1 architecture, it supports USB Audio Class 2.0 and USB Audio Class 1.0, asynchronous mode and sample rates up to 384kHz.

The complete source code, together with the free XMOS xTIMEcomposer development tools and XCORE multi-core micro-controller devices allow the implementer to select the exact mix of interfaces and processing required.

The XMOS USB Audio solution is deployed as a framework with reference design applications extending and customising this framework. These reference designs have particular qualified feature sets and an accompanying reference hardware platform.

This software design guide assumes the reader is familiar with the XC language and xCORE devices. For more information see XMOS Programming Guide[1].

The reader should also familiarise themselves with the XMOS USB Device Library[2] and the XMOS USB Device Design Guide[3]

The reader should always refer to the supplied CHANGELOG and README files for known issues etc in a specific release

[1] https://www.xmos.com/published/xmos-programming-guide
[2] http://www.xmos.com/published/xuddg
[3] https://www.xmos.com/zh/node/17007?page=9

Table of Contents

1 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[4]
	USB Audio Class 2.0[5]
	USB Firmware Upgrade (DFU) 1.1[6]
	USB Midi Device Class 1.0[7]
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
	XS1 L-Series
	XS1 U-Series
	XS1 G-Series (Not recommended for new designs)
Requirements	
Development Tools	xTIMEcomposer Development Tools v14 or later
USB	External ULPI USB Phy (If using XS1 G/L-Series)
Audio	External audio DAC/ADC/CODECs (and required supporting componentry) supporting I2S/TDM
Boot/Storage	Compatible SPI Flash device
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.	

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

2 Hardware Platforms

IN THIS CHAPTER

The following sections describe the hardware platforms that support development with the XMOS USB Audio software framework.

2.1 xCORE-200 Multi-Channel Audio Board

The XMOS xCORE-200 Multi-channel Audio board[8] (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 Multichannel Audio Platform supports either USB or network audio source, streaming 8 analogue input and 8 analogue output audio channels simultaneously - at up to 192kHz. Ideal for mixing two sources and providing main and headphone monitor output feeds.

For full details regarding the hardware please refer to xCORE-200 Multichannel Audio Platform Hardware Manual[9].

The reference board has an associated firmware application that uses the USB Audio 2.0 software reference platform. Details of this application can be found in section §6.6.

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

2.1.1 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. The jumper allows channel selection when the ADC/DAC is used in TDM mode

2.1.2 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.

2.1.3 MIDI

MIDI I/O is provided on the board via a standard Gameport connector. 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.

2.1.4 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.

2.1.5 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.

2.2 xCORE-200 Microphone Array Board

The XMOS xCORE-200 Microphone Array board[10] (XK-USB-MIC-UF216) is design available from XMOS based on a dual-tile XMOS xCORE-200 device.

The board integrates the following building blocks: multiple omni-directional microphones, on-board low-jitter clock sources, configurable user input buttons and a USB2.0 device for connectivity. making it an ideal platform for a range of multichannel microphone aggregation products.

The board is powered by an XUF216-512 xCORE-200 multicore microcontroller. This device has sixteen 32bit logical cores that deliver up to 2000MIPS completely deterministically. In addition the XUF216 has powerful DSP properyies with native 32bit/64 instructions delivering up to 1000MMACS.

Figure 1 shows the block layout of the xCORE-200 Microhone Array board.

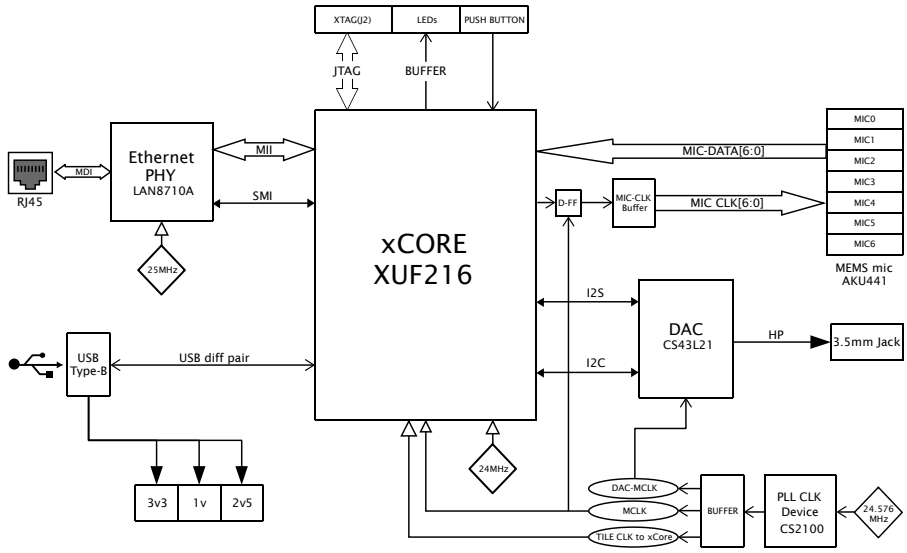

Figure 1: xCORE-200 Microphone Array Board Block Diagram

For full details regarding the hardware please refer to xCORE Microphone Array Hardware Manual[11].

The reference board has an associated firmware application that uses the USB Audio 2.0 software reference platform. Details of this application can be found in section §6.7.

[10] https://www.xmos.com/support/boards?product=20258
[11] https://www.xmos.com/download/private/xCORE-Microphone-Array-Hardware-Manual%281v1%29.pdf

2.2.1 Microphones

The xCORE Microphone Array board features 7 MEMS microphones with PDM (Pulse Density Modulation) output.

Figure 2 shows the microphone arrangement on the board.

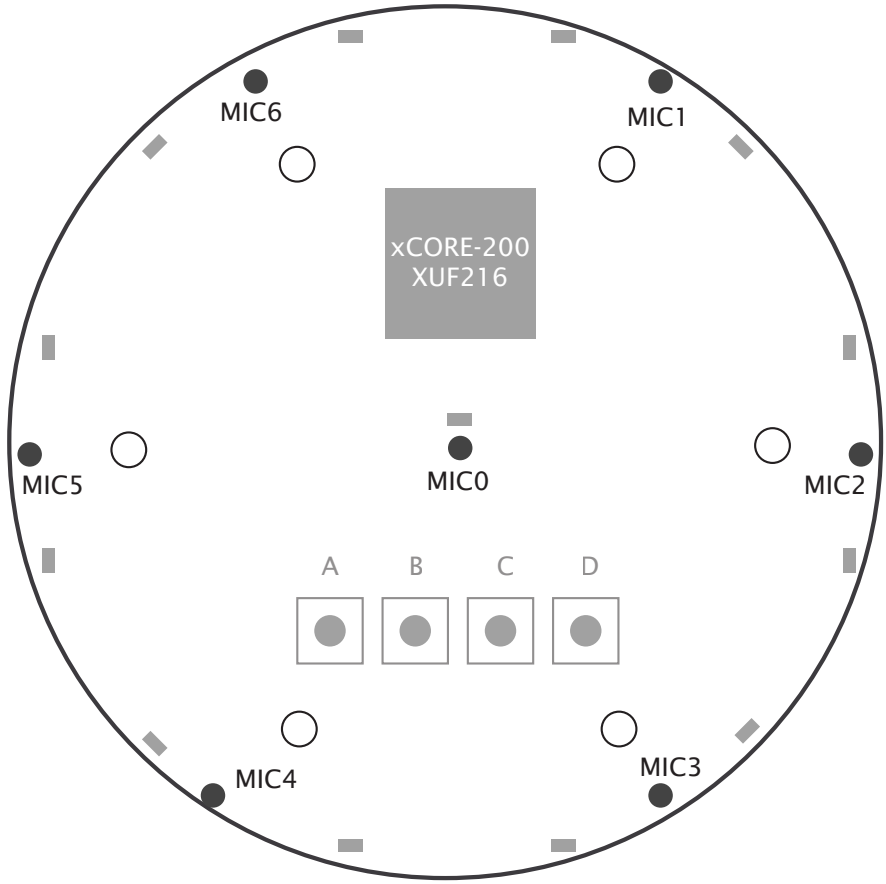

Figure 2: xCORE-200 Microphone Array Board Microphone Arrangement

2.2.2 Analogue Output

As well at 7 PDM microphones the board also provides a stereo DAC (CS43L21) with integrated headphone amplifier. The CS43L21 is connected to the xCORE-200 through an I2S interface and is configured using an I2C interface.

2.2.3 Audio Clocking

The board provides a low-jitter clock-source, an 24.576MHz oscillator, to serve as reference clock to the CS2100-CP (Cirrus Logic) Fractional-N PLL (U22).

The CS2100 generates a low-jitter output signal that is distributed to the xCORE-200 device and DAC. The CS2100 device is configured using the I2C interface.

2.2.4 Buttons, LEDs and Other IO

The board has 13 LEDs that are controlled by the xCORE-200 GPIO. The layout of the LEDs is shown in Figure 3.

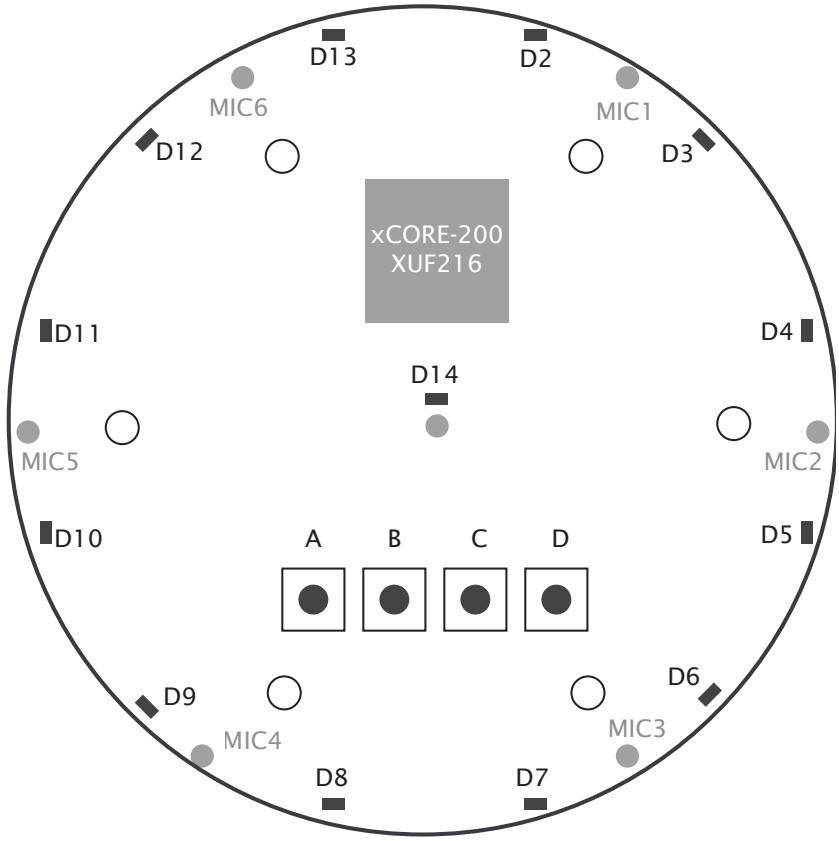

Figure 3: xCORE-200 Microphone Array Board LED Arrangement

LED 0 to LED 11 (D2-D13) are positioned around the edge of the board, one each side of every microphone. LED 12 (D14) is positioned next to the middle microphone.

A green LED (PGOOD) by the USB connector indicates a 3V3 power good signal.

Four general purpose push-button switches are provided. When pressed, each button creates a connection from the I/O to GND.

A standard XMOS xSYS interface (J2) is provided to allow host debug of the board via JTAG.

The board also includes Ethernet conextivity, however, this is outside the scope of this documentation.

2.3 USB Multi-function Audio (MFA) Kit

It is not recommended to use this hardware as a basic for a new design

The XMOS Multi-function Audio kit[12] (XK-USB-AUDIO-U8-2C-AB) is a hardware reference design available from XMOS based on a single tile XMOS U-series device.

- A main board which includes the XMOS U-series device and all audio hardware
- A "USB Slice" board which contains USB connectivity

The separate USB slice board allows flexibility in the connection method to the USB audio source/sink as well as other functionality such as 3rd party authentication ICs and any required USB switching. This also means the XMOS device can be used as a USB device or host using the same main board.

This document addresses the combination of the main board with the USB AB slice (part numbers XK-USB-AUDIO-U8-2C and XA-SK-USB-AB respectively). This provides a standard USB Audio device hardware configuration using the B socket on the USB AB slice.

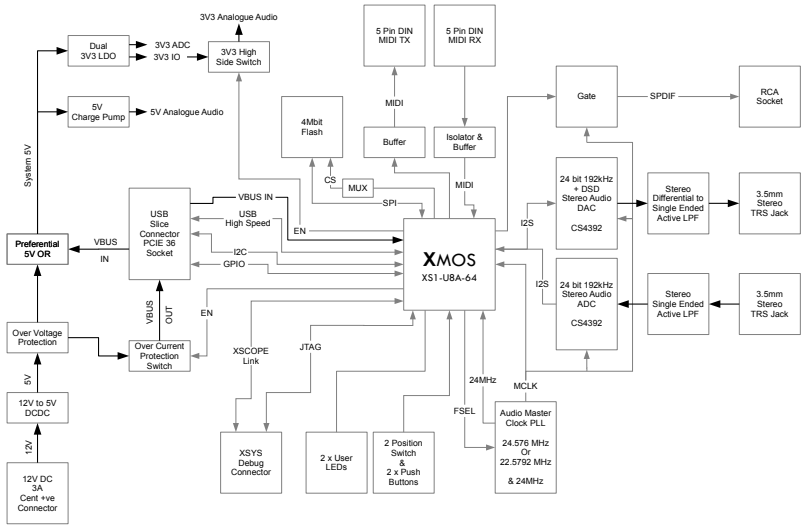

Figure 4: Multi-function Audio Kit Block Diagram

The core board includes a U-Series device with integrated USB PHY, a stereo DAC (with support for Direct Stream Digital) and a stereo ADC. Both ADC and DAC

[12] http://www.xmos.com/products/reference-designs/mfa

support sample frequencies up to 192kHz. As well as analogue channels the main board also has MIDI input and output connectors and a COAX connector for S/PDIF output.

In addition the main board also includes two LEDs, two buttons and one two-position switch for use by the user application.

2.4 U16 Multi-Channel USB Audio Kit

It is not recommended to use this hardware as a basic for a new design

The XMOS U16 Multi-Channel USB Audio kit[13] is a hardware development platform available from XMOS based on a dual tile XMOS U-series device.

- A sliceKIT core board which includes the XMOS U-series device (XP-SKC-U16)
- A "USB Slice" board which contains USB connectivity (XA-SK-USB-AB)
- A double-slot slice card including audio hardware and connectors (XA-SK-AUDIO8)

The separate USB slice board allows flexibility in the connection method to the USB audio source/sink as well as other functionality such as 3rd party authentication ICs and any required USB switching. This also means the XMOS device can be used as a USB device or host using the same main board.

This document addresses the combination of the main board with the USB AB slice (part numbers XP-SKC-U16 and XA-SK-USB-AB respectively). This provides a standard USB Audio device hardware configuration using the B socket on the USB AB slice.

The core board includes a U-Series device with integrated USB PHY and required supporting componentry.

Please note, for correct operation the following core-board jumper settings are required:

- J14 (DIA/ALT) should be set to ALT
- J15 (D12 XOVER) should be set to ON

The double-slot audio slice (XA-SK-AUDIO8) includes separate multi-channel DAC and ADC providing 8 channels of both analogue output and input. Both DAC and ADC devices support sample frequencies up to 192kHz with the DAC supporting Direct Stream Digital (DSD).

As well as analogue channels the audio-slice also has MIDI input and output connectors and both COAX and optical connectors for digital output.

Additionally the slice also includes an LED matrix and three push-buttons for use by the user application.

[13] http://www.xmos.com/usbaudio16mc

2.5 USB Audio 2.0 DJ Kit

This hardware should not be used for the basis of a new design.

The XMOS USB Audio 2.0 DJ kit (XR-USB-AUDIO-2.0-4C)[14] is a hardware reference design available from XMOS based on the XMOS U8 device.

The DJ naming simply comes from the fact the board has 4 input and 4 output audio channels - a common configuration for a DJ controller.

The kit is made up of two boards a “core” board and an “audio slice” board. Part numbers XP-SKC-SU1 and XA-SK-AUDIO respectively.

The core board includes a U-Series device with integrated USB PHY. The audio slice board is equipped with two stereo audio CODECs giving 4 channels of input and 4 channels of output at sample frequencies up to 192kHz.

In addition to analogue channels the audio slice board also has MIDI input and output connectors and a COAX connector for S/PDIF output.

2.6 USB Audio 2.0 Reference Design Board

This hardware should not be used for the basis of a new design.

The USB Audio 2.0 Reference Design[15] is a stereo hardware reference design available from XMOS based on an XMOS L8 device (previously named L1). The diagram in Figure 5 shows the block layout of the USB Audio 2.0 Reference Design board. The main purpose of the XS1 L-Series device is to provide a USB Audio interface to the USB PHY and route the audio to the audio CODEC and S/PDIF output. Note, although the software supports MIDI, there are no MIDI connectors on the board.

For full hardware details please refer to the USB Audio 2.0 Ref Design XS1-L1 Hardware Manual[16].

The reference board has an associated firmware application that uses the USB Audio 2.0 software reference platform. Details of this application can be found in section §6.1.

[14] https://www.xmos.com/support/boards?product=15404
[15] https://www.xmos.com/support/boards?product=14772
[16] https://www.xmos.com/published/usb-audio-20-ref-design-xs1-l1-hardware-manual

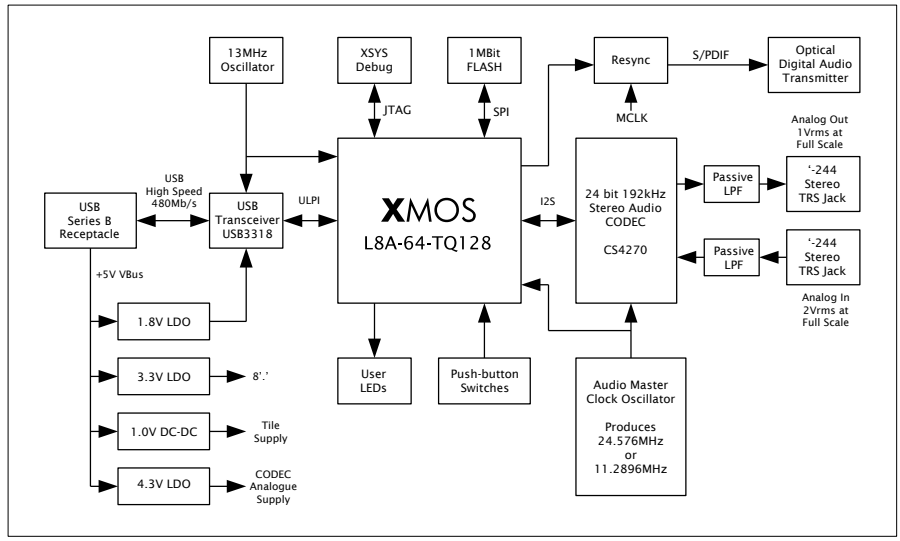

Figure 5: USB Audio 2.0 Reference Design Block Diagram

2.7 USB Audio 2.0 Multichannel Reference Design Board

This hardware should not be used for the basis of a new design.

The USB Audio 2.0 Multichannel Reference Design (XR-USB-AUDIO-2.0-MC)[17] is a hardware reference design available from XMOS based on the XMOS L16 device (previously named L2)

Figure 6 shows the block layout of the USB Audio 2.0 Multichannel Reference Design board.

The board supports six analogue inputs and eight analogue outputs (via a CS4244 CODEC), digital input and output (via coax and optical connectors) and MIDI input and output. For full details please refer to USB Audio 2.0 Reference Design, XS1-L2 Edition Hardware Manual[18].

The reference board has an associated firmware application that uses the USB Audio 2.0 software reference platform. Details of this application can be found in section §6.3.

[17] https://www.xmos.com/support/boards?product=14771
[18] https://www.xmos.com/download/public/USB-Audio-2.0-MC-Hardware-Manual%281.6%29.pdf

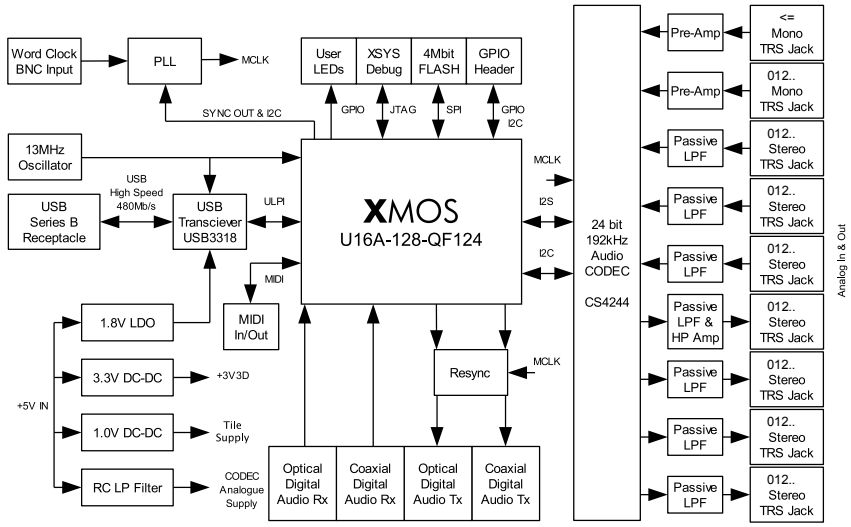

Figure 6: USB Audio 2.0 Multichannel Reference Design Block Diagram

3 Software Architecture

IN THIS CHAPTER

The following sections describe the software architecture of the XMOS USB Audio framework.

XMOS USB Audio solutions are provided as a framework with reference design applications customising and extending this framework to provide the required functionality. These applications execute on a reference hardware platform.

3.1 The USB Audio System Architecture

The XMOS USB Audio platform consists of a series of communicating components. Every system is required to have the shared components listed in Figure 7.

Component	Description
XMOS USB Device Driver (XUD)	Handles the low level USB I/O.
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.
Decoupler	Manages delivery of audio packets between the endpoint buffer component and the audio components. It can also handle volume control processing.
Audio Driver	Handles audio I/O over I2S and manages audio data to/from other digital audio I/O components.

Figure 7: Shared Components

In addition Figure 8 shows components that can be added to a design:

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

Figure 8: Optional Components

Figure 9 shows how the components interact with each other. The green circles represent cores with arrows indicating inter-core communications.

This section will now examine these components in further detail.

3.2 XMOS USB Device (XUD) Library

All low level communication with the USB host is handled by the XMOS USB Device (XUD) library.

The `XUD_Manager()` function runs in its own core and communicates with endpoint cores though a mixture of shared memory and channel communications.

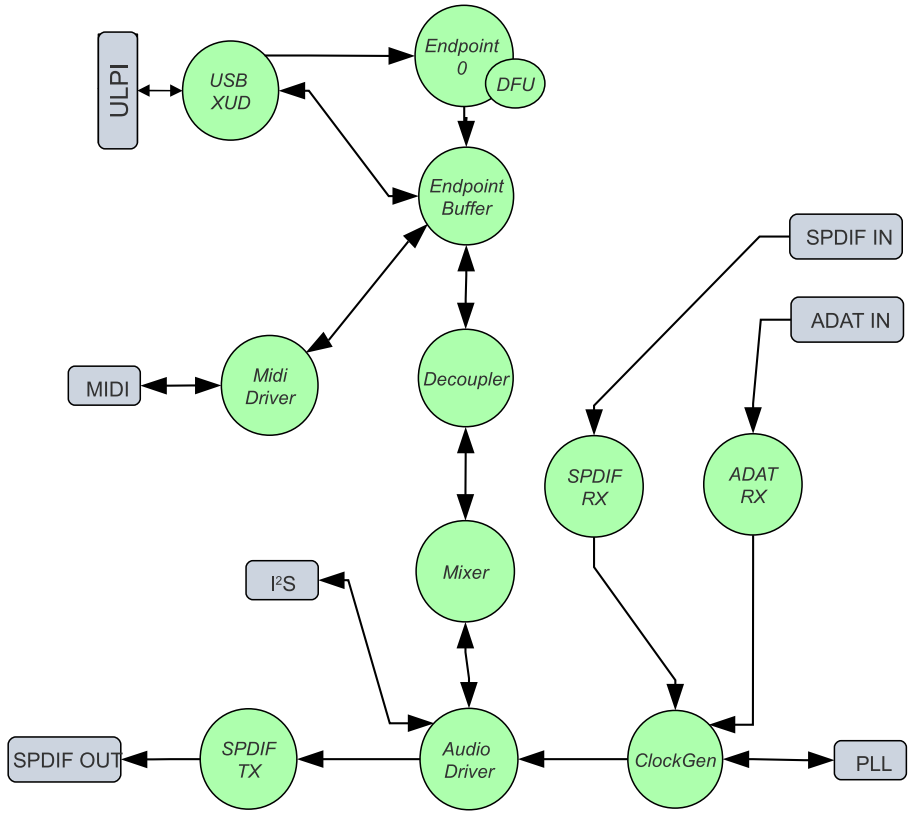

Figure 9: USB Audio Core Diagram

For more details and full XUD API documentation please refer to XMOS USB Device (XUD) Library[19]

Figure 9 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 decoupler core, MIDI data from the MIDI core etc.

[19] http://www.xmos.com/published/xuddg

3.3 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.

3.3.1 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 sc_usb_device (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 the XMOS USB Device Design Guide[20]

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 §7.1).

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.

[20] https://www.xmos.com/zh/node/17007?page=9

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.

3.3.2 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.

3.3.3 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.

3.3.3.1 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.

3.3.3.2 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.

3.3.3.3 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 §3.6.

3.4 Audio Endpoints (Endpoint Buffer and Decoupler)

3.4.1 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.

3.4.2 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`.

3.4.3 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.

3.4.4 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()	
...		

Figure 10: 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)

3.4.4.1 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.

3.4.4.2 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. Figure 11 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

Figure 11: 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.

3.5 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.

Figure 12 shows the signals used to communicate audio between the XMOS device and the CODEC.

Figure 12: 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 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.

Figure 13 shows some example clock frequencies and divides for different sample rates (note that this reflects the single tile L-Series reference board configuration):

Figure 13: Clock Divides used in single tile L-Series Ref Design

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

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.

3.5.1 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.

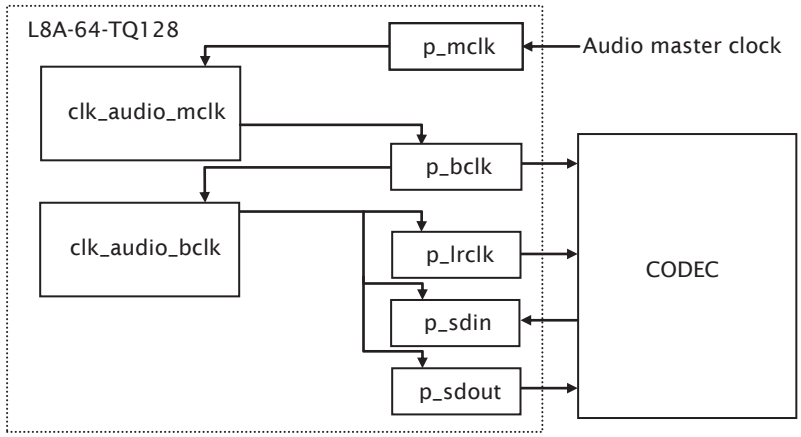

Figure 14: Ports and Clocks (CODEC slave)

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.

`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:

Figure 15: Output patterns

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

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.

3.5.2 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.

3.6 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 §3.3.3.3, 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.

3.6.1 Control

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

Figure 16: Mixer Component 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.

3.6.2 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.

3.7 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).

Figure 17: S/PDIF Capabilities

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

3.7.1 Clocking

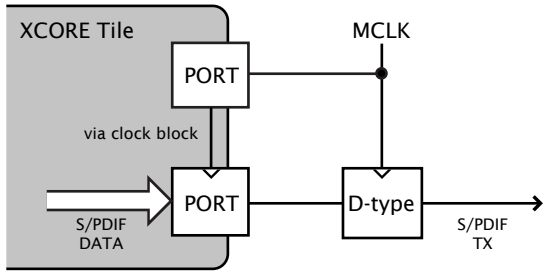

Figure 18: D-Type Jitter Reduction

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).

3.7.2 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

Figure 19: S/PDIF Component Protocol

3.7.3 Output stream structure

The stream is composed of words with the following structure shown in Figure 20. 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 Figure 21.

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

Figure 20: S/PDIF Stream Structure

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

Figure 21: Channel Status Bits

3.8 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:

Figure 22: S/PDIF RX 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:

Figure 23: S/PDIF RX 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 specification for further details on format, user bits etc.

3.8.1 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* (§3.10)

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`):

```
/* Returns 1 for bad parity, else 0 */
static inline int badParity(unsigned x)
{
    unsigned X = (x>>4);
    crc32(X, 0, 1);
    return X & 1;
}
```

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;
    }
}
```

3.9 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 §7.3.

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.

3.9.1 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.

3.10 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.

3.11 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`.

3.12 PDM Microphones

3.13 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:

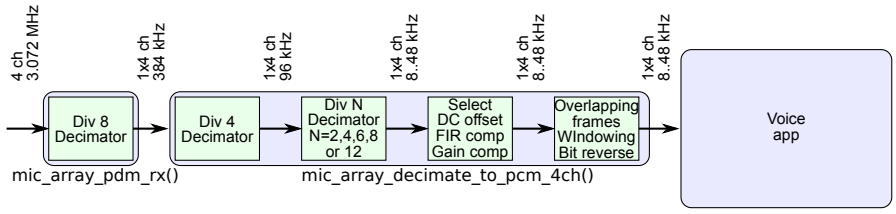

Figure 24: One to four channel count PDM interface

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

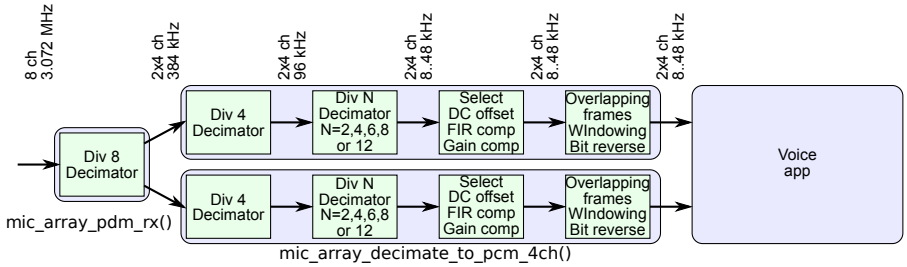

Figure 25: Five to eight count PDM interface

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.

3.13.1 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:

Figure 26: PDM microphone data and signal wires

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.

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.

3.13.2 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.

3.14 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 §3.5
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)	

Figure 27: 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[21] for further details.

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

4 Features & Options

IN THIS CHAPTER

This section looks at some of the available features of the USB Audio design.

4.1 Device Firmware Upgrade (DFU)

The DFU interface handles updates to the boot image of the device. The DFU code is called from the Endpoint 0 core.

The interface links USB to the XMOS flash user library (see XM-000953-PC). In Application mode the DFU can accept commands to reset the device into DFU mode. There are two ways to do this:

- The host can send a `DETACH` request and then reset the device. If the device is reset by the host within a specified timeout, it will start in DFU mode (this is initially set to one second and is configurable from the host).
- The host can send a custom user request `XMOS_DFU_RESETDEVICE` to the DFU interface that resets the device immediately into DFU mode.

Once the device is in DFU mode. The DFU interface can accept commands defined by the DFU 1.1 class specification[22]. In addition the interface accepts the custom command `XMOS_DFU_REVERTFACTORY` which reverts the active boot image to the factory image. Note that the XMOS specific command request identifiers are defined in `dfu_types.h` within `module_dfu`.

4.2 USB Audio Class Version Support

The XMOS USB Audio framework supports both USB Audio Class 1.0 and Audio Class 2.0.

USB Audio Class 2.0 offers many improvements over USB Audio Class 1.0, most notable is the complete support for high-speed operation. This means that Audio

[22] http://www.usb.org/developers/devclass_docs/DFU_1.1.pdf*USB

Class devices are no longer limited to full-speed operation allowing greater channel counts, sample frequencies and sample bit-depths. Additional improvement, amoungst 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

4.2.1 Driver Support

4.2.1.1 Audio Class 1.0

Audio Class 1.0 is fully supported in Apple OSX. Audio Class 1.0 is fully supported in all modern Microsoft Windows operating systems (i.e. Windows XP and later).

4.2.1.2 Audio Class 2.0

Audio Class 2.0 is fully supported in Apple OSX since version 10.6.4. Audio Class 2.0 is not supported natively by Windows operating systems. It is therefore required that a driver is installed. Documentation of Windows drivers is beyond the scope of this document, please contact XMOS for further details.

4.2.2 Audio Class 1.0 Mode and Fall-back

The normal default for XMOS USB Audio applications is to run as a high-speed Audio Class 2.0 device. However, some products may prefer to run in Audio Class 1.0 mode, this is normally to allow “driver-less” operation with Windows operating systems.

To ensure specification compliance, Audio Class 1.0 mode *always* operates at full-speed USB.

The device will operate in full-speed Audio Class 1.0 mode if one of the following is true:

- The code is compiled for USB Audio Class 1.0 only.
- The code is compiled for USB Audio Class 2.0 and it is connected to the host over a full speed link (and the Audio Class fall back is enabled).

The options to control this behavior are detailed in *usb_audio_sec_custom_defines_api*.

When running in Audio Class 1.0 mode the following restrictions are applied:

- MIDI is disabled.
- DFU is disabled (Since Windows operating systems would prompt for a DFU driver to be installed)

Due to bandwidth limitations of full-speed USB the following sample-frequency restrictions are also applied:

- Sample rate is limited to a maximum of 48kHz if both input and output are enabled.
- Sample rate is limited to a maximum of 96kHz if only input *or* output is enabled.

4.3 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 HID interface
2. A Get Report Descriptor request is enabled in `endpoint0`.
3. Endpoint data handling is enabled in the `buffer` core

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 `descriptors.h` should be modified. The default report size is 1 byte with the format as follows:

Figure 28: 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 `Vendor_ReadHidButtons(unsigned char h` is called from `buffer()`. This function is passed an array `hidData[]` by reference. The programmer should report the state of his buttons into this array. For example, if a volume up command is desired, bit 3 should be set to 1, else 0.

Since the `Vendor_ReadHidButtons()` function is called from the `buffer` logical core, care should be taken not to add to much execution time to this function since this could cause issues with servicing other endpoints.

For a full example please see the HID section in §6.1.

4.4 Apple MFi compatibility

XMOS devices are capable of operating with Apple iPod, iPhone, and iPad devices that feature USB host support. Information regarding this functionality is protected by the Made For iPod (MFi) program and associated licensing.

Please contact XMOS for details and further documentation.

4.5 Audio Stream Formats

The design currently supports up to 3 different stream formats for output/playback, selectable at run time. This is implemented using Alternative Settings to the AudioStreaming interfaces.

An AudioStreaming 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 AudioStreaming interface. Whenever an AudioStreaming 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.

For further information refer to 3.16.2 of USB Audio Device Class Definition for Audio Devices[23]

Note, a 0-bandwidth alternative setting 0 is always implmented by the design (as required by the USB specifications).

Customisatble parameters for the Alternate Settings are as follows.:

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

Currently only a single format is supported for the input/recording stream

[23] http://www.usb.org/developers/devclass_docs/Audio2.0_final.zip

4.5.1 Audio Subslot

An audio subslot holds a single audio sample. See USB Device Class Definition for Audio Data Formats[24] for full details. This is represented by *bSubslotSize* in the devices descriptors

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

Typically, since it is run on a 32-bit machine, the value 4 is used for subslot - this means that packing/unpacking samples is trivial. Other values can be used (currently 4, 3 and 2 are supported by the design).

Other values may be used for the 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 many Android based hosts support only 16bit 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*

4.5.2 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* <= *bSubslotSize* * 8. Supported values are 16, 24 and 32.

The following defines

- The following defines affect high-speed operation:
  - *HS_STREAM_FORMAT_OUTPUT_1_RESOLUTION_BITS*
  - *HS_STREAM_FORMAT_OUTPUT_2_RESOLUTION_BITS*
  - *HS_STREAM_FORMAT_OUTPUT_3_RESOLUTION_BITS*

[24] http://www.usb.org/developers/devclass_docs/Audio2.0_final.zip

- The following defines affect full-speed operation:
  - *FS_STREAM_FORMAT_OUTPUT_1_RESOLUTION_BITS*
  - *FS_STREAM_FORMAT_OUTPUT_2_RESOLUTION_BITS*
  - *FS_STREAM_FORMAT_OUTPUT_3_RESOLUTION_BITS*

4.5.3 Audio Format

The design supports two audio formats, PCM and 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[25]). This use of a RAW/DSD format in an alternative setting is termed *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_TYPEI_RAW_DATA
- UAC_FORMAT_TYPEI_PCM

Currently DSD is only supported on the output/playback stream

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.

4.6 DSD over PCM (DoP)

While Native DSD support is available in Windows though a driver, OSX incorporates a USB driver that only supports PCM, this is also true of the central audio engine, CoreAudio. It is therefore not possible to use the scheme defined above using the built in driver support of OSX.

Since the Apple OS only allows a PCM path a method of transporting DSD audio data over PCM frames has been developed. This data can then be sent via the native USB Audio support.

The XMOS USB Audio design(s) implement the method described in DoP Open Standard 1.1[26]

[25] http://www.usb.org/developers/devclass_docs/Audio2.0_final.zip
[26] http://dsd-guide.com/sites/default/files/white-papers/DoP_openStandard_1v1.pdf

Standard DSD has a sample size of 1 bit and a sample rate of 2.8224MHz - this is 64x the speed of 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 core (*audio.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.

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

5 Programming Guide

IN THIS CHAPTER

The following sections provide a guide on how to program the USB audio software platform including instructions for building and running programs and creating your own custom USB audio applications.

5.1 Getting Started

5.1.1 Building and Running

To build, select the relavant project (e.g. `app_usb_aud_l1`) in the Project Explorer and click the **Build** icon.

To install the software, open the xTIMEcomposer Studio and follow these steps:

1. Choose *File* ▶ *Import*.
2. Choose *General* ▶ *Existing Projects into Workspace* and click **Next**.
3. Click **Browse** next to *Select archive file* and select the file firmware ZIP file.
4. Make sure the projects you want to import are ticked in the *Projects* list. Import all the components and whichever applications you are interested in.
5. Click **Finish**.

To build, select the relevant project (e.g. `app_usb_aud_l1`) in the Project Explorer and click the **Build** icon.

From the command line, you can follow these steps:

1. To install, unzip the package zip.
2. To build, change into the relevant application directory (e.g. `app_usb_aud_l1`) and execute the command:

```
xmake all
```

The main Makefile for the project is in the app directory (e.g. `app_usb_aud_l1`). This file specifies build options and used modules. The Makefile uses the common build infrastructure in `module_xmos_common`. This system includes the source files from the relevant modules and is documented within `module_xmos_common`.

5.1.2 Installing the application onto flash

To upgrade the firmware you must, firstly:

1. Plug the USB Audio board into your computer.
2. Connect the xTAG-2 to the USB Audio board and plug the xTAG-2 into your PC or Mac.

To upgrade the flash from xTIMEcomposer Studio, follow these steps:

1. Start xTIMEcomposer Studio and open a workspace.
2. Choose *File* ▶ *Import* ▶ *C/XC* ▶ *C/XC Executable*.
3. Click **Browse** and select the new firmware (XE) file.
4. Click **Next** and **Finish**.
5. A Debug Configurations window is displayed. Click **Close**.
6. Choose *Run* ▶ *Flash Configurations*.
7. Double-click *xCORE application* to create a new Flash configuration.
8. Browse for the XE file in the *Project* and *C/XC Application* boxes.
9. Ensure the *xTAG-2* device appears in the target list.
10. Click **Flash**.

From the command line:

1. Open the XMOS command line tools (Desktop Tools Prompt) and execute the following command:

```
xflash <binary>.xe
```

5.2 Project Structure

5.2.1 Applications and Modules

The code is split into several module directories. The code for these modules can be included by adding the module name to the `USED_MODULES` define in an application Makefile:

module_xud	Low level USB device library
module_usb_shared	Common code for USB applications
module_usb_device	Common code for USB device applications
module_usb_audio	Common code for USB audio applications
module_spdif_tx	S/PDIF transmit code
module_spdif_rx	S/PDIF receive code
module_adat_rx	ADAT receive code
module_usb_midi	MIDI I/O code
module_dfu	Device Firmware Upgrade code

Figure 29: Modules used by USB Audio

There are multiple application directories that contain Makefiles that build into executables:

app_usb_aud_l1	USB Audio 2.0 Reference Design application
app_usb_aud_l2	USB Audio 2.0 Multichannel Reference Design application
app_usb_aud_skc_u16	U16 SliceKit with Audio Slice application
app_usb_aud_xk_u8_2c	Multi-function Audio board application
app_usb_aud_skc_su1	DJ kit application

Figure 30: USB Audio Reference Applications

5.3 Build Configurations

Due to the flexibility of the framework there are many different build options. For example input and output channel count, Audio Class version, interface types etc. A “build configuration” is a set of build options that combine to produce a certain feature set.

Build configurations are listed in the application makefile with their associated options, they can be built within the xTIMEComposer GUI or via the command like as follows:

```
xmake CONFIG=<config name>
```

When a reference design application is compiled using “build all” (*xmake all* on command line) all configurations are automatically built.

A naming scheme is employed in each application to link a feature set to a build configuration/binary. Different variations of the same basic scheme are used. This scheme is described in the next section.

5.4 Validated Build Configurations

It is not possible for all build configuration permutations to be exhaustively tested. XMOS therefore test a subset of build configurations for proper behaviour, these are based on popular device configurations.

The presence of a build configuration in an application signifies it as a Validated Build Configuration and should be considered supported.

5.5 Configuration Naming Scheme

This section describes the naming scheme for the provided build configurations (and therefore binaries) generated for each build configuration

Each relevant build option is assigned a position in the configuration name, with a character denoting the options value (normally 'x' is used to denote "off" or "disabled")

For example, Figure 31 lists the build options for the single tile L-Series Reference Design.

Figure 31: Single tile L-Series build options

Build Option Name	Options	Denoted by
Audio Class Version	1 or 2	1 or 2
Audio Input	on or off	i or x
Audio Output	on or off	o or x
MIDI	on or off	m or x
S/PDIF Output	on or off	s or x

For example a binary named 2ioxs would indicate Audio Class 2.0 with input and output enabled, MIDI disabled, SPDIF output enabled.

5.6 A USB Audio Application

This section provides a walk through of the single tile USB Audio Reference Design (L-Series) example, which can be found in the `app_usb_aud_l1` directory.

In each application directory the `src` directory is arranged into two folders:

#. An `core` directory containing source items that must be made available to the USB Audio framework

1. An `extensions` directory that includes extensions to the framework such as CODEC config etc

The `core` folder for each application contains:

1. A `.xn` file to describe the hardware platform the app will run on
2. A custom defines file: `customdefines.h` for framework configuration

5.6.1 Custom Defines

The `customdefines.h` file contains all the #defines required to tailor the USB audio framework to the particular application at hand. Typically these over-ride default values in *devicedefines.h* in *module_usb_audio*.

First there are defines to determine overall capability. For this appliction S/PDIF output and DFU are enabled. Note that *ifndef* is used to check that the option is not already defined in the makefile.

```
/* Enable/Disable MIDI - Default is MIDI off */
#ifndef MIDI
#define MIDI                (0)
#endif

/* Enable/Disable SPDIF - Default is SPDIF on */
#ifndef SPDIF_TX
#define SPDIF_TX            (1)
#endif
```

Next, the file defines the audio properties of the application. This application has stereo in and stereo out with an S/PDIF output that duplicates analogue channels 1 and 2 (note channels are indexed from 0):

```
/* Number of USB streaming channels - Default is 2 in 2 out */
#ifndef NUM_USB_CHAN_IN
#define NUM_USB_CHAN_IN     (2)         /* Device to Host */
#endif
#ifndef NUM_USB_CHAN_OUT
#define NUM_USB_CHAN_OUT    (2)         /* Host to Device */
#endif

/* Number of IS2 chans to DAC..*/
#ifndef I2S_CHANS_DAC
#define I2S_CHANS_DAC       (2)
#endif

/* Number of I2S chans from ADC */
#ifndef I2S_CHANS_ADC
#define I2S_CHANS_ADC       (2)
#endif

/* Index of SPDIF TX channel (duplicated DAC channels 1/2) */
#define SPDIF_TX_INDEX      (0)
```

The file then sets some defines for the master clocks on the hardware and the maximum sample-rate for the device.

```
/* Master clock defines (in Hz) */
#define MCLK_441            (256*44100)    /* 44.1, 88.2 etc */
#define MCLK_48             (512*48000)    /* 48, 96 etc */

/* Maximum frequency device runs at */
#ifndef MAX_FREQ
#define MAX_FREQ            (192000)
#endif
```

Finally, there are some general USB identification defines to be set. These are set for the XMOS reference design but vary per manufacturer:

```
#define VENDOR_ID           (0x20B1) /* XMOS VID */
#define PID_AUDIO_2         (0x0002) /* L1 USB Audio Reference Design PID */
#define PID_AUDIO_1         (0x0003) /* L1 USB Audio Reference Design PID */
```

For a full description of all the defines that can be set in `customdefines.h` see §7.1

5.6.2 Configuration Functions

In addition to the custom defines file, the application needs to provide implementations of user functions that are specific to the application.

For *app_usb_aud_l1* the implementations can be found in *audiohw.xc*.

Firstly, code is required to initialise the external audio hardware. In the case of the CODEC on the L1 Refence Design board there is no required action so the funciton is left empty:

```
void AudioHwInit(chanend ?c_codec)
{
    return;
}
```

On every sample-rate change a call is made to *AudioHwConfig()*. In the case of the CODEC on the L1 Reference Design baord the CODEC must be reset and set the relevant clock input from the two oscillators on the board.

Both the CODEC reset line and clock selection line are attached to the 32 bit port 32A. This is accessed through the `port32A_peek` and `port32A_out` functions:

```
#define PORT32A_PEEK(X) {asm("peek %0, res[%1]":"=r"(X):"r"(XS1_PORT_32A))
  ↪ ;}
#define PORT32A_OUT(X)  {asm("out res[%0], %1"::"r"(XS1_PORT_32A),"r"(X));}
```

```
/* Configures the CODEC for the required sample frequency.
 * CODEC reset and frequency select are connected to port 32A
 *
 * Port 32A is shared with other functionality (LEDs etc) so we
 * access via inline assembly. We also take care to retain the
 * state of the other bits.
 */
void AudioHwConfig(unsigned samFreq, unsigned mClk, chanend ?c_codec,
  ↪ unsigned dsdMode,
    unsigned samRes_DAC, unsigned samRes_ADC)
{
    timer t;
    unsigned time;
    unsigned tmp;

    /* Put codec in reset and set master clock select appropriately */

    /* Read current port output */
    PORT32A_PEEK(tmp);

    /* Put CODEC reset line low */
    tmp &= (~P32A_COD_RST);

    if ((samFreq % 22050) == 0)
    {
        /* Frequency select low for 441000 etc */
        tmp &= (~P32A_CLK_SEL);
    }
    else //if((samFreq % 24000) == 0)
    {
        /* Frequency select high for 48000 etc */
        tmp |= P32A_CLK_SEL;
    }

    PORT32A_OUT(tmp);

    /* Hold in reset for 2ms */
    t :> time;
    time += 200000;
    t when timerafter(time) :> int _;

    /* Codec out of reset */
    PORT32A_PEEK(tmp);
    tmp |= P32A_COD_RST;
    PORT32A_OUT(tmp);
}
```

Finally, the application has functions for audio streaming start/stop that enable/disable an LED on the board (also on port 32A):

```
#include <xs1.h>
#include "port32A.h"

/* Functions that handle functions that must occur on stream
 * start/stop e.g. DAC mute/un-mute.These need implementing
 * for a specific design.
 *
 * Implementations for the L1 USB Audio Reference Design
 */

/* Any actions required for stream start e.g. DAC un-mute - run every
 * stream start.
 *
 * For L1 USB Audio Reference Design we illuminate LED B (connected
 * to port 32A)
 *
 * Since this port is shared with other functionality inline assembly
 * is used to access the port resource.
 */
void UserAudioStreamStart(void)
{
    int x;

    /* Peek at current port value using port 32A resource ID */
    asm("peek %0, res[%1]":"=r"(x):"r"(XS1_PORT_32A));

    x |= P32A_LED_B;

    /* Output to port */
    asm("out res[%0], %1"::"r"(XS1_PORT_32A),"r"(x));
}

/* Any actions required on stream stop e.g. DAC mute - run every
 * stream stop
 * For L1 USB Audio Reference Design we extinguish LED B (connected
 * to port 32A)
 */
void UserAudioStreamStop(void)
{
    int x;

    asm("peek %0, res[%1]":"=r"(x):"r"(XS1_PORT_32A));
    x &= (~P32A_LED_B);
    asm("out res[%0], %1"::"r"(XS1_PORT_32A),"r"(x));
}
```

5.6.3 The main program

The `main()` function is shared across all applications is therefore part of the framework. It is located in `sc_usb_audio` and contains:

- A declaration of all the ports used in the framework. These vary depending on the PCB an application is running on.
- A `main` function which declares some channels and then has a `par` statement which runs the required cores in parallel.

The framework supports devices with multiple tiles so it uses the `on tile[n]:` syntax.

The first core run is the XUD library:

```
#if (AUDIO_CLASS==2)
XUD_Manager(c_xud_out, ENDPOINT_COUNT_OUT, c_xud_in, ENDPOINT_COUNT_IN,
    c_sof, epTypeTableOut, epTypeTableIn, p_usb_rst,
    clk, 1, XUD_SPEED_HS, XUD_PWR_CFG);
#else
XUD_Manager(c_xud_out, ENDPOINT_COUNT_OUT, c_xud_in, ENDPOINT_COUNT_IN,
    c_sof, epTypeTableOut, epTypeTableIn, p_usb_rst,
    clk, 1, XUD_SPEED_FS, XUD_PWR_CFG);
#endif
```

The make up of the channel arrays connecting to this driver are described in §7.3.

The channels connected to the XUD driver are fed into the buffer and decouple cores:

```
buffer(c_xud_out[ENDPOINT_NUMBER_OUT_AUDIO],    /* Audio Out*/
    c_xud_in[ENDPOINT_NUMBER_IN_AUDIO],         /* Audio In */
#if (NUM_USB_CHAN_IN == 0) || defined(UAC_FORCE_FEEDBACK_EP)
    c_xud_in[ENDPOINT_NUMBER_IN_FEEDBACK],      /* Audio FB */
#endif
#ifdef MIDI
    c_xud_out[ENDPOINT_NUMBER_OUT_MIDI],        /* MIDI Out */ // 2
    c_xud_in[ENDPOINT_NUMBER_IN_MIDI],          /* MIDI In */  // 4
    c_midi,
#endif
#ifdef IAP
    c_xud_out[ENDPOINT_NUMBER_OUT_IAP],   /* iAP Out */
    c_xud_in[ENDPOINT_NUMBER_IN_IAP],     /* iAP In */
#ifdef IAP_INT_EP
    c_xud_in[ENDPOINT_NUMBER_IN_IAP_INT], /* iAP Interrupt In */
#endif
    c_iap,
#ifdef IAP_EA_NATIVE_TRANS
    c_xud_out[ENDPOINT_NUMBER_OUT_IAP_EA_NATIVE_TRANS],
    c_xud_in[ENDPOINT_NUMBER_IN_IAP_EA_NATIVE_TRANS],
    c_EANativeTransport_ctrl,
    c_ea_data,
#endif
#endif
#if defined(SPDIF_RX) || defined(ADAT_RX)
    /* Audio Interrupt - only used for interrupts on external clock change
      ↪ */
    c_xud_in[ENDPOINT_NUMBER_IN_INTERRUPT],
    c_clk_int,
#endif
    c_sof, c_aud_ctl, p_for_mclk_count
#ifdef HID_CONTROLS
    , c_xud_in[ENDPOINT_NUMBER_IN_HID]
#endif
#ifdef CHAN_BUFF_CTRL
    , c_buff_ctrl
#endif
);
```

```
{
    thread_speed();
    decouple(c_mix_out
#ifdef CHAN_BUFF_CTRL
        , c_buff_ctrl
#endif
    );
}
```

These then connect to the audio driver which controls the I2S output and S/PDIF output (if enabled). If S/PDIF output is enabled, this component spawns into two cores as opposed to one.

```
{
    thread_speed();
#ifdef MIXER
#define AUDIO_CHANNEL c_mix_out
#else
#define AUDIO_CHANNEL c_aud_in
#endif
    audio(AUDIO_CHANNEL,
#if defined(SPDIF_TX) && (SPDIF_TX_TILE != AUDIO_IO_TILE)
        c_spdif_tx,
#endif
#if defined(SPDIF_RX) || defined(ADAT_RX)
        c_dig_rx,
#endif
        c_aud_cfg, c_adc
#if XUD_TILE != 0
        , dfuInterface
#endif
#if (NUM_PDM_MICS > 0)
        , c_pdm_pcm
#endif
#ifdef RUN_DSP_TASK
        , i_dsp
#endif
    );
}

#if defined(SPDIF_RX) || defined(ADAT_RX)
{
    thread_speed();

    clockGen(c_spdif_rx, c_adat_rx, p_pll_clk, c_dig_rx, c_clk_ctl,
      ↪ c_clk_int);

}
#endif
```

Finally, if MIDI is enabled you need a core to drive the MIDI input and output. The MIDI core also optionally handles authentication with Apple devices. Due to licensing issues this code is only available to Apple MFI licensees. Please contact XMOS for details.

```
on tile[MIDI_TILE]:
{
    thread_speed();
    usb_midi(p_midi_rx, p_midi_tx, clk_midi, c_midi, 0, null, null, null,
      ↪ null);
}
```

5.7 Adding Custom Code

The flexibility of the USB audio solution means that you can modify the reference applications to change the feature set or add extra functionality. Any part of the software can be altered with the exception of the XUD library.

The reference designs have been verified against a variety of host OS types, across different samples rates. However, modifications to the code may invalidate the results of this verification and you are strongly encouraged to fully re- test the resulting software.

The general steps are:

1. Make a copy of the eclipse project or application directory (.e.g. `app_usb_aud_l1` or `app_usb_aud_l2`) you wish to base your code on, to a separate directory with a different name.

2. Make a copy of any modules you wish to alter (most of the time you probably do not want to do this). Update the Makefile of your new application to use these new custom modules.

3. Make appropriate changes to the code, rebuild and reflash the device for testing.

Once you have made a copy, you need to:

1. Provide a `.xn` file for your board (updating the *TARGET* variable in the Makefile appropriately).

2. Update `device_defines.h` with the specific defines you wish to set.

3. Update `main.xc`.

4. Add any custom code in other files you need.

The following sections show some example changes with a high level overview of how to change the code.

5.7.1 Example: Changing output format

You may wish to customize the digital output format e.g. for a CODEC that expects sample data left justified with respect to the word clock.

To do this you need to alter the main audio driver loop in `audio.xc`. After the alteration you need to re-test the functionality. The XMOS Timing Analyzer can help guarantee that your changes do not break the timing requirement of this core.

5.7.2 Example: Adding DSP to output stream

To add some DSP requires an extra core of computation, so some existing functionality needs to be removed (e.g. S/PDIF). Follow these steps to update the code:

1. Remove some functionality using the defines in §7.1 to free up a core.

2. Add another core to do the DSP. This core will probably have three XC channels: one channel to receive samples from decoupler core and another to output to the audio driver—this way the core 'intercepts' audio data on its way to the audio driver; the third channel can receive control commands from Endpoint 0.

3. Implement the DSP on this core. This needs to be synchronous (i.e. for every sample received from the decoupler, a sample needs to be outputted to the audio driver).

6 USB Audio Applications

IN THIS CHAPTER

In addition to the overall framework, reference design applications are provided. These applications provide qualified configurations of the framework which support and are validated on accompanying hardware. This section looks at how the various applictions customise and extend the framework.

6.1 USB Audio 2.0 Reference Design (L-Series) Application

The USB Audio 2.0 Reference Design is an application of the USB audio framework specifically for the hardware described in §2.6 and is implemented on the L-Series single tile device (500MIPS). The code can be found in *app_usb_aud_l2*

The software design supports two channels of audio at sample frequencies up to 192kHz and uses the following components:

- XMOS USB Device Driver (XUD)
- Endpoint 0
- Endpoint buffer
- Decoupler
- Audio Driver
- Device Firmware Upgrade (DFU)
- S/PDIF Transmitter *or* MIDI

The diagrams Figure 32 and Figure 33 show the software layout of the code running on the XS1-L chip. Each unit runs in a single core concurrently with the others units. The lines show the communication between each functional unit. Due to the MIPS requirement of the USB driver (see §3.14), only six cores can be run on the single tile L-Series device so only one of S/PDIF transmit or MIDI can be supported.

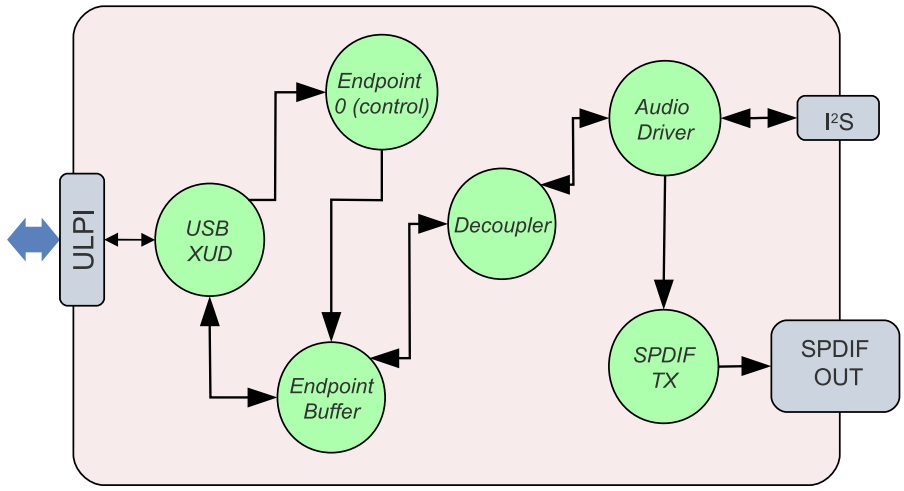

Figure 32: Single Tile L-Series Software Core Diagram (with S/PDIF TX)

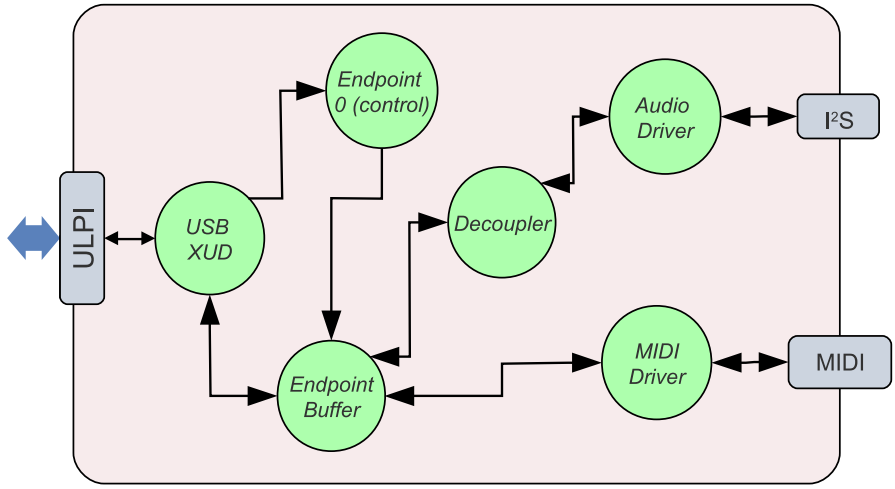

Figure 33: Single Tile L-Series Software Core Diagram (with MIDI I/O)

6.1.1 Port 32A

Port 32A on the XS1-L device is a 32-bit wide port that has several separate signal bit signal connected to it, accessed by multiple cores. To this end, any output to this port must be *read-modify-write* i.e. to change a single bit of the port, the software reads the current value being driven across 32 bits, flips a bit and then outputs the modified value.

This method of port usage (i.e. sharing a port between cores) is outside the standard XC usage model so is implemented using inline assembly as required. The peek instruction is used to get the current output value on the port:

```
/* Peek at current port value using port 32A resource ID */
asm("peek %0, res[%1]":=r"(x):"r"(XS1_PORT_32A));
```

The required output value is then assembled using the relevant bit-wise operation(s) before the out instruction is used directly to output data to the port:

```
/* Output to port */
asm("out res[%0], %1"::"r"(XS1_PORT_32A),"r"(x));
```

The table Figure 34 shows the signals connected to port 32A on the USB Audio Class 2.0 reference design board. Note, they are all *outputs* from the XS1-L device.

Figure 34: Port 32A Signals

Pin	Port	Signal
XD49	P32A0	USB_PHY_RST_N
XD50	P32A1	CODEC_RST_N
XD51	P32A2	MCLK_SEL
XD52	P32A3	LED_A
XD53	P32A4	LED_B

6.1.2 Clocking

The board has two on-board oscillators for master clock generation. These produce 11.2896MHz for sample rates 44.1, 88.2, 176.4KHz etc and 24.567MHz for sample rates 48, 96, 192kHz etc.

The required master clock is selected from one of these using an external mux circuit via port *P32A[2]* (pin 2 of port 32A). Setting *P32A[2]* high selects 11.2896MHz, low selects 24.576MHz.

The reference design board uses a 24 bit, 192kHz stereo audio CODEC (Cirrus Logic CS4270).

The CODEC is configured to operate in *stand-alone mode* meaning that no serial configuration interface is required. The digital audio interface is set to I2S mode with all clocks being inputs (i.e. slave mode).

The CODEC has three internal modes depending on the sampling rate used. These change the oversampling ratio used internally in the CODEC. The three modes are shown below:

In stand-alone mode, the CODEC automatically determines which mode to operate in based on input clock rates.

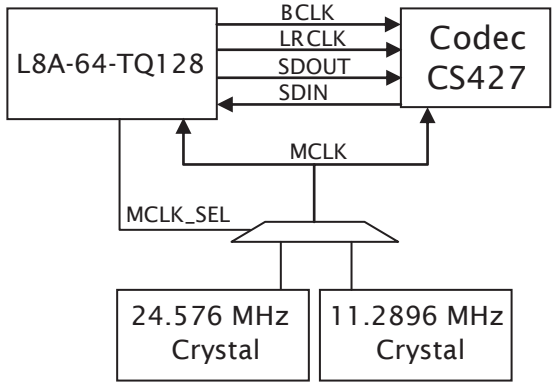

Figure 35: Audio Clock Connections

CODEC mode	CODEC sample rate range (kHz)
Single-Speed	4-54
Double-Speed	50-108
Quad-Speed	100-216

Figure 36: CODEC Modes

The internal master clock dividers are set using the MDIV pins. MDIV is tied low and MDIV2 is connected to bit 2 of port 32A (as well as to the master clock select). With MDIV2 low, the master clock must be 256Fs in single-speed mode, 128Fs in double-speed mode and 64Fs in quad-speed mode. This allows an 11.2896MHz master clock to be used for sample rates of 44.1, 88.2 and 176.4kHz.

With MDIV2 high, the master clock must be 512Fs in single-speed mode, 256Fs in double-speed mode and 128Fs in quad-speed mode. This allows a 24.576MHz master clock to be used for sample rates of 48, 96 and 192kHz.

When changing sample frequency, the `CodecConfig()` function first puts the CODEC into reset by setting *P32A[1]* low. It selects the required master clock/CODEC dividers and keeps the CODEC in reset for 1ms to allow the clocks to stabilize. The CODEC is brought out of reset by setting *P32A[1]* back high.

6.1.3 HID

The reference design implements basic HID controls. The call to `vendor_ReadHidButtons()` simply reads from buttons A and B and returns their state in the relevant bits depending on the desired functionality (play/pause/skip etc). Note the buttons are active low, the HID controls active high. The buttons are therefore read and then inverted.

```
/* Write HID Report Data into hidData array
 *
 * Bits are as follows:
 * 0: Play/Pause
 * 1: Scan Next Track
 * 2: Scan Prev Track
 * 3: Volume Up
 * 4: Volime Down
 * 5: Mute
 */
void UserReadHIDButtons(unsigned char hidData[])
{
#ifndef MIDI
    unsigned a, b;

    p_but_a :> a;
    p_but_b :> b;

    a = (~a) & 1;
    b = (~b) & 1;

    /* Assign buttons A and B to Vol Up/Down */
    hidData[0] = (a << HID_CONTROL_VOLUP_SHIFT) | (b <<
      ↪ HID_CONTROL_VOLDN_SHIFT);
#endif
}
```

In the example above the buttons are assigned to volume up/down.

6.1.4 Validated Build Options

The reference design can be built in several ways by changing the build options. These are described in *usb_audio_sec_custom_defines_api*.

The design has only been fully validated against the build options as set in the application as distributed. See §5.3 for details and binary naming.

In practise, due to the similarities between the U-Series and L-Series feature set, it is fully expected that all listed U-Series configurations will operate as expected on the L-Series and vice versa.

6.1.4.1 Configuration 2ioxs

This configuration runs in high-speed Audio Class 2.0 mode, has the mixer disabled, supports 2 channels in, 2 channels out and supports sample rates up to 192kHz and S/PDIF transmit.

6.1.4.2 Configuration 2iomx

This configuration disables S/PDIF and enables MIDI.

This configuration can be achieved by in the Makefile by defining SPDIF_TX as zero:

```
-DSPDIF_TX=0
```

and MIDI as 1:

```
-DMIDI=1
```

6.1.4.3 Configuration 1ioxs

This configuration is similar to the first configuration apart from it runs in Audio 1.0 over full-speed USB.

This is achieved in the Makefile by:

```
-DAUDIO_CLASS=1
```

6.2 The USB Audio 2.0 DJ Kit (U-Series)

The USB Audio 2.0 Reference Design is an application of the USB audio framework specifically for the hardware described in §2.5 and is implemented on the U-Series single tile device (500MIPS). The software design supports four channels of audio at sample frequencies up to 192kHz and uses the following components:

- XMOS USB Device Driver (XUD)
- Endpoint 0
- Endpoint buffer
- Decoupler
- Audio Driver
- Device Firmware Upgrade (DFU)
- S/PDIF Transmitter *or* MIDI

The software layout is the identical to the single tile L-Series Reference Design and therefore the diagrams Figure 32 and Figure 33 show the software layout of the code running on the XS1-U chip.

As with the L-Series, each unit runs in a single core concurrently with the others units.

Due to the MIPS requirement of the USB driver (see §3.14), only six cores can be run on the single tile L-Series device so only one of S/PDIF transmit or MIDI can be supported.

6.2.1 Clocking and Clock Selection

The actual hardware involved in the clock generation is somewhat different to the single tile L-Series board. Instead of two separate oscillators and switching logic a single oscillator with a Phaselink PLL is used to generate fixed 24.576MHz and 22.5792MHz master-clocks.

This makes no change for the selection of master-clock in terms of software interaction: A single pin is (bit 1 of port 4C) is still used to select between the two master-clock frequencies.

The advantages of this system are fewer components and a smaller board area.

When changing sample frequency, the `CodecConfig()` function first puts the CODEC into reset by setting *P4C[2]* low. It selects the required master clock and keeps the CODEC in reset for 1ms to allow the clocks to stabilize. The CODEC is brought out of reset by setting *P4C[2]* back high.

6.2.2 CODEC Configuration

The board is equipped with two stereo audio CODECs (Cirrus Logic CS4270) giving 4 channels of input and 4 channels of output. Configuration of these CODECs takes place using I2C, with both sharing the same I2C bus. The design uses the open source I2C component sc_i2c[27]

6.2.3 U-Series ADC

The codebase includes code exampling how the ADC built into the U-Series device can be used. Once setup a pin is used to cause the ADC to sample, this sample is then sent via a channel to the xCORE device.

On the DJ kit the ADC is clocked via the same pin as the I2S LR clock. Since this means that a ADC sample is received every audio sample the ADC is setup and it's data received in the audio driver core (`audio.xc`).

The code simply writes the ADC value to the global variable `g_adcVal` for use elsewhere in the program as required. The ADC code is enabled by defining `SU1_ADC_ENABLE` as `1`.

6.2.4 HID Example

The codebase includes an example of a HID volume control implementation based on ADC data. This code should be considered an example only since an absolute ADC input does not serve as an ideal input to a relative HID volume control. Buttons (such as that on the single tile L-Series board) or a Rotary Encoder would be a more fitting choice of input component.

This code is enabled if `HID_CONTROLS`, `SU1_ADC_ENABLE` and `ADC_VOL_CONTROL` are all defined as `1`.

[27] http://www.github.com/xcore/sc_i2c

The `Vendor_ReadHIDButtons()` function simply looks at the value from the ADC, if is near the maximum value it reports a volume up, near the minimum value a volume down is reported. If the ADC value is mid-range no event is reported. The code is shown below:

```
void Vendor_ReadHIDButtons(unsigned char hidData[])
{
    unsigned adcVal;
    int diff;

    hidData[0] = 0;

#if defined(ADC_VOL_CONTROL) && (ADC_VOL_CONTROL == 1)
    adcVal = g_adcVal >> 20;

    if(adcVal < (ADC_MIN + THRESH))
    {
        /* Volume down */
        hidData[0] = 0x10;
    }
    else if (adcVal > (ADC_MAX - THRESH))
    {
        /* Volume up */
        hidData[0] = 0x08;
    }
```

6.2.5 Validated Build Options

The reference design can be built in several ways by changing the build options. These are described in *usb_audio_sec_custom_defines_api*.

The design has only been fully validated against the build options as set in the application as distributed. See §5.3 for details and binary naming scheme.

These fully validated build configurations are listed below. In practise, due to the similarities between the U-Series and L-Series feature set, it is fully expected that all listed U-Series configurations will operate as expected on the L-Series and vice versa.

6.2.5.1 Configuration 2ioxs

This configuration runs in high-speed Audio Class 2.0 mode, has the mixer disabled, supports 2 channels in, 2 channels out, supports sample rates up to 192kHz and S/PDIF transmit.

6.2.5.2 Configuration 2iomx

This configuration disables S/PDIF and enables MIDI.

This configuration can be achieved by in the Makefile by defining SPDIF_TX as zero:

```
-DSPDIF_TX=0
```

and MIDI as 1:

```
-DMIDI=1
```

6.2.5.3 Configuration 2ixxx

This configuration is input only (NUM_USB_CHAN_OUT set to zero). I.e. a microphone application or similar.

6.2.5.4 Configuration 1ioxs

This configuration is similar to the first configuration apart from it runs in Audio 1.0 over full-speed USB.

This is achieved in the Makefile by:

```
-DAUDIO_CLASS=1
```

6.2.5.5 Configuration 1xoxs

This configuration is similar to the configuration above in that it runs in Audio 1.0 over full-speed USB. However, the it is output only (i.e. the input path is disabled with -DNUM_USB_CHAN_IN=0

6.3 The USB Audio 2.0 Multichannel Reference Design (L-Series) Software

The USB Audio 2.0 Multichannel Reference Design is an application of the USB audio framework specifically for the hardware described in §2.6 and is implemented on an L-Series dual tile device (1000MIPS). The software design supports up to 16 channels of audio in and 10 channels of audio out and supports sample frequencies up to 192 kHz and uses the following components:

- XMOS USB Device Driver (XUD)
- Endpoint 0
- Endpoint buffer
- Decoupler

- Audio Driver
- Device Firmware Upgrade (DFU)
- Mixer
- S/PDIF Transmitter
- S/PDIF Receiver
- ADAT Receiver
- Clockgen
- MIDI

Figure 37 shows the software layout of the USB Audio 2.0 Multichannel Reference Design.

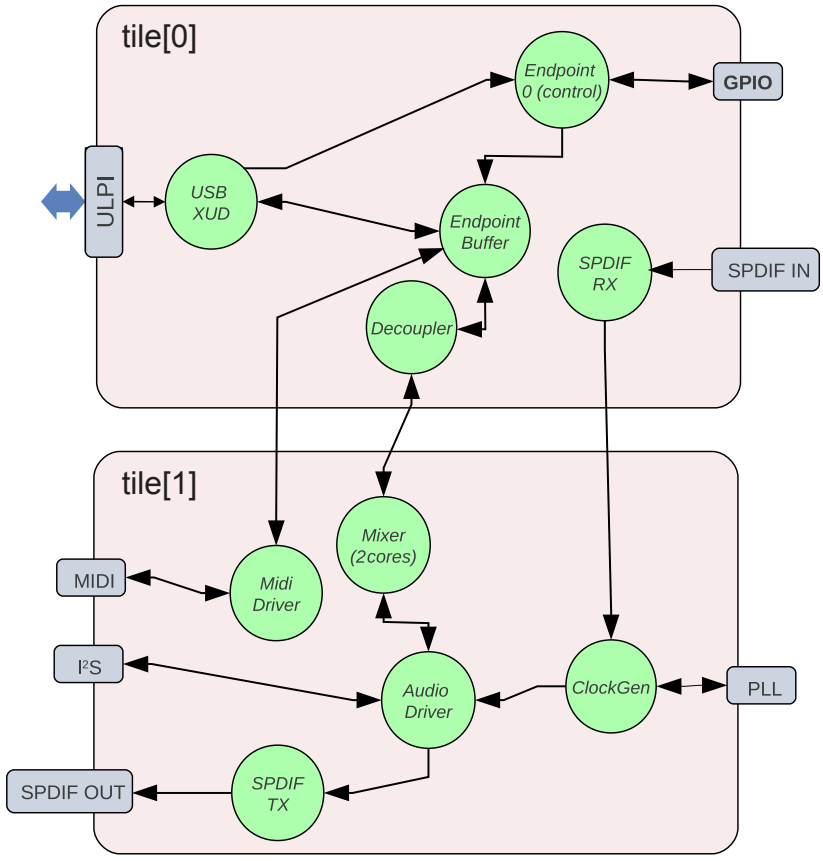

Figure 37: Dual Tile L-Series Reference Design Core Layout

6.3.1 Clocking

For complete clocking flexibility the dual tile L-Series reference design drives a reference clock to an external fractional-n clock multiplier IC (Cirrus Logic CS2300). This in turn generates the master clock used over the design. This is described in §3.10.

6.3.2 Validated Build Options

The reference design can be built in several ways by changing the option described in *usb_audio_sec_custom_defines_api*. However, the design has only been validated against the build options as set in the application as distributed with the following four variations.

6.3.2.1 Configuration 1

All the #defines are set as per the distributed application. It has the mixer enabled, supports 16 channels in, 10 channels out and supports sample rates up to 96kHz.

6.3.2.2 Configuration 2

The same as Configuration 1 but with the CODEC set as I2S master (and the XCORE Tile as slave).

This configuration can be achieved by commenting out the following line in `customdefines.h`:

```
//#define CODEC_SLAVE        1
```

6.3.2.3 Configuration 3

This configuration supports sample rates up to 192kHz but only supports 10 channels in and out. It also disables ADAT receive and the mixer. It can be achieved by commenting out the following lines in `customdefines.h`:

```
//#define MIXER
//#define ADAT_RX            1
```

and changing the following defines to:

```
#define NUM_USB_CHAN_IN  (10)
#define I2S_CHANS_ADC    (6)
#define SPDIF_RX_INDEX   (8)
```

6.3.2.4 Configuration 4

The same as Configuration 3 but with the CODEC set as I2S master. This configuration can be made by making the changes for Configuration 3 and commenting out the following line in `customdefines.h`:

```
//#define CODEC_SLAVE        1
```

6.4 The Multi-function Audio Kit (U-Series)

Provided is an application of the USB audio framework specifically for the hardware described in §2.3 and is implemented on the U-Series single tile device (500MIPS). The application assumes a standard USB B socket (i.e. USB device) is attached as the USB connectivity method.

The software design supports 2 channels channels of audio at sample frequencies up to 192kHz and uses the following components:

- XMOS USB Device Driver (XUD)
- Endpoint 0
- Endpoint buffer
- Decoupler
- Audio Driver
- Device Firmware Upgrade (DFU)
- S/PDIF Transmitter *or* MIDI

The software layout is the identical to the single tile L-Series Reference Design and therefore the diagrams Figure 32 and Figure 33 show the software layout of the code running on the XS1-U chip.

As with the L-Series, each unit runs in a single core concurrently with the others units. The lines show the communication between each functional unit.

Due to the MIPS requirement of the USB driver (see §3.14), only six cores can be run on the single tile L-Series device so only one of S/PDIF transmit or MIDI can be supported.

6.4.1 Clocking and Clock Selection

A single oscillator with a Phaselink PLL is used to generate fixed 24.576MHz and 22.5792MHz master-clocks.

This makes no change for the selection of master-clock in terms of software interaction: A single pin is (bit 1 of port 32A) is used to select between the two master-clock frequencies.

When changing sample frequency, the AudioHwConfig() function first puts the both the DAC and ADC into reset by setting *P4C[0]* and *P4C[1]* low. It selects the required master clock and keeps both the DAC and ADC in reset for 1ms to allow the clocks to stabilize. The DAC and ADC are brought out of reset by setting *P4C[0]* and *P4C[1]* back high.

6.4.2 DAC and ADC Configuration

The board is equipped with a single stereo audio DAC (Cirrus Logic CS4392) and a single stereo ADC (Cirrus Logic 5340) giving 2 channels of input and 2 channels of output.

Configuration of the DAC takes place using I2C. The design uses the open source I2C component sc_i2c[28] No configuration of the ADC is required in software, it is set into slave mode via its configuration pins on the board.

6.4.3 U-Series ADC

The codebase includes code exampling how the ADC built into the U-Series device can be used. Once setup a pin is used to cause the ADC to sample, this sample is then sent via a channel to the xCORE device.

On the multi-function audio board the ADC is clocked via the same pin as the I2S LR clock. Since this means that a ADC sample is received every audio sample the ADC is setup and it's data received in the audio driver core (`audio.xc`).

The ADC inputs for the U8 device are simply pinned out to test point headers. As such there is no example functionality attached to the ADC data.

6.4.4 HID Example

The codebase includes an example of a HID controls implementation using the two buttons and switch on the multi-function audio board.

This example code is enabled if `HID_CONTROLS` are all defined as `1`. When this define is enabled a call to the function `Vendor_ReadHIDButtons()` is enabled and must be implemented. Failing to do so will result in a build error.

The example `Vendor_ReadHIDButtons()` firstly reads the state of the buttons and switch. These inputs are all connected to the same 4-bit port. Since the buttons are active low and the HID report is active high the value read is inverted. Some bitwise operations are then used to exact the individual states of the buttons and switch.

If the switch input is low (i.e. high when inverted) then the button states are shifted up into the position required perform volume up and down and written into the *hidData[]* array:

[28] http://www.github.com/xcore/sc_i2c

```
hidData[0] = (a << HID_CONTROL_VOLUP_SHIFT) | (b << HID_CONTROL_VOLDN_SHIFT
  ↪ );
```

If the switch input is high (i.e. low when inverted) then the buttons states are used to either indicate play/pause or next/previous. Based on counter and a small state-machine a single click on either button provides a play/pause command. A double tap on button A or B provides a previous or next command respectively.

The full code listing is shown below:

```
void UserReadHIDButtons(unsigned char hidData[])
{
    /* Variables for buttons a & b and switch sw */
    unsigned a, b, sw, tmp;

    p_sw :> tmp;

    /* Buttons are active low */
    tmp = ~tmp;

    a = (tmp & (P_GPI_BUTA_MASK))>>P_GPI_BUTA_SHIFT;
    b = (tmp & (P_GPI_BUTB_MASK))>>P_GPI_BUTB_SHIFT;
    sw = (tmp & (P_GPI_SW1_MASK))>>P_GPI_SW1_SHIFT;

    if(sw)
    {
        /* Assign buttons A and B to Vol Up/Down */
        hidData[0] = (a << HID_CONTROL_VOLUP_SHIFT) | (b <<
          ↪ HID_CONTROL_VOLDN_SHIFT);
    }
    else
    {
        /* Assign buttons A and B to play for single tap, next/prev for
          ↪ double tap */
        if(b)
        {
            multicontrol_count++;
                wait_counter = 0;
            lastA = 0;
        }
        else if(a)
        {
            multicontrol_count++;
                wait_counter = 0;
            lastA = 1;
        }
        else
        {
            if(multicontrol_count > THRESH)
            {
                state++;
            }

            wait_counter++;

            if(wait_counter > MULTIPRESS_WAIT)
            {
                    if(state == STATE_PLAY)
                {
                            hidData[0] = (1 << HID_CONTROL_PLAYPAUSE_SHIFT)
                              ↪ ;
                    }
                    else if(state == STATE_NEXTPREV)
                {
                    if(lastA)
                                hidData[0] = (1 << HID_CONTROL_PREV_SHIFT);
                        else
                                hidData[0] = (1 << HID_CONTROL_NEXT_SHIFT);
                }
                    state = STATE_IDLE;
                }
                multicontrol_count = 0;
            }
    }
}
```

6.4.5 Validated Build Options

The reference design can be built in several ways by changing the build options. These are described in §7.1.

The design has only been fully validated against the build options as set in the application as distributed. See §5.3 for details and binary naming scheme.

These fully validated build configurations are listed below. In practise, due to the similarities between the U-Series and L-Series feature set, it is fully expected that all listed U-Series configurations will operate as expected on the L-Series and vice versa.

6.4.5.1 Configuration 2ioxs

This configuration runs in high-speed Audio Class 2.0 mode, has the mixer disabled, supports 2 channels in, 2 channels out, supports sample rates up to 192kHz and S/PDIF transmit.

6.4.5.2 Configuration 2iomx

This configuration disables S/PDIF and enables MIDI.

This configuration can be achieved by in the Makefile by defining `SPDIF_TX` as zero:

```
-DSPDIF_TX=0
```

and `MIDI` as 1:

```
-DMIDI=1
```

6.4.5.3 Configuration 2ixxx

This configuration is input only (`NUM_USB_CHAN_OUT` set to zero). I.e. a microphone application or similar.

6.4.5.4 Configuration 1ioxs

This configuration is similar to the first configuration apart from it runs in Audio 1.0 over full-speed USB.

This is achieved in the Makefile by:

```
-DAUDIO_CLASS=1
```

6.4.5.5 Configuration 1xoxs

This configuration is similar to the configuration above in that it runs in Audio 1.0 over full-speed USB. However, the it is output only (i.e. the input path is disabled with `-DNUM_USB_CHAN_IN=0`

6.5 The U-Series Multi-Channel USB Audio Kit

An application of the USB audio framework is provideed specifically for the hardware described in §2.4 and is implemented on the U-Series dual tile device (1000MIPS). The application assumes a standard USB B socket (i.e. USB device) is provided as the USB connectivity method. The related code can be found in *app_usb_aud_u16_audio8*.

The design supports 10 channels channels of audio input and output at sample frequencies up to 192kHz and uses the following components:

- XMOS USB Device Driver (XUD)
- Endpoint 0
- Endpoint buffer
- Decoupler
- Audio Driver
- Device Firmware Upgrade (DFU)
- S/PDIF Transmitter
- MIDI

The software layout is the identical to the single tile L-Series Multi-channel Reference Design and therefore the diagram Figure 37 shows the software arrangement of the code running on the XS1-U chip.

As with the L-Series, each unit runs in a single core concurrently with the others units. The lines show the communication between each functional unit.

6.5.1 Clocking and Clock Selection

The XA-SK-AUDIO8 double-slot slice includes two options for master clock generation:

- A single oscillator with a Phaselink PLL to generate fixed 24.576MHz and 22.5792MHz master-clocks
- A Cirrus Logic CS2100 clock multiplier allowing the master clock to be generated from a XCore derived reference.

The master clock source is controlled by a mux which, in turn, is controlled by bit 1 of *PORT 4D*:

Figure 38: Master Clock Source Selection

Value	Source
0	Master clock is sourced from PhaseLink PLL
1	Master clock is source from Cirrus Clock Multiplier

The current version of the supplied application only supports the use of the fixed master-clocks from the PhaseLink part.

The clock-select from the phaselink part is controlled via bit 2 of *PORT 4E*:

Figure 39: Master Clock Frequency Select

Value	Frequency
0	24.576MHz
1	22.579MHz

6.5.2 DAC and ADC Configuration

The board is equipped with a single multi-channel audio DAC (Cirrus Logic CS4384) and a single multi-channel ADC (Cirrus Logic CS5368) giving 8 channels of analogue output and 8 channels of analogue input.

Configuration of both the DAC and ADC takes place using I2C. The design uses the I2C component sc_i2c[29].

The reset lines of the DAC and ADC are connected to bits 0 and 1 of *PORT 4E* respectively.

6.5.3 AudioHwInit()

The AudioHwInit() function is implemented to perform the following:

- Initialise the I2C master software module
- Puts the audio hardware into reset
- Enables the power to the audio hardware
- Select the PhaseLink PLL as the audio master clock source.

6.5.4 AudioHwConfig()

The AudioHwConfig() function is called on every sample frequency change.

[29] http://www.github.com/xcore/sc_i2c

The AudioHwConfig() function first puts the both the DAC and ADC into reset by setting *P4E[0]* and *P4E[1]* low. It then selects the required master clock and keeps both the DAC and ADC in reset for a period in order allow the clocks to stabilize.

The DAC and ADC are brought out of reset by setting *P4E[0]* and *P4E[1]* back high.

Various registers are then written to the ADC and DAC as required.

6.5.5 Validated Build Options

The reference design can be built in several ways by changing the build options. These are described in §7.1.

The design has only been fully validated against the build options as set in the application as distributed. See §5.3 for details and binary naming scheme.

These fully validated build configurations are listed below. In practise, due to the similarities between the U-Series and L-Series feature set, it is fully expected that all listed U-Series configurations will operate as expected on the L-Series and vice versa.

6.5.5.1 Configuration 2ioxs

This configuration runs in high-speed Audio Class 2.0 mode, has the mixer core is enabled (for volume processing only, supports 10 channels in, 10 channels out, supports sample rates up to 192kHz and S/PDIF transmit.

6.6 The xCORE-200 Multi-Channel Audio Board

An application of the USB audio framework is provided specifically for the hardware described in §2.1 and is implemented on an xCORE-200-series dual tile device. The related code can be found in *app_usb_aud_xk_216_mc*.

The design supports upto 10 channels of analogue audio input/output at sample-rates up to 192kHz assuming the use of I2S. This can be further increased by utilising TDM.

The design uses the following components:

- XMOS USB Device Driver (XUD)
- Endpoint 0
- Endpoint buffer
- Decoupler
- Audio Driver
- Device Firmware Upgrade (DFU)

- S/PDIF Transmitter
- MIDI

The software layout is the identical to the dual tile L-Series Multi-channel Reference Design and therefore the diagram Figure 37 shows the software arrangement of the code running on the xCORE-200 device.

As with the L/U-Series, each unit runs in a single core concurrently with the others units. The lines show the communication between each functional unit.

6.6.1 Clocking and Clock Selection

The board includes two options for master clock generation:

- A single oscillator with a Phaselink PLL to generate fixed 24.576MHz and 22.5792MHz master-clocks
- A Cirrus Logic CS2100 clock multiplier allowing the master clock to be generated from a XCore derived reference.

The master clock source is controlled by a mux which, in turn, is controlled by bit 5 of *PORT 8C*:

Figure 40: Master Clock Source Selection

Value	Source
0	Master clock is sourced from PhaseLink PLL
1	Master clock is source from Cirrus Clock Multiplier

The clock-select from the phaselink part is controlled via bit 7 of *PORT 8C*:

Figure 41: Master Clock Frequency Select

Value	Frequency
0	24.576MHz
1	22.579MHz

6.6.2 DAC and ADC Configuration

The board is equipped with a single multi-channel audio DAC (Cirrus Logic CS4384) and a single multi-channel ADC (Cirrus Logic CS5368) giving 8 channels of analogue output and 8 channels of analogue input.

Configuration of both the DAC and ADC takes place using I2C. The design uses the I2C component sc_i2c[30].

The reset lines of the DAC and ADC are connected to bits 1 and 6 of *PORT 8C* respectively.

[30] http://www.github.com/xcore/sc_i2c

6.6.3 AudioHwInit()

The AudioHwInit() function is implemented to perform the following:

- Initialise the I2C master software module
- Puts the audio hardware into reset
- Enables the power to the audio hardware
- Select the PhaseLink PLL as the audio master clock source.

6.6.4 AudioHwConfig()

The AudioHwConfig() function is called on every sample frequency change.

The AudioHwConfig() function first puts the both the DAC and ADC into reset by setting *P8C[1]* and *P8C[6]* low. It then selects the required master clock and keeps both the DAC and ADC in reset for a period in order allow the clocks to stabilize.

The DAC and ADC are brought out of reset by setting *P8C[1]* and *P8C[6]* back high.

Various registers are then written to the ADC and DAC as required.

6.6.5 Validated Build Options

The reference design can be built in several ways by changing the build options. These are described in §7.1.

The design has only been fully validated against the build options as set in the application as distributed in the Makefile. See §5.3 for details and binary naming scheme.

These fully validated build configurations are enumerated in the supplied Makefile

In practise, due to the similarities between the U/L/xCORE-200 Series feature set, it is fully expected that all listed U-Series configurations will operate as expected on the L-Series and vice versa.

The build configuration naming scheme employed in the makefile is as follows:

Figure 42: Build config naming scheme

Feature	Option 1	Option 2
Audio Class	1	2
Input	enabled: i (channel count)	disabled: x
Output	enabled: i (channel count)	disabled: x
MIDI	enabled: m	disabled: x
SPDIF output	enabled: s	disabled: x
SPDIF input	enabled: s	disabled: x
ADAT output	enabled: a	disabled: x
ADAT input	enabled: a	disabled: x
DSD output	enabled: d	disabled: x

e.g. A build config named 2i10o10xsxxx would signify: Audio class 2.0, input and output enabled (10 channels each), no MIDI SPDIF output, no SPDIF input, no ADAT or DSD

In addition to this the terms *tdm* or *slave* may be appended to the build configuration name to indicate the I2S mode employed.

6.7 The xCORE-200 Array Microphone Board

An application of the USB audio framework is provided specifically for the hardware described in §2.2 and is implemented on an xCORE-200-series dual tile device. The related code can be found in *app_usb_aud_array_mic*.

The design supports upto 2 channels of analogue audio output at sample-rates from the on-board DAC. The design also supports input from the 7 PDM microphones.

The design uses the following components:

- XMOS USB Device Driver (XUD)
- Endpoint 0

- Endpoint buffer
- Decoupler
- Audio Driver
- Device Firmware Upgrade (DFU)
- PDM Microphone integration

The software layout is the identical to the dual tile L-Series Multi-channel Reference Design and therefore the diagram Figure 37 shows the software arrangement of the code running on the xCORE-200 device.

As with the L/U-Series, each unit runs in a single core concurrently with the others units. The lines show the communication between each functional unit.

The provided application also includes an example of basic microphone data processing.

6.7.1 Clocking and Clock Selection

The board includes an external fractional-N clock multiplier (Cirrus Logic CS2100) for audio clock generation.

This allows the audio master clock to be generated from an reference clock provided by the xCORE, optionally derived from some external source e.g. an incoming digital steam.

This functionality is primarily included on the board to allow for Ethernet AVB, where syncing to an external clock is required. In the USB audio design the IC is simply used for static master clock generation.

The system wide audio master-clock is connected to the AUX output of the CS2100 part. By default, without configuration, the CS2100 part outputs the 24.576MHz REF input to this output.

The clock multiply ratio is programmed into the CS2100 via the I2C bus.

By default a core is used to drive a fixed reference to the CS2100 part using a timer and port I/O. Since this I/O is located on a 4-bit port it cannot be directly output from a clock-block (which would save a core).

In order to reduce core count the following could be done:

- Move the I/O to a 1-bit port and drive the clock directly from a clock-block
- Combine this (computationally simple) task into another task
- Use a clocking methodology that does not require this REF signal as previously explained, it is unlikely the clocking methodology would be employed in a production environment if locking to an external clock is not required.

6.7.2 DAC Configuration

The board is equipped with a single stereo audio DAC with integrated headphone amplifier (Cirrus Logic CS43L21)

Configuration of both the DAC takes place using I2C. The design uses the I2C component sc_i2c[31].

The reset lines of the DAC is connected to bits 0 *PORT 4F*.

6.7.3 AudioHwInit()

The AudioHwInit() function is called on power up and is implemented to perform the following:

- Puts the DAC into reset
- Initialise the I2C master software module
- Initialises the CS2100 part over I2C
- Configures the CS2100 part to output a ratio for a suitable initial master clock frequency (*DEFAULT_MCLK_FREQ*)

6.7.4 AudioHwConfig()

The AudioHwConfig() function is called on every sample frequency change.

The AudioHwConfig() function first puts the both the DAC/headphone-amp and into reset by writing to *PORT 4F*.

It then sets the required ratio in the CS2100 via I2C based on the `mClk` parameter. After a delay, in order to allow the master clock from the CS2100 to settle the DAC is take out of reset. The DAC is then configured via I2C, this primarily involves switching the DAC into I2S slave mode

6.7.5 Mic Processing Example

The provided application includes a basic example of processing the data from the PDM microphones. This basic processing example is located in `user_pdm_process()`. It takes a block of PCM microphone samples as an input and writes processed samples into the `output` array parameter.

The processing involves applying a simple gain globally to all of the microphones. Normal operation sees this gain applied to the data from the 7 microphones and then written to `output[0-6]`. The gain is increased and decreased by pressing buttons B and C respectively.

[31] http://www.github.com/xcore/sc_i2c

The example also provides a simple summing example, where all 7 microphones are summed into `output[0]` with the original microphone signals output to `output[1..7]`. This functionality is enabled and disabled using Button A.

6.7.6 Validated Build Options

The reference design can be built in several ways by changing the build options. These are described in §7.1.

The design has only been fully validated against the build options as set in the application as distributed in the Makefile. See §5.3 for details and binary naming schemes.

These fully validated build configurations are enumerated in the supplied Makefile

In practise, due to the similarities between the U/L/xCORE-200 Series feature set, it is fully expected that all listed U-Series configurations will operate as expected on the L-Series and vice versa.

The build configuration naming scheme employed in the makefile is as follows:

Figure 43: Build config naming scheme

Feature	Option 1	Option 2
Audio Class	1	2
Input	enabled: i (channel count)	disabled: x
Output	enabled: i (channel count)	disabled: x

e.g. A build config named 2i8o2 would signify: Audio class 2.0, input and output enabled (8 in, 2 out).

7 API

IN THIS CHAPTER

7.1 Configuration Defines

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

These defines should be over-ridden in the mandatory `customdefines.h` file or in `Makefile` for a relevant build configuration.

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

7.1.1 Code location (tile)

Macro	**AUDIO_IO_TILE**
Description	Location (tile) of audio I/O. Default: 0

Macro	**XUD_TILE**
Description	Location (tile) of audio I/O. Default: 0

Macro	**IAP_TILE**
Description	Location (tile) of IAP. Default: AUDIO_IO_TILE

Macro	MIDI_TILE
Description	Location (tile) of MIDI I/O. Default: AUDIO_IO_TILE

7.1.2 Channel Counts

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

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

Macro	DSD_CHANS_DAC
Description	Number of DSD output channels. Default: 0 (disabled)

Macro	I2S_CHANS_DAC
Description	Number of IS2 channesl to DAC/CODEC. Must be a multiple of 2. Default: NONE (Must be defined by app)

Macro	I2S_CHANS_ADC
Description	Number of I2S channels from ADC/CODEC. Must be a multiple of 2. Default: NONE (Must be defined by app)

7.1.3 Frequencies and Clocks

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

Macro	MIN_FREQ
Description	Min supported sample frequency for device (Hz). Default 44100

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

Macro	MCLK_441
Description	Master clock defines for 44100 rates (in Hz). Default: NONE (Must be defined by app)

Macro	MCLK_48
Description	Master clock defines for 48000 rates (in Hz). Default: NONE (Must be defined by app)

7.1.4 Audio Class

Macro	AUDIO_CLASS
Description	USB Audio Class Version. Default: 2 (Audio Class version 2.0)

Macro	AUDIO_CLASS_FALLBACK
Description	Whether or not to fall back to Audio Class 1.0 in USB Full-speed. Default: 0 (Disabled)

Macro	FULL_SPEED_AUDIO_2
Description	Whether or not to run UAC2 in full-speed. When disabled device can either operate in UAC1 mode in full-speed (if AUDIO_CLASS_FALLBACK enabled) or return “null” descriptors. Default: 1 (Enabled) when AUDIO_CLASS_FALLBACK disabled.

7.1.5 System Feature Configuration

7.1.5.1 MIDI

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

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

7.1.5.2 S/PDIF

Macro	SPDIF_TX
Description	Enables SPDIF Tx. Default: 0 (Disabled)

Macro	SPDIF_TX_INDEX
Description	Defines which output channels (stereo) should be output on S/PDIF. Note, Output channels indexed from 0. Default: 0 (i.e. channels 0 & 1)

Macro	SPDIF_RX
Description	Enables SPDIF Rx. Default: 0 (Disabled)

Macro	SPDIF_RX_INDEX
Description	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)

7.1.5.3 ADAT

Macro	ADAT_RX
Description	Enables ADAT Rx. Default: 0 (Disabled)

Macro	ADAT_RX_INDEX
Description	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 ADAT_RX enabled)

7.1.5.4 PDM Microphones

Macro	NUM_PDM_MICS
Description	Number of PDM microphones in the design. Default: None

7.1.5.5 DFU

Macro	DFU
Description	Enable DFU functionality. A driver required for Windows operation. Default: 1 (Enabled)

7.1.5.6 HID

Macro	HID_CONTROLS
Description	Enable HID playback controls functionality. 1 for enabled, 0 for disabled. Default 0 (Disabled)

7.1.5.7 CODEC Interface

Macro	CODEC_MASTER
Description	

7.1.6 USB Device Configuration

Macro	VENDOR_STR
Description	Vendor String used by the device. This is also pre-pended to various strings used by the design. Default: “XMOS”

Macro	VENDOR_ID
Description	USB Vendor ID (or VID) as assigned by the USB-IF. Default: 0x20B1 (XMOS)

Macro	PRODUCT_STR
Description	USB Product String for the device. If defined will be used for both PRODUCT_STR_A2 and PRODUCT_STR_A1 Default: Undefined

Macro	PRODUCT_STR_A2
Description	Product string for Audio Class 2.0 mode. Default: “xCore USB Audio 2.0”

Macro	PRODUCT_STR_A1
Description	Product string for Audio Class 1.0 mode. Default: “xCore USB Audio 1.0”

Macro	PID_AUDIO_1
Description	USB Product ID (PID) for Audio Class 1.0 mode. Only required if AUDIO_CLASS == 1 or AUDIO_CLASS_FALLBACK is enabled. Default: 0x0003

Macro	PID_AUDIO_2
Description	USB Product ID (PID) for Audio Class 2.0 mode. Default: 0x0002

Macro	BCD_DEVICE
Description	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).

7.1.7 Stream Formats

7.1.7.1 Output/Playback

Macro	OUTPUT_FORMAT_COUNT
Description	Number of supported output stream formats. Values 1,2,3 supported Default: 2

Macro	STREAM_FORMAT_OUTPUT_1_RESOLUTION_BITS

Continued on next page

Description	Sample resolution (bits) of output stream Alternate 1. Default: 24 if Alternate 1 is PCM, else 32 if DSD/RAW Note, 24 on the lowests alt in case of OUTPUT_FORMAT_COUNT = 1 leaving 24bit as the designs default resolution.

Macro	**STREAM_FORMAT_OUTPUT_2_RESOLUTION_BITS**
Description	Sample resolution (bits) of output stream Alternate 2. Default: 16 if Alternate 2 is PCM, else 32 if DSD/RAW

Macro	**STREAM_FORMAT_OUTPUT_3_RESOLUTION_BITS**
Description	Sample resolution (bits) of output stream Alternate 3. Default: 32 if Alternate 2 is PCM, else 32 if DSD/RAW

Macro	**HS_STREAM_FORMAT_OUTPUT_1_SUBSLOT_BYTES**
Description	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 catchs 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

Macro	**HS_STREAM_FORMAT_OUTPUT_2_SUBSLOT_BYTES**
Description	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 catchs 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

Macro	**HS_STREAM_FORMAT_OUTPUT_3_SUBSLOT_BYTES**

Continued on next page

Description	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 catchs 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

Macro	**FS_STREAM_FORMAT_OUTPUT_1_SUBSLOT_BYTES**
Description	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

Macro	**FS_STREAM_FORMAT_OUTPUT_2_SUBSLOT_BYTES**
Description	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

Macro	**FS_STREAM_FORMAT_OUTPUT_3_SUBSLOT_BYTES**
Description	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

Macro	**STREAM_FORMAT_OUTPUT_1_DATAFORMAT**
Description	Sample audio data-format if output stream Alternate 1. Default: UAC_FORMAT_TYPEI_RAW_DATA when Alternate 1 is RAW/DSD else UAC_FORMAT_TYPEI_PCM

Macro	STREAM_FORMAT_OUTPUT_2_DATAFORMAT
Description	Sample audio data-format if output stream Alternate 2. Default: UAC_FORMAT_TYPEI_RAW_DATA when Alternate 2 is RAW/DSD else UAC_FORMAT_TYPEI_PCM

Macro	STREAM_FORMAT_OUTPUT_3_DATAFORMAT
Description	Sample audio data-format if output stream Alternate 3. Default: UAC_FORMAT_TYPEI_RAW_DATA when Alternate 3 is RAW/DSD else UAC_FORMAT_TYPEI_PCM

7.1.7.2 Input/Recording

Macro	INPUT_FORMAT_COUNT
Description	Number of supported input stream formats. Default: 1

Macro	STREAM_FORMAT_INPUT_1_RESOLUTION_BITS
Description	Sample resolution (bits) of input stream Alternate 1. Default: 24

Macro	HS_STREAM_FORMAT_INPUT_1_SUBSLOT_BYTES
Description	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 catchs 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

Macro	FS_STREAM_FORMAT_INPUT_1_SUBSLOT_BYTES

Continued on next page

Description	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: STREAM_FORMAT_INPUT_1_RESOLUTION_BITS / 8

Macro	**STREAM_FORMAT_INPUT_1_DATAFORMAT**
Description	Sample audio data-format for input stream Alternate 1. Default: UAC_FORMAT_TYPEI_PCM

7.1.8 Volume Control

Macro	**OUTPUT_VOLUME_CONTROL**
Description	Enable/disable output volume control including all processing and descriptor support. Default: 1 (Enabled)

Macro	**INPUT_VOLUME_CONTROL**
Description	Enable/disable input volume control including all processing and descriptor support. Default: 1 (Enabled)

Macro	**MIN_VOLUME**
Description	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)

Macro	**MAX_VOLUME**
Description	The maximum volume setting. This is a signed 8.8 fixed point number. Default: 0x0000 (0db)

Macro	VOLUME_RES
Description	The resolution of the volume control in db as a 8.8 fixed point number. Default: 0x100 (1db)

7.1.9 Mixing Parameters

Macro	MIXER
Description	Enable “mixer” core. Default: 0 (Disabled)

Macro	MAX_MIX_COUNT
Description	Number of seperate mixes to perform. Default: 8 if MIXER enabled, else 0

Macro	MIX_INPUTS
Description	Number of channels input into the mixer. Note, total number of mixer nodes is MIX_INPUTS * MAX_MIX_COUNT Default: 18

Macro	MIN_MIXER_VOLUME
Description	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)

Macro	MAX_MIXER_VOLUME
Description	The maximum volume setting for the mixer. This is a signed 8.8 fixed point number. Default: 0x0000 (0db)

Macro	VOLUME_RES_MIXER
Description	The resolution of the volume control in db as a 8.8 fixed point number. Default: 0x100 (1db)

7.1.10 Power

Macro	SELF_POWERED
Description	Report as self to the host when enabled, else reports as bus-powered. This affects descriptors and XUD usage. Default: 0 (Disabled)

Macro	BMAX_POWER
Description	Power drawn from the host (in mA x 2). Default: 0 when SELF_POWERED enabled else 250 (500mA)

7.2 Required User Function Definitions

The following functions need to be defined by an application using the XMOS USB Audio framework.

7.2.1 External Audio Hardware Configuration Functions

Function	AudioHwInit	
Description	This function is called when the audio core starts after the device boots up and should initialize the external audio harware e.g. clocking, DAC, ADC etc	
Type	`void AudioHwInit(chanend ?c_codec)`	
Parameters	`c_codec`	An optional chanend that was original passed into audio() that can be used to communicate with other cores.

Function	AudioHwConfig
Description	This function is called when the audio core starts or changes sample rate. It should configure the extenal audio hardware to run at the specified sample rate given the supplied master clock frequency.

Continued on next page

Type	`void AudioHwConfig(unsigned samFreq,`
	`unsigned mclk,`
	`chanend ?c_codec,`
	`unsigned dsdMode,`
	`unsigned sampRes_DAC,`
	`unsigned sampRes_ADC)`	

Parameters	`samFreq`	The sample frequency in Hz that the hardware should be configured to (in Hz).
	`mclk`	The master clock frequency that is required in Hz.
	`c_codec`	An optional chanend that was original passed into audio() that can be used to communicate with other cores.
	`dsdMode`	Signifies if the audio hardware should be configured for DSD operation
	`sampRes_DAC`	The sample resolution of the DAC stream
	`sampRes_ADC`	The sample resolution of the ADC stream

7.2.2 Audio Streaming Functions

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

Function	**AudioStreamStart**
Description	This function is called when the audio stream from device to host starts.
Type	`void AudioStreamStart(void)`

Function	**AudioStreamStop**
Description	This function is called when the audio stream from device to host stops.
Type	`void AudioStreamStop(void)`

7.2.3 Host Active

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

This is called on a change in state.

Function	AudioStreamStart	
Description		
Type	`void AudioStreamStart(int active)`	
Parameters	`active`	Indicates if the host is active or not. 1 for active else 0.

7.2.4 HID Controls

The following function is called when the device wishes to read physical user input (buttons etc).

Function	UserReadHIDButtons	
Description		
Type	`void UserReadHIDButtons(unsigned char hidData[])`	
Parameters	`hidData`	The function should write relevant HID bits into this array. The bit ordering and functionality is defined by the HID report descriptor used.

7.3 Component API

The following functions can be called from the top level main of an application and implement the various components described in §3.1.

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

Continued on next page

Type	int XUD_Manager(chanend c_epOut[], int noEpOut, chanend c_epIn[], int noEpIn, chanend ?c_sof, XUD_EpType epTypeTableOut[], XUD_EpType epTypeTableIn[], out port ?p_usb_rst, clock ?clk, unsigned rstMask, XUD_BusSpeed_t desiredSpeed, XUD_PwrConfig pwrConfig)

Continued on next page

Parameters	`c_epOut`	An array of channel ends, one channel end per output endpoint (USB OUT transaction); this includes a channel to obtain requests on Endpoint 0.
	`noEpOut`	The number of output endpoints, should be at least 1 (for Endpoint 0).
	`c_epIn`	An array of channel ends, one channel end per input endpoint (USB IN transaction); this includes a channel to respond to requests on Endpoint 0.
	`noEpIn`	The number of input endpoints, should be at least 1 (for Endpoint 0).
	`c_sof`	A channel to receive SOF tokens on. This channel must be connected to a process that can receive a token once every 125 ms. If tokens are not read, the USB layer will lock up. If no SOF tokens are required `null` should be used for this parameter.
	`epTypeTableOut`	See `epTypeTableIn`.
	`epTypeTableIn`	This and `epTypeTableOut` are two arrays indicating the type of the endpoint. Legal types include: `XUD_EPTYPE_CTL` (Endpoint 0), `XUD_EPTYPE_BUL` (Bulk endpoint), `XUD_EPTYPE_ISO` (Isochronous endpoint), `XUD_EPTYPE_INT` (Interrupt endpoint), `XUD_EPTYPE_DIS` (Endpoint not used). The first array contains the endpoint types for each of the OUT endpoints, the second array contains the endpoint types for each of the IN endpoints.
	`p_usb_rst`	The port to used to connect to an external phy reset line. Should be `null` for U-Series.
	`clk`	The clock block to use for the p_usb_rst port - this should not be clock block 0. Should be `null` for U-Series.
	`rstMask`	The mask to use when taking an external phy into/out of reset. The mask is ORed into the port to disable reset, and unset when deasserting reset. Use '-1' as a default mask if this port is not shared.
	`desiredSpeed`	This parameter specifies what speed the device will attempt to run at i.e. full-speed (ie 12Mbps) or high-speed (480Mbps) if supported by the host. Pass `XUD_SPEED_HS` if high-speed is desired or `XUD_SPEED_FS` if not. Low speed USB is not supported by XUD.
	`pwrConfig`	Specifies whether the device is bus or self-powered. When self-powered the XUD will monitor the VBUS line for host disconnections. This is required for compliance reasons. Valid values are `XUD_PWR_SELF` and `XUD_PWR_BUS`.

Continued on next page

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)

Function	**Endpoint0**
Description	Function implementing Endpoint 0 for enumeration, control and configuration of USB audio devices. It uses the descriptors defined in `descriptors_2.h`.

Type	`void Endpoint0(chanend c_ep0_out,`
	`chanend c_ep0_in,`
	`chanend c_audioCtrl,`
	`chanend ?c_mix_ctl,`
	`chanend ?c_clk_ctl,`
	`chanend ?c_EANativeTransport_ctr,`
	`client interface i_dfu dfuInterface)`

Continued on next page

Parameters	`c_ep0_out`	Chanend connected to the XUD_Manager() out endpoint array
	`c_ep0_in`	Chanend connected to the XUD_Manager() in endpoint array
	`c_audioCtrl`	Chanend connected to the decouple thread for control audio (sample rate changes etc.)
	`c_mix_ctl`	Optional chanend to be connected to the mixer thread if present
	`c_clk_ctl`	Optional chanend to be connected to the clockgen thread if present.
	`c_usb_test`	Optional chanend to be connected to XUD if test modes required.
	`c_EANativeTransport_ctrl`	Optional chanend to be connected to EA Native endpoint manager if present

Function	**buffer**
Description	USB Audio Buffering Thread. This function buffers USB audio data between the XUD layer and the decouple thread. Most of the chanend parameters to the function should be connected to XUD_Manager()

Type	`void buffer(chanend c_aud_out,`
	`chanend c_aud_in,`
	`chanend c_aud_fb,`
	`chanend c_sof,`
	`chanend c_aud_ctl,`
	`in port p_off_mclk)`

Continued on next page

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_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)

Function	**decouple**
Description	Manage the data transfer between the USB audio buffer and the Audio I/O driver.
Type	`void decouple(chanend c_audio_out)`
Parameters	`c_audio_out` Channel connected to the audio() or mixer() threads

Function	**mixer**
Description	Digital sample mixer. This thread mixes audio streams between the decouple() thread and the audio() thread.
Type	`void mixer(chanend c_to_host, chanend c_to_audio, chanend c_mix_ctl)`

Continued on next page

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

Function	**audio**
Description	The audio driver thread. This function drives I2S ports and handles samples to/from other digital I/O threads.
Type	`void audio(chanend c_in, chanend ?c_config, chanend ?c_adc)`

Parameters	`c_in`	Audio sample channel connected to the mixer() thread or the decouple() thread
	`c_dig`	channel connected to the clockGen() thread for receiving/transmitting samples
	`c_config`	An optional channel that will be passed on to the CODEC configuration functions.

Function	**clockGen**
Description	Clock generation and digital audio I/O handling.

Type	`void`
	`clockGen(streaming chanend ?c_spdif_rx,`
	`chanend ?c_adat_rx,`
	`out port p,`
	`chanend c_audio,`
	`chanend c_clk_ctl,`
	`chanend c_clk_int)`

Continued on next page

Parameters	`c_spdif_rx`	channel connected to S/PDIF receive thread
	`c_adat_rx`	channel connect to ADAT receive thread
	`p`	port to output 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

Function	**SpdifReceive**
Description	S/PDIF receive function. This function needs 1 thread and no memory other than ~2800 bytes of program code. It can do 11025, 12000, 22050, 24000, 44100, 48000, 88200, 96000, and 192000 Hz. When the decoder encounters a long series of zeros it will lower the divider; when it encounters a short series of 0-1 transitions it will increase the divider. Output: the received 24-bit sample values are output as a word on the streaming channel end. Each value is shifted up by 4-bits with the bottom four bits being one of FRAME_X, FRAME_Y, or FRAME_Z. The bottom four bits should be removed whereupon the sample value should be sign extended. The function does not return unless compiled with TEST defined in which case it returns any time that it loses synchronisation.

Type	`void SpdifReceive(in buffered port:4 p,`
	`streaming chanend c,`
	`int initial_divider,`
	`clock clk)`

Parameters	`p` S/PDIF input port. This port must be 4-bit buffered, declared as `in buffered port:4` `c` channel to output samples to `initial_divider` initial divide for initial estimate of sample rate For a 100Mhz reference clock, use an initial divider of 1 for 192000, 2 for 96000/88200, and 4 for 48000/44100. `clk` clock block sourced from the 100 MHz reference clock.

Function	adatReceiver48000
Description	ADAT Receive Thread (48kHz sample rate). When a data rame is received, samples will be output onto the streaming channel At first a word 0x000000Z1 will be output, where Z are the user data; after that eight words 0x0ZZZZZZ0 will be output where ZZZZZZ is a 24-bit sample value. The eight words may refer to sample values on eight channels, or on fewer channels if muxing is used. The function will return if it cannot lock onto a 48,000 Hz signal. Normally the 48000 function is called in a while(1) loop. If both 44,100 and 48,000 need to be supported, they should be called in sequence in a while(1) loop. Note that the functions are large, and that 44,100 should not be called if 44.1 KHz does not need to be supported.
Type	`void` `adatReceiver48000(buffered in port:32 p, chanend oChan)`
Parameters	`p` ADAT port - should be 1-bit and clocked at 100MHz `oChan` channel on which decoded samples are output

Function	adatReceiver44100
Description	ADAT Receive Thread (44.1kHz sample rate). When a data rame is received, samples will be output onto the streaming channel At first a word 0x000000Z1 will be output, where Z are the user data; after that eight words 0x0ZZZZZZ0 will be output where ZZZZZZ is a 24-bit sample value. The eight words may refer to sample values on eight channels, or on fewer channels if muxing is used. The function will return if it cannot lock onto a 44,100 Hz signal. Normally the 44,100 function is called in a while(1) loop. If both 44,100 and 48,000 need to be supported, they should be called in sequence in a while(1) loop. Note that the functions are large, and that 48,000 should not be called if 48 Khz does not need to be supported.
Type	`void` `adatReceiver44100(buffered in port:32 p, chanend oChan)`
Parameters	`p` ADAT port - should be 1-bit and clocked at 100MHz `oChan` channel on which decoded samples are output

Function	usb_midi
Description	USB MIDI I/O thread. This function passes MIDI data from USB to UART I/O.

Type	`void`
	`usb_midi(buffered in port:1 ?p_midi_in,`
	`port ?p_midi_out,`
	`clock ?clk_midi,`
	`chanend ?c_midi,`
	`unsigned cable_number,`
	`chanend ?c_iap,`
	`chanend ?c_i2c,`
	`port ?p_scl,`
	`port ?p_sda)`

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.

8 Frequently Asked Questions

Why does the USBView tool from Microsoft show errors in the devices descriptors?

The USBView tool supports USB Audio Class 1.0 only

How do I set the maximum sample rate of the device?

See MAX_FREQ define in *usb_audio_sec_custom_defines_api*