

USB Audio Software Design Guide

Document Revision 1.2

Publication Date: 2011/01/14

Copyright © 2011 XMOS Limited, All Rights Reserved.



1 Overview

The XMOS USB audio solution is an *USB Audio Class 2.0* compliant device over high speed USB 2.0. Based on the XMOS XS1 architecture, it supports USB Audio Class 2.0 asynchronous mode at up to 192kHz. The complete source code, together with the free XMOS development tools and XMOS programmable devices allow the implementer to select the exact mix of interfaces and processing required. DSP algorithms can be implemented using the available bandwidth of the XCore processors.

In addition to a mix of components to build a USB audio application, XMOS provide two reference designs: *USB Audio 2.0 Reference Design (XS1-L1)* and the *USB Audio 2.0 Multichannel Reference Design (XS1-L2)*. These are applications based on the USB audio framework with a particular qualified feature set and an accompanying reference hardware platform.

This software design guide assumes the reader is familiar with the XC language and XMOS XS1 devices. For more information see [\[XC09\]](#) or go to: www.xmos.com

1.1 Summary

Functionality	
Provides USB interface to digital audio I/O.	
Supported Standards	
USB	USB 1.0 USB 2.0 USB Audio Class 1.0 [USBAud10] USB Audio Class 2.0 [USBAud20] USB Firmware Upgrade 1.1 [DFU11] USB Midi Device 1.0 [USBMidi10]
Audio	I2S S/PDIF ADAT MIDI
Supported Sample Frequencies	
44.1kHz, 48kHz, 88.2kHz, 96kHz, 176.4kHz, 192kHz	
Supported Devices	
XMOS Devices	XS1-L1 XS1-L2
Requirements	
Development Tools	XMOS Development Tools v10.4 or later
USB	1 x USB Phy (ULPI)
Audio	Audio input/output device supporting I2S
Boot/Storage	Compatible SPI Flash device
Licensing and Support	
Reference code provided without charge under license from XMOS. Contact support@xmos.com for details. Reference code is maintained by XMOS Limited.	

2 Hardware Platforms

The following sections describe the hardware platforms that support development with the XMOS USB audio software platform.

2.1 USB Audio 2.0 Hardware Reference Design (XS1-L1)

The USB Audio 2.0 Reference Design (XS-L1) is a hardware reference design available from XMOS:

<http://www.xmos.com/products/development-kits/usbaudio2>

The following diagram shows the block layout of the USB Audio 2.0 Reference Design board. The main purpose of the XS1-L1 is to provide a USB audio interface to the USB PHY and route the audio to the audio CODEC and S/PDIF output. Note that although the software supports MIDI, there are no MIDI connectors on the board. For full details please refer to the hardware manual found via on the website.

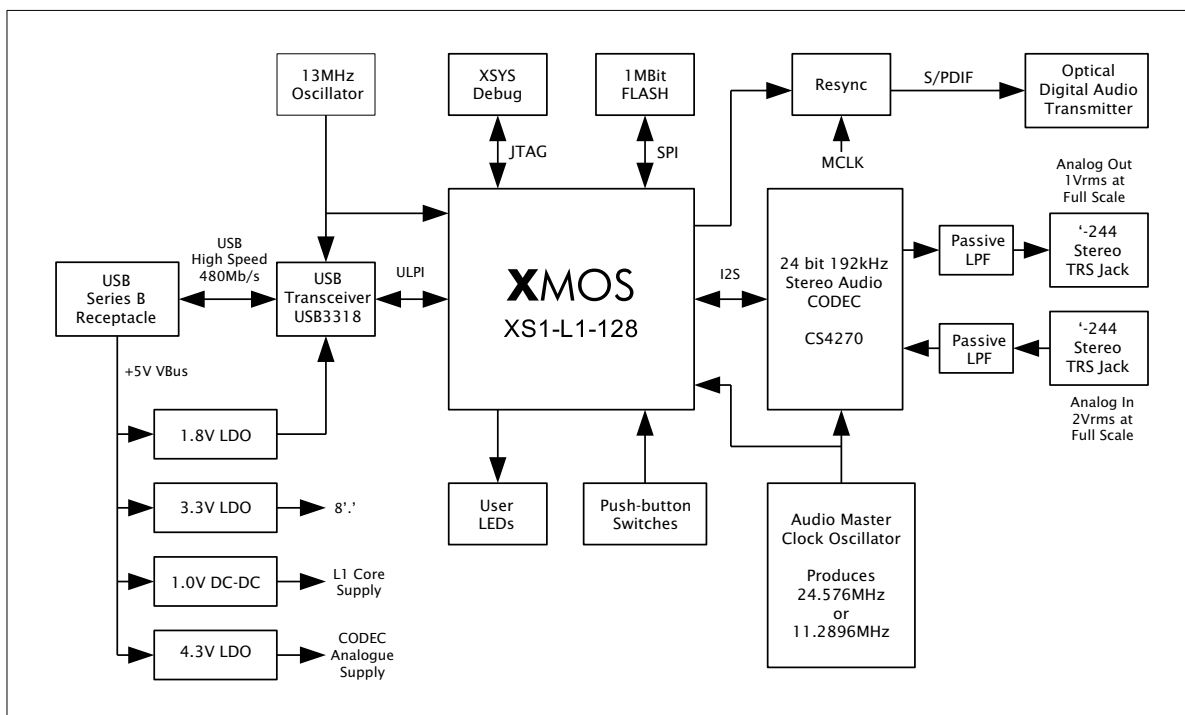


Figure 1: USB Audio 2.0 Reference Design Block Diagram

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 3.2.

2.2 USB Audio 2.0 Multichannel Hardware Reference Design (XS1-L2)

The USB Audio 2.0 Multichannel Reference Design (XS-L2) is a hardware reference design available from XMOS:

<http://www.xmos.com/products/development-kits/usbaudio2mc>

The following diagram shows the block layout of the USB Audio 2.0 Multichannel Reference Design board. The board supports six analogue inputs, eight analogue outputs, digital input, digital output and MIDI. For full details please refer to the hardware manual available on the website.

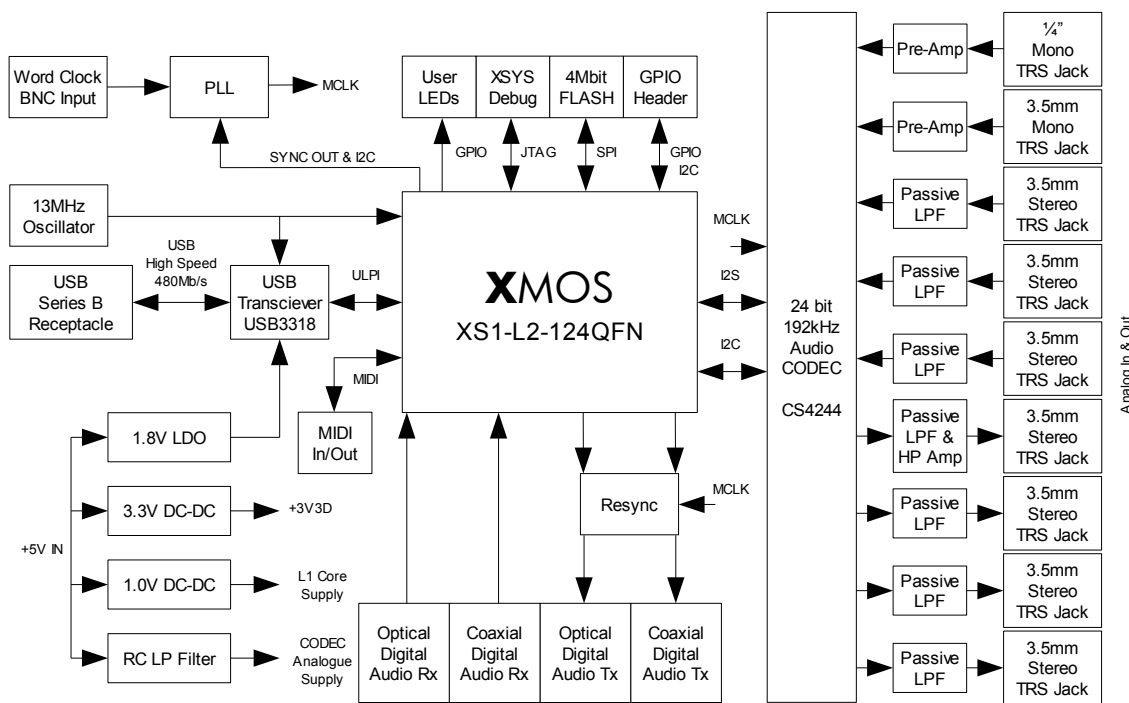


Figure 2: USB Audio 2.0 Multichannel Reference Design Block Diagram

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 3.3.

3 System Description

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

3.1 The USB Audio 2.0 System Architecture

The XMOS USB Audio 2.0 System consists of a series of communicating components. Every system has the following core components:

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.
Endpoint buffer	Buffers audio 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.
Audio Driver	Handles digital audio I/O over I2S and manages audio data from other digital audio I/O components.

Table 1: Core Components

In addition the following optional components can be added:

Component	Description
Device Firmware Upgrade (DFU)	Allows firmware upgrade over USB. This is an optional part of the Endpoint 0 component.
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.
ADAT Receiver	Inputs samples of an ADAT digital audio interface.
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.

Table 2: Optional Components



S/PDIF or ADAT input require use of the Clockgen component and an external clock generator for clock recovery.

The following diagram shows how the components interact with each other:

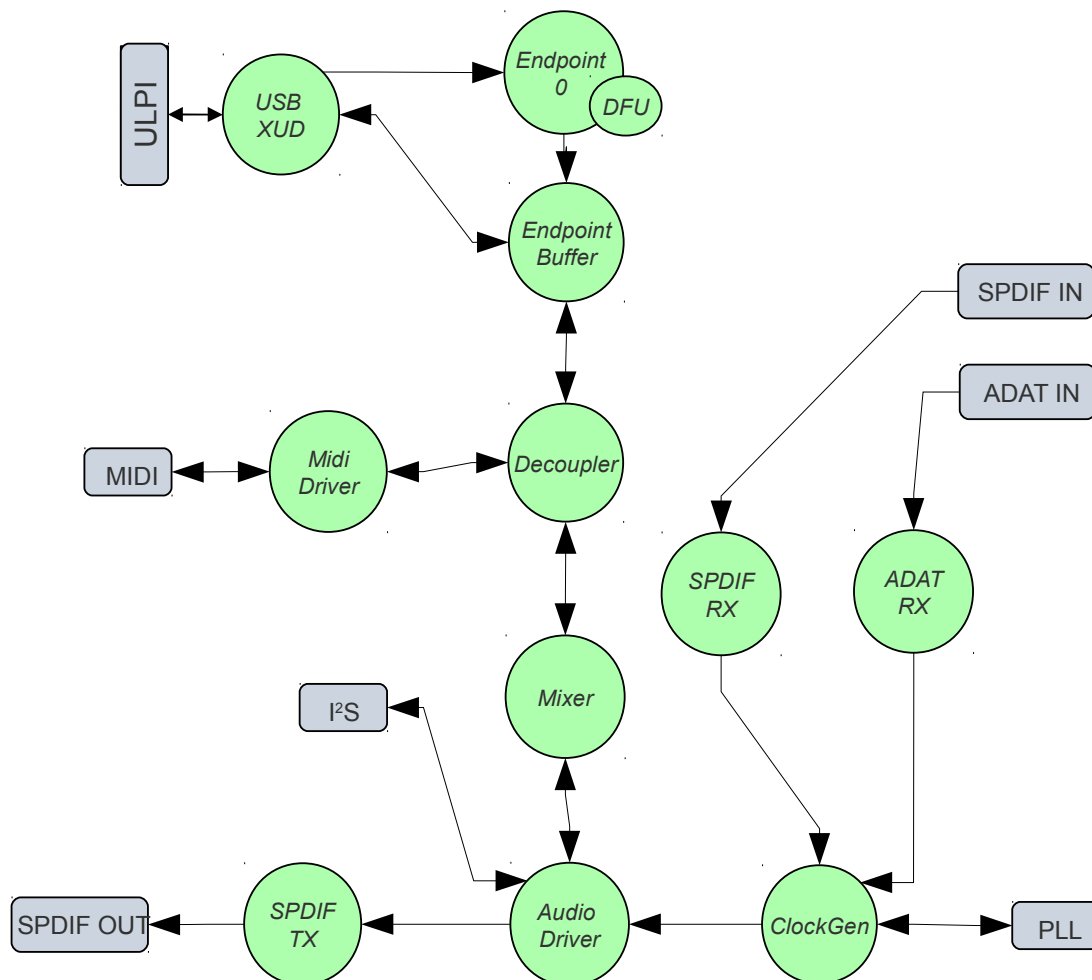


Figure 3: USB Audio Component Architecture

In addition to the overall framework, two reference design applications are provided (described in Sections 3.2 and 3.3). These applications provide qualified configurations of the framework which support and are validated on accompanying hardware.

3.2 The USB Audio 2.0 Reference Design (XS1-L1)

The USB Audio 2.0 Reference Design is an application of the USB audio framework specifically for the hardware described in Section 2.1 and is implemented on the XS1-L1 single core device (500MIPS). 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 following diagrams show the software layout of the code running on the XS1-L1 chip. Each unit runs in a single thread concurrently with the others units. The lines show the communication between each functional unit. Due to the MIPS requirement of the USB driver (see Section 3.13), only six threads can be run on the single core L1 device so only one of S/PDIF transmit or MIDI can be supported.

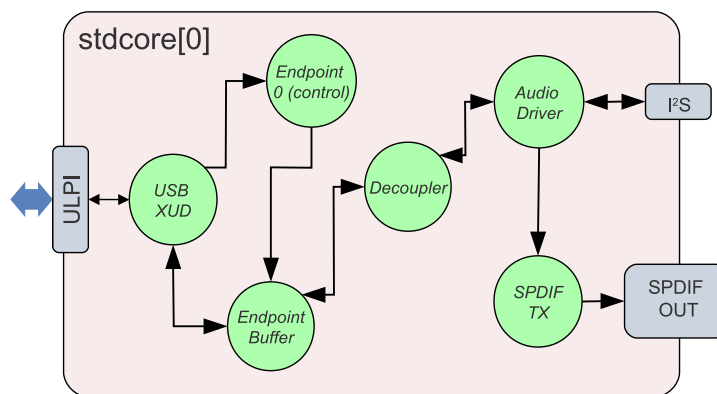


Figure 4: L1 Software Thread Diagram (with S/PDIF TX)

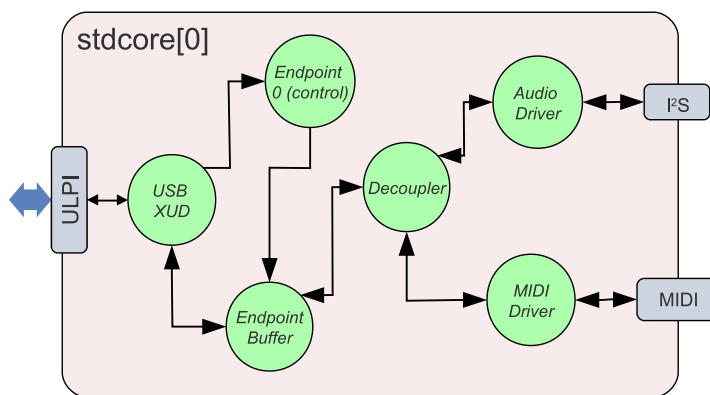


Figure 5: L1 Software Thread Diagram (with MIDI I/O)

3.2.1 Port 32A

Port 32A on the XS1-L1 is a 32-bit wide port that has several separate signal bit signals connected to it, accessed by multiple threads. 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 is outside the XC usage model so is implemented in assembly in the file `port32A.S`. The key functions are `port32A_peek` which gets the current output value on the port and `port32a_out` which outputs to the port.

The following table 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-L1.

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

Table 3: Port 32A Signals

3.2.2 Clocking

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

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

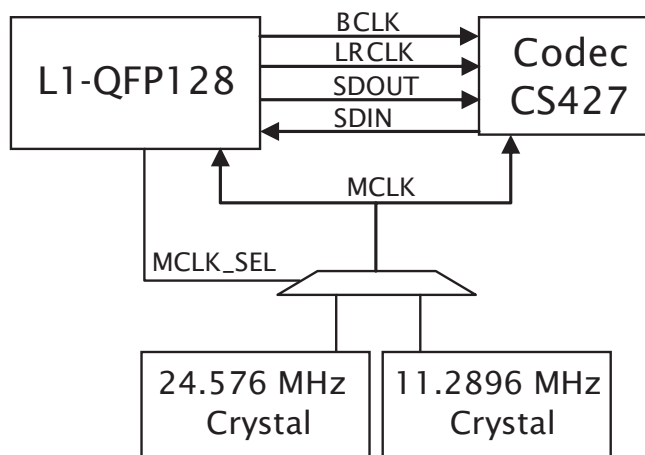


Figure 6: Audio Clock Connections

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:

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

Table 4: CODEC Modes

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

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.

3.2.3 Validated Build Options

The reference design can be built in several ways by changing the option described in 5.1. However, the design has only been validated against the build options as set in the application as distributed with the following two variations.

Configuration 1 This configuration is with all the #defines set as per the distributed application. It has the mixer disabled, supports 2 channels in, 2 channels out and supports sample rates up to 192kHz and S/PDIF transmit.

Configuration 2 This configuration disabled S/PDIF and enables MIDI.

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

```
//#define SPDIF          1
```

and changing the MIDI define to:

```
#define MIDI             1
```

3.3 The USB Audio 2.0 Multichannel Reference Design (XS1-L2)

The USB Audio 2.0 Multichannel Reference Design is an application of the USB audio framework specifically for the hardware described in Section 2.1 and is implemented on an XS1-L2 single core 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

The following thread diagram shows the software layout of the USB Audio 2.0 Multichannel Reference Design.

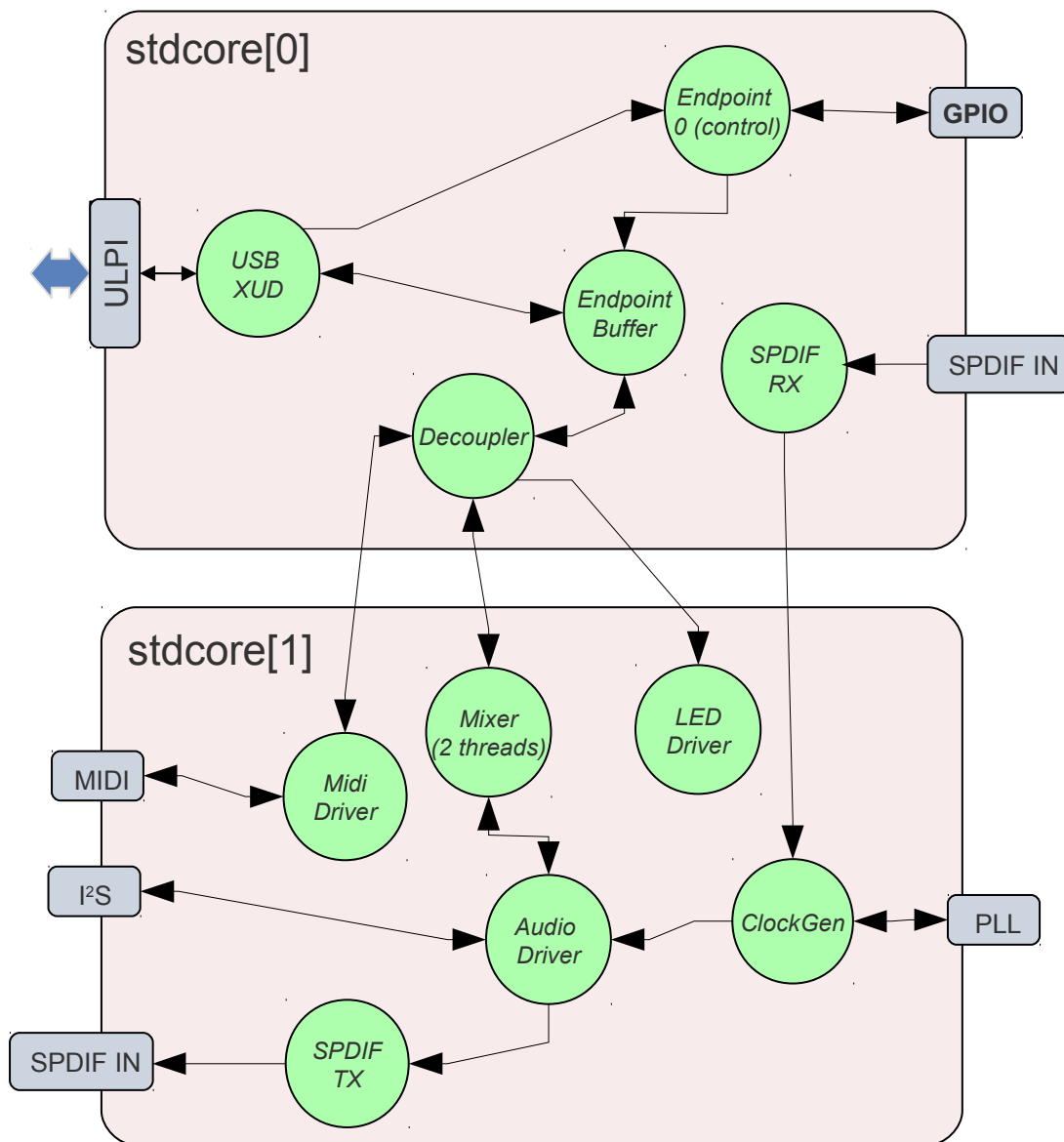


Figure 7: L2 Reference Design Thread Layout

3.3.1 Clocking

For complete clocking flexibility the L2 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 Section 3.11.

3.3.2 Validated Build Options

The reference design can be built in several ways by changing the option described in 5.1. However, the design has only been validated against the build options as set in the application as distributed with the following four variations.

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.

Configuration 2 The same as Configuration 1 but with the CODEC set as I2S master (and the XCore as slave).

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

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

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)
```

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
```

3.4 USB Interface

The low-level USB interface is controlled by the XMOS USB Device (XUD) driver. This driver is described in the XUD library documentation [\[Xud\]](#).

The low level USB interface is serviced by three threads:

- The Endpoint 0 thread controls the enumeration/configuration tasks of the USB device.
- The buffering thread sends/receives audio data packets from the XUD library over XC channels and places them into memory buffers.
- The decoupler reads/writes packets to/from the buffering thread via memory and send/receives samples (and optionally MIDI data) over XC channels to the audio components of the software.

3.4.1 Audio Class 1.0

The default for the application is to run as a USB Audio Class 2.0 device. However, the device can also run as a USB Audio Class 1.0 device. Note that to ensure specification compliance, Audio Class 1.0 mode always operates at full-speed.

The device will operated in full-speed Audio Class 1.0 mode if:

- 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 Audio Class fall back is enabled).

The options to control this behavior are detailed in Section [5.1](#). When running in Audio Class 1.0 mode the following restrictions apply:

- MIDI is disabled.

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.

3.4.2 Endpoint 0: Management and Control

Endpoint 0 (`endpoint0.xc`) controls the management tasks of the USB device. These tasks can be split into enumeration, reset, audio configuration and firmware upgrade.

When the host sends a packet to Endpoint 0, the XUD library layer first examines the packet to handle low level management. After this the Endpoint 0 thread inspects the packet and sends it to one of the following subsystems depending on the packet:

- The Audio Class Interface
- The DFU Interface

Finally, if the XUD layer indicates that a reset is required, the reset request is handled.

Startup/Enumeration When the device is first attached to a host, enumeration occurs. The device presents several interfaces to the host.

Mode	Interfaces
Application mode	Audio Class 2/Audio Class 1 DFU Class 1.1 MIDI Device Class 1.0
DFU mode	DFU Class 1.1

Table 5: USB devices presented to host

The device initially starts in Application mode.

Section 3.5 describes how DFU mode is used. The audio device class (1 or 2) is set at compile time—see 5.1.

Enumeration requests are largely handled by the call to `DescriptorRequests()` using data structures found in the `descriptors_2.h` file. These structures use defines which can be customized—see 5.1.

After enumeration a message is sent to the audio driver thread. This contains either an audio frequency to run at, or a special DFU message which conveys that the device is in DFU mode and no audio is required.

Reset On receiving a reset request, three steps occur:

1. Depending on the DFU state, the device may be set into DFU mode.
2. A message is sent to the audio driver to alert it of an impending reset. At this point audio capture/playback is stopped until the device restarts.
3. The low-level XUD function is called to perform a USB reset.

Audio Request: Setting The Sample Rate When the host requests a change of sample rate, it sends a command to Endpoint 0. The device initiates a reset. When the device restarts, the new sample rate is registered with the audio driver thread.

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

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

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

If the effect of the volume control array on the audio input and output is implemented by the decoupler, the decoupler thread reads the volume values from this array. Note that this array is shared between Endpoint 0 and the decoupler thread. This is done in a safe manner, since only Endpoint 0 can write to the array, word update is atomic between threads and the decoupler thread only reads from the array (ordering between writes and reads is unimportant in this case). Inline assembly is used by the decoupler thread to access the array, avoiding the parallel usage checks in 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 Section 3.7.

3.4.3 Audio Endpoints (Endpoint Buffer and Decoupler)

Endpoint Buffer All endpoints other than Endpoint 0 are handled in one thread. This thread is implemented in the file `usb_buffer.xc`. This loop *must* be responsive to the XUD library and as such it simply receives or transmits buffers which are passed to it via shared memory from the decoupler.

This thread is also responsible for feedback calculation based on SOF notification and reads from the port counter of a port connected to the master clock.

Decoupler The decoupler supplies the USB buffering thread with buffers to transmit/receive data 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 thread 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`.

Buffering Scheme Both audio and MIDI use a similar buffering scheme for USB data. This scheme is executed by co-operation between the buffering thread, the decouple thread and the XUD library.

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

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

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

1. The decouple thread passes a pointer to the buffering thread pointing into a FIFO of data and signals to the XUD library that the buffering thread is ready to receive.
2. The buffering thread 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 thread updates the write pointer of the FIFO and provides a new pointer to the buffering thread to fill.
4. Upon request from the audio thread, the decoupler thread sends samples to the audio thread by reading samples out of the FIFO.

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

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

The first operation the interrupt handler does 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 thread 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 thread 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
inuint() outuint()	outuint()	Audio system requests sample exchange Interrupt fires and inuint performed Decouple sends ack
	testct()	Checks for CT indicating SF change
inuint() inuint() inuint() ...	inuint()	Word indication ACK input (No SF change) Sample transfer (Device to Host)
outuint() outuint() outuint() outuint() ...		Sample transfer (Host to Device)

Table 6: Decouple/Audio System Channel Communication

3.4.4 Aysnc Feedback

The device uses a feedback endpoint to report the rate at which audio is output to the CODEC. This feedback is in accordance with the *USB Audio Class 2.0 specification*.

After each received USB SOF token, the buffering thread takes a timestamp from a port clocked off the master clock. By subtracting the timestamp 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 thread.

3.4.5 USB Rate Control

The Audio thread 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. The following table shows the allowed number of samples per packet for each example audio frequency.

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

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

Table 7: Allowed samples per packet

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

3.5 Device Firmware Upgrade (DFU)

The DFU interface handles updates to the boot image of the device. The interface links USB to the XMOS flash user library (see [\[ToolsUserGuide\]](#)). 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_RESETEDEVICE 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 *USB DFU 1.1 class specification* [\[DFU11\]](#). 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`.

3.6 Audio Driver

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

The audio thread 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 XS1 device acts as the master generating the Bit Clock (BCLK) and Left-Right Clock (LRCLK, also called Word Clock) signals. Although the reference designs use the Cirrus CS4270/CS42448, any CODEC that supports I2S and can be used.

The following table shows the signals used to communicate audio between the XS1 device and the CODEC.

Signal	Description
LRCLK	The word clock, transition at the start of a sample
BCLK	The bit clock, clocks data in and out
SDIN	Sample data in (from CODEC to XS1-L1)
SDOUT	Sample data out (from XS1-L1 to CODEC)
MCLK	The master clock running the CODEC

Table 8: CODEC Signals

The bit clock controls the rate at which data is transmitted to and from the CODEC. In the case where the XS1 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.

The following table shows some example clock frequencies and divides for different sample rates (note that this reflects the L1 reference board configuration):

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

Table 9: Clock Divides used in L1 Ref Design

The master clock must be supplied by an external source eg. clock generator chip, 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 XS1 device.

3.6.1 Port Configuration (CODEC Slave)

The default software configuration is CODEC Slave (XS1 master). That is, the XS1 provides the BCLK and LRCLK signals to the CODEC.

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

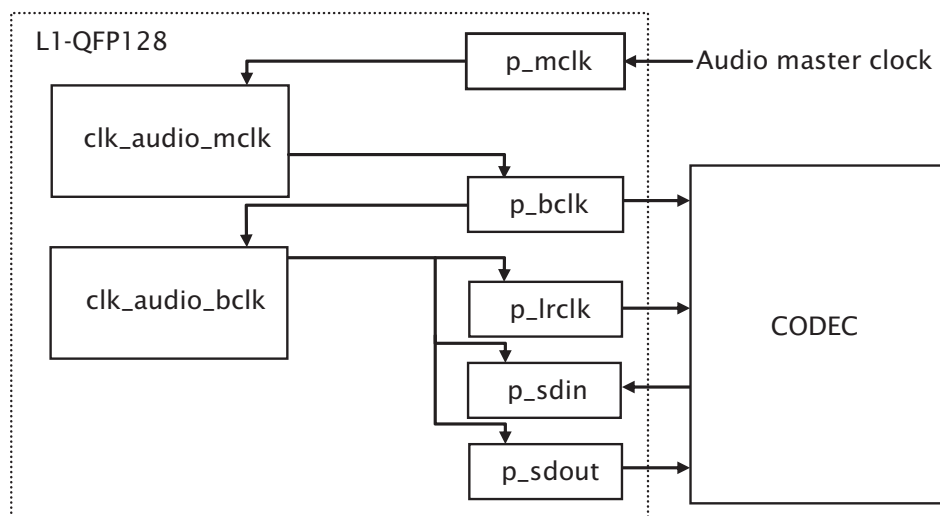


Figure 8: Ports and Clocks (CODEC slave)

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

The L1 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.

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 MCLK and BCLK. The following table shows the pattern for different values of this divide:

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

Table 10: Output patterns

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

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

3.6.2 Changing Audio Sample Frequency

When the host changes sample frequency, a new frequency is sent to the audio driver thread by Endpoint 0. First, a change of sample frequency is reported by sending the new frequency over an XC channel. The audio thread detects this using the `select` function on a channel (a default case such that processing can continue if no signal is present on the channel).

Upon receiving the change of sample frequency request, the audio thread stops the I2S interface and calls the CODEC/port configuraton functions. Once this is complete, the I2S interface is restarted at the new frequency.

3.7 Digital Mixer

The mixer thread takes outgoing audio from the decoupler and incoming audio from the audio driver. 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 threads and can perform eight mixes with up to 18 inputs at sample rates up to 96kHz and two mixes with up to 18 inputs at higher sample rates. The component automatically 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.
- 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 than the 18 possible mix inputs to each mixer, there is a mapping from all the possible inputs to the mixer inputs.

After the mix occurs, the final outputs are created. There are two output destinations:

- The USB inputs to the host—these samples are sent to the decoupler.
- 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 3.4.2, 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 thread.

3.7.1 Control

The mixers can receive the following control commands from the Endpoint 0 thread:

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

Table 11: Mixer Component Commands

3.7.2 Host Control

The mixer can be controlled from a host PC. XMOS has a sample application showing how to control the mixer. For details, contact XMOS.

3.8 S/PDIF Transmit

XS1 devices can support S/PDIF transmit up to 192kHz. The S/PDIF transmitter uses a lookup table to encode the audio data. It receives samples from the Audio thread two at a time, one for each channel. For each sample, it performs a lookup on each byte, generating 16 bits of encoded data which it outputs to the port.

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

The thread takes PCM audio samples via a channel and outputs them in S/PDIF format to a port. Audio samples are encapsulated into S/PDIF words (adding preamble, parity, channel status and validity bits) and transmitted in biphase-mark encoding (BMC) with respect to an *external* master clock. Note that a minor change to the SpdifTransmitPortConfig function would enable *internal* master clock generation (e.g. when clock source is already locked to desired audio clock).

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

Table 12: S/PDIF Capabilities

3.8.1 Clocking

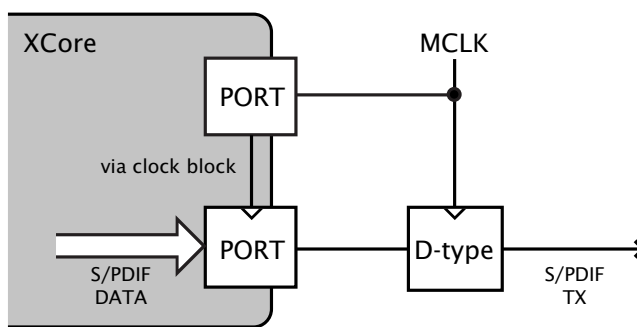


Figure 9: D-Type Jitter Reduction

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). 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).

3.8.2 Usage

The interface is normal channel with streaming built-ins (outuint, inuint). Data format is 24-bit left-aligned in a 32-bit word: 0x12345600

The following protocol is used on the channel:

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

Table 13: S/PDIF Component Protocol

3.8.3 Output stream structure

The stream is composed of words with the following structure:

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

Table 14: S/PDIF Stream Structure

The channel status bits are 0x0nc07A4, where c=1 for left channel, c=2 for right channel and n indicates sampling frequency:

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

Table 15: Channel Status Bits

3.9 S/PDIF Receive

XS1 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 outputs audio samples over a *streaming channel end* where data can be input using the built-in input operator.

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

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

Table 16: S/PDIF RX Word Structure

The tag has one of three values:

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

Table 17: S/PDIF RX Tags

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

3.9.1 Integration

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 Clock Recovery).

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

If bad parity is detected the word/sample is ignored, else the tag is inspected for channel 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;
    sample = (data << 4) & 0xFFFFF00;

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

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

3.10 ADAT Receive

The ADAT 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 [5.3](#).

The component outputs 32 bits words split into nine word frames. The frames are layed 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

The following code is an example of code that could read the output of the ADAT component:

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

The 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 enables synchronization at a higher level, in that on the channel a single odd word is always read followed by eight words of data.

3.10.1 Integration

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.

3.11 Clock Recovery

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 XS1.

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

The clock recovery thread (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 thread simply generates this clock using a local timer, based on the XS1 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 thread. 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 thread gets clock selection Get/Set commands from Endpoint 0 via the `c_clk_ctl` channel. This thread also records the validity of external clocks, which is also queried through the same channel from Endpoint 0.

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

3.12 MIDI




The MIDI driver implements a 31250 baud UART input and output. On receiving 32-bit USB MIDI events from the decoupler, 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 and passed on to the decoupler. The MIDI thread is implemented in the file `usb_midi.xc`.

3.13 Resource Usage

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

Component	Threads	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 3.6
S/PDIF Tx	1	3.5 (2 code)	1 x 1 bit port
S/PDIF Rx	1	3.7 (3.7 code)	1 x 1 bit port
ADAT Rx	1	3.2 (3.2 code)	1 x 1 bit port
Midi	1	6.5 (1.5 code)	2 x 1 bit ports
Mixer	2	8.7 (6.5 code)	
ClockGen	1	2.5 (2.4 code)	

Table 18: Resource Usage

-  These resource estimates are based on the L2 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 thread to function correctly (i.e. on a 500MHz parts only six threads can run).
-  The ULPI ports are a fixed set of ports on the XS1-L1 or XS1-L2 device. When using these ports, other ports are unavailable when ULPI is active. See [\[XS1HwCL\]](#) for further details.

4 Programming Guide

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.

4.1 Getting Started

4.1.1 Building with the XDE

To install the software, open the XDE (Xilinx Development Tools) 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 `app_usb_aud_11` or `app_usb_aud_12` project in the Project Explorer and click the **Build** icon.

4.1.2 Building from the command line

To install, unzip the package zip.

To build, change into the `app_usb_aud_11` or `app_usb_aud_12` directory and execute the command:

```
xmake all
```

4.1.3 Makefiles

The main Makefile for the project is in the `app_usb_aud_11` or `app_usb_aud_12` directory. This file specifies build options and used modules. The Makefile uses the com-

mon build infrastructure in `module_xmos_common`. This system includes the source files from the relevant modules and is documented within `module_xmos_common`.

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.

4.1.4 Using the XMOS Development Environment

To upgrade the flash from the XDE, follow these steps:

1. Start the XMOS Development Environment 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* ► *Run Configurations*.
7. Double-click *Flash Programmer* to create a new configuration.
8. Browse for the XE file in the *Project* and *C/XC Application* boxes.
9. Ensure the XTAG-2 device appears in the adapter list.
10. Click **Run**.

4.1.5 Using Command Line Tools

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

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

4.2 Code Structure

4.2.1 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:

<code>module_xud</code>	Low level USB library
<code>module_usb_shared</code>	Common code for USB applications
<code>module_usb_aud_shared</code>	Common code for USB audio applications
<code>module_spdif_tx</code>	S/PDIF transmit code
<code>module_spdif_rx</code>	S/PDIF receive code
<code>module_adat_rx_v3</code>	ADAT receive code
<code>module_usb_midi</code>	MIDI I/O code
<code>module_dfu</code>	Device Firmware Upgrade code
<code>module_xmos_common</code>	Common build infrastructure

Table 19: USB Audio Modules

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

<code>app_usb_aud_l1</code>	USB Audio 2.0 Reference Design code
<code>app_usb_aud_l2</code>	USB Audio 2.0 Multichannel Reference Design code

Table 20: USB Audio Reference Applications

4.3 A USB Audio Application (walkthrough)

This tutorial provides a walk through of the USB Audio Reference Design (XS-L1) example, which can be found in the `app_usb_aud_l1` directory. The application uses the USB audio framework modules and provides on top:

1. A `.xnl` file to describe the hardware.

2. A custom defines file: `customdefines.h`
3. Source code providing definitions of application support functions for the USB audio framework.
4. The application itself; in particular a top-level main function describing which components are run where on the device.

4.3.1 Custom Defines

The `customdefines.h` file contains all the `#defines` required to tailor the USB audio framework to the particular application at hand. First there are defines to determine overall capability. For this reference design input and output, S/PDIF output and DFU are enabled:

```
/* Enable Fall-back to audio class 1.0 a FS */
#define AUDIO_CLASS (2)

/* Enable Fall-back to audio class 1.0 when connected to FS hub */
#define AUDIO_CLASS_FALLBACK 1

/* Enable audio input */
#define INPUT 1

/* Enable audio output */
#define OUTPUT 1

/* Enable S/PDIF output */
#define SPDIF 1

/* Run the CODEC as slave, Xcore as master */
#define CODEC_SLAVE 1

/* Enable DFU interface, Note, requires a driver for Windows */
#define DFU 1

/* Disable MIDI */
#define MIDI 0
```

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:

```
/* Number of USB streaming channels */
#define NUM_USB_CHAN_IN    (2)        /* Host to device (D/A) */
#define NUM_USB_CHAN_OUT  (2)        /* Device to host (A/D) */

/* Number of I2S chans to DAC..*/
#define I2S_CHANS_DAC      (2)

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

/* 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 */
#define MAX_FREQ           (192000)

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

/* Default frequency device reports as running at */
/* Audio Class 1.0 friendly freq */
#define DEFAULT_FREQ      (48000)
```

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 */
#define BCD_DEVICE   (0x0310) /* Device release number in BCD: 0xJJMN
                               * JJ: Major, M: Minor, N: Sub-minor */
```

For a full description of all the defines that can be set in `customdefines.h` see Section [5.1](#).

4.3.2 Configuration Functions

In addition to the custom defines file, the application needs to provide definitions of user functions that are specific to the application. Firstly, code is required to handle the CODEC. On boot up you do not need to do anything with the CODEC so there is just an empty function:

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

On sample rate changes, you need to reset the CODEC 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 toggled through the `port32A_peek` and `port32A_out` functions:


```
void CodecConfig(unsigned samFreq, unsigned mClk, chanend ?c_codec)
{
    timer t;
    unsigned time;
    unsigned portVal;
    unsigned tmp;

    /* Put codec in reset and set master clock select appropriately */
    if ((samFreq % 22050) == 0)
    {
        portVal = P32A_USB_RST;
    }
    else if((samFreq % 24000) == 0)
    {
        portVal = (P32A_USB_RST | P32A_CLK_SEL);
    }
    else
    {
        if (samFreq == 1234)
            return;
        printintln(samFreq);
        printstr("Unrecognised sample freq in ConfigCodec\n");
        assert(0);
    }

    tmp = port32A_peek();
    tmp &= (P32A_LED_A | P32A_LED_B);
    port32A_out(portVal | tmp);

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

    /* Codec out of reset */
    portVal |= P32A_COD_RST;
    tmp = port32A_peek();
    tmp &= (P32A_LED_A | P32A_LED_B);
    port32A_out(portVal | tmp);
}
```

Since the clocks come from fixed oscillators on this board, the clock configuration functions do not need to do anything. This will be different if the clocks came from

an external PLL chip:

```
/* These functions must be implemented for the clocking
   arrangement of specific design */

void ClockingInit()
{
    /* For L1 reference */
}

void ClockingConfig(unsigned mClkFreq)
{
    /* For L1 reference */
}
```

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

```
#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)
 */
void AudioStreamStart(void) {
    int x;
    x = port32A_peek();
    x |= P32A_LED_B;
    port32A_out(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 AudioStreamStop(void) {
    int x;
    x = port32A_peek();
    x &= (~P32A_LED_B);
    port32A_out(x);
}
```

4.3.3 The main program

The main program can be found in `main.xc`. It contains:

- A declaration of all the ports used in the design. 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 threads in parallel.

This is a single core program so all threads are on the same core. For a multicore program different threads can be placed on different cores using the `on stdcore[n]:` syntax.

The first thread run is the XUD driver:

```
#if (AUDIO_CLASS==2)
    XUD_Manager(c_xud_out, NUM_EP_OUT, c_xud_in, NUM_EP_IN,
               c_sof, epTypeTableOut, epTypeTableIn, p_usb_rst,
               clk, 1, XUD_SPEED_HS, null);
#else
    XUD_Manager(c_xud_out, NUM_EP_OUT, c_xud_in, NUM_EP_IN,
               c_sof, epTypeTableOut, epTypeTableIn, p_usb_rst,
               clk, 1, XUD_SPEED_FS, null);
#endif
```

The make up of the channel arrays connecting to this driver are described in Section 5.3.

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

```
{
    thread_speed();
    Endpoint0( c_xud_out[0], c_xud_in[0], c_aud_ctl, null, null);
}
```

```
{
    thread_speed();
    buffer(c_xud_out[1], c_xud_in[2], c_xud_in[1],
          c_xud_out[2], c_xud_in[3], c_xud_in[4],
          c_sof, c_aud_ctl, p_for_mclk_count);
}
```

```
{
    thread_speed();
    decouple(c_mix_out,
            c_midi,
            null);
}
```

```
{
    thread_speed();
}
```

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 threads as opposed to one.

```
    audio(c_mix_out, null, null);  
}
```

Finally, if MIDI is enabled you need a thread to drive the MIDI input and output:

```
    thread_speed();  
    usb_midi(p_midi_rx, p_midi_tx, clk_midi, c_midi, 0);
```

4.4 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 retest the resulting software.

The general steps are:

1. Make a copy of the eclipse project or application directory (app_usb_aud_11 or app_usb_aud_12) 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.

4.4.1 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 thread.

4.4.2 Example: Adding DSP to output stream

To add some DSP requires an extra thread 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 Section *custom_defines_api*.
2. Add another thread to do the DSP. This thread will probably have three XC channels: one channel to receive samples from decoupler thread and another to output to the audio driver—this way the thread ‘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 thread. This needs to be synchronous (i.e. for every sample received from the decoupler, a sample needs to be outputted to the audio driver).
4. Update the Endpoint 0 code to accept custom requests to the audio class interface to control the DSP. It can then forward the changes onto the DSP thread.
5. Update host drivers to use these custom requests.

5 API

5.1 Custom Defines

An application using the USB audio framework needs to have a defines file called `customedefines.h`. This file can set the following defines:

5.1.1 System Feature Configuration

Define	Description	Default
INPUT	Define for enabling audio input, in descriptors, buffering and so on.	defined
DFU	Define to enable DFU interface. Requires a custom driver for Windows.	defined
DFU_CUSTOM_FLASH_DEVICE	Define to enable use of custom flash device for DFU interface.	not defined
MIDI	Define to enable MIDI input and output.	defined
CODEC_SLAVE	If defined the CODEC acts as I2S slave (and the XCore as master) otherwise the CODEC acts as master.	defined
NUM_USB_CHAN_IN	Number of audio channels the USB audio interface has from host to the device.	10
NUM_USB_CHAN_OUT	Number of audio channels the USB audio interface has from device to host.	10
MAX_FREQ	Maximum frequency device runs at in Hz	96000
I2S_CHANS_DAC	Number of I2S audio channels output to the codec. This must be a multiple of 2.	8
I2S_CHANS_ADC	Number of I2S audio channels input from the codec. This must be a multiple of 2.	8
SPDIF	Define to Enable S/PDIF output. If OUTPUT is not defined, zero-ed samples are emitted. The S/PDIF audio channels will be two channels immediately following I2S_CHANS_DAC.	defined
SPDIF_RX	Define to enable S/PDIF input.	not defined
ADAT_RX	Define to enable ADAT input.	not defined
MIXER	Define to enable the MIXER.	not defined

Define	Description	Default
MIN_VOLUME	The minimum volume setting above -inf. This is a signed 8.8 fixed point number that must be strictly greater than -128 (0x8000).	0x8100
MAX_VOLUME	The maximum volume setting for the mixer in db. This is a signed 8.8 fixed point number.	0
VOLUME_RES	The resolution of the volume control in db as a 8.8 fixed point number.	0x100
MIN_MIXER_VOLUME	The minimum volume setting for the mixer unit above -inf. This is a signed 8.8 fixed point number that must be strictly greater than -128 (0x8000).	0x8080
MAX_MIXER_VOLUME	The maximum volume setting for the mixer. This is a signed 8.8 fixed point number.	0x0600
VOLUME_RES_MIXER	The resolution of the volume control in db as a 8.8 fixed point number.	0x080

5.1.2 USB Device Configuration Options

Define	Description	Default
VENDOR_ID	Vendor ID	(0x20B1)
PID_AUDIO_2	Product ID (Audio Class 2)	N/A
PID_AUDIO_1	Product ID (Audio Class 1)	N/A
BCD_DEVICE	Device release number in BCD form	N/A
VENDOR_STR	String identifying vendor	XMOS
SERIAL_STR	String identifying serial number	"0000"

5.2 Required User Function Definitions

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

5.2.1 Codec Configuration Functions

void **CodecInit**(chanend ?*c_codec*)

This function is called when the audio thread starts after the device boots up and should initialize the CODEC.

Parameters

- **c_codec** – An optional chanend that was original passed into [audio\(\)](#) that can be used to communicate with other threads.

void **CodecConfig**(unsigned *samFreq*,
 unsigned *mclk*,
 chanend ?*c_codec*)

This function is called when the audio thread starts or changes sample rate. It should configure the CODEC to run at the specified sample rate given the supplied master clock frequency.

Parameters

- **samFreq** – The sample frequency in Hz that the CODEC should be configured to play.
- **mclk** – The master clock frequency that will be supplied to the CODEC in Hz.
- **c_codec** – An optional chanend that was original passed into [audio\(\)](#) that can be used to communicate with other threads.

5.2.2 Clocking Configuration Functions

void **ClockingInit**(void)

This function is called when the audio thread starts are device boot. It should initialize any external clocking hardware.

void **ClockingConfig**(unsigned *mClkFreq*)

This function is called when the audio thread starts or changes sample frequency. It should configure any external clocking hardware such that the master clock signal being fed into the XCore and CODEC is the same as the specified frequency.

Parameters

- **mClkFreq** – The required clock frequency in Hz.

5.2.3 Audio Streaming Functions

void **AudioStreamStart**(void)

This function is called when the audio stream from device to host starts.

void **AudioStreamStop**(void)

This function is called when the audio stream from device to host stops.

5.3 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](#).

```
int XUD_Manager(chanend c_ep_out[],
               int noEpOut,
               chanend c_ep_in[],
               int noEpIn,
               chanend ?c_sof,
               XUD_EpType epTypeTableOut[],
               XUD_EpType epTypeTableIn[],
               out port p_usb_rst,
               clock clk,
               unsigned rstMask,
               unsigned desiredSpeed,
               chanend ?c_usb_testmode)
```

This performs the low level USB I/O operations.

Note that this needs to run in a thread with at least 80 MIPS worst case execution speed.

Parameters

- **c_ep_out** – An array of channel ends, one channel end per output endpoint (USB OUT transaction); this includes a channel to obtain requests on Endpoint 0.
- **num_out** – The number of output endpoints, should be at least 1 (for Endpoint 0).
- **c_ep_in** – 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.
- **num_in** – The number of output 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 block up. If no SOF tokens are required `null` should be used as this channel.

- **ep_type_table_out** – See `ep_type_table_in`
- **ep_type_table_in** – This and `ep_type_table_out` are two arrays indicating the type of channel ends. Legal types include: `XUD_EPTYPE_CTL` (Endpoint 0), `XUD_EPTYPE_BUL` (Bulk endpoint), `XUD_EPTYPE_ISO` (Isochronous 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 send reset signals to.
- **clk** – The clock block to use for the USB reset - this should not be clock block 0.
- **reset_mask** – The mask to use when sending a reset. The mask is ORed into the port to enable reset, and unset when deasserting reset. Use '-1' as a default mask if this port is not shared.
- **desired_speed** – This parameter specifies whether the device must be full-speed (ie, USB-1.0) or whether high-speed is acceptable if supported by the host (ie, USB-2.0). Pass `XUD_SPEED_HS` if high-speed is allowed, and `XUD_SPEED_FS` if not. Low speed USB is not supported by XUD.
- **test_mode** – This should always be null.

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

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

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

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

```
void Endpoint0(chanend c_ep0_out,  
              chanend c_ep0_in,  
              chanend c_audioCtrl,  
              chanend ?c_mix_ctl,  
              chanend ?c_clk_ctl)
```

Function implementing Endpoint 0 for enumeration, control and configuration of USB audio devices.

It uses the descriptors defined in `descriptors_2.h`.

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.

```
void buffer(chanend c_aud_out,  
           chanend c_aud_in,  
           chanend c_aud_fb,  
           chanend c_midi_from_host,  
           chanend c_midi_to_host,  
           chanend c_int,  
           chanend c_sof,  
           chanend c_aud_ctl,  
           in port p_off_mclk)
```

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\(\)](#)

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_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)

```
void decouple(chanend c_audio_out,  
              chanend ?c_led,  
              chanend ?c_midi,  
              chanend ?c_clk_int)
```

Manage the data transfer between the USB audio buffer and the Audio I/O driver.

Parameters

- **c_audio_out** – Channel connected to the [audio\(\)](#) or [mixer\(\)](#) threads
- **c_led** – Optional chanend connected to an led driver thread for debugging purposes
- **c_midi** – Optional chanend connect to [usb_midi\(\)](#) thread if present
- **c_clk_int** – Optional chanend connected to the [clockGen\(\)](#) thread if present

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

Digital sample mixer.

This thread mixes audio streams between the [decouple\(\)](#) thread and the [audio\(\)](#) thread.

Parameters

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

```
void audio(chanend c_in,  
          chanend ?c_dig,  
          chanend ?c_config)
```

The audio driver thread.

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

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.

```
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)
```

Clock generation and digital audio I/O handling.

Parameters

- **c_spdif_rx** – channel connected to S/PDIF receive thread
- **c_adat_rx** – channel connect to ADAT receive thread
- **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

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

S/PDIF receiver.

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 kHz.w

For a 100MHz reference clock, use a divider of 1 for 192000, 2 for 96000/88200, 4 for 48000/44100 on clock b. 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: whole word with bits 0-3 set to preamble.

Parameters

- **p** – S/PDIF output port
- **c** – channel to output samples to
- **initial_divider** – initial divide for initial estimate of sample rate

- **b** – clock block set to 100MHz

void **adatReceiver48000**(buffered in port:32 *p*,
chanend *oChan*)
ADAT Receive Thread (48kHz sample rate).

Parameters

- **p** – ADAT port - should be 1-bit and clocked at 100MHz
- **oChan** – channel on which decoded samples are output

The function will return if it cannot lock onto a 44,100/48,000 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 it does not need to be supported.

void **adatReceiver44100**(buffered in port:32 *p*,
chanend *oChan*)
ADAT Receive Thread (44.1kHz sample rate).

Parameters

- **p** – ADAT port - should be 1-bit and clocked at 100MHz
- **oChan** – channel on which decoded samples are output

The function will return if it cannot lock onto a 44,100/48,000 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 it does not need to be supported.

void **usb_midi**(in port *?p_midi_in*,
out port *?p_midi_out*,
clock *?clk_midi*,
chanend *c_midi*,
unsigned *cable_number*)

USB MIDI I/O thread.

This function passes MIDI data from USB to UART I/O.

Parameters

- **p_midi_in** – 1-bit input port for MIDI
- **p_midi_out** – 1-bit output port for MIDI
- **clk_midi** – clock block used for clockin the UART; should have a rate of 100MHz
- **c_midi** – chanend connected to the [decouple\(\)](#) thread
- **cable_number** – the cable number of the MIDI implementation. This should be set to 0.

6 References

[XC09] Douglas Watt. Programming in XC on XMOS Devices. Xmos Ltd, 2009.

http://www.xmos.com/published/xc_en

[ToolsUserGuide] Douglas Watt and Huw Geddes. The XMOS Tools User Guide. Xmos Ltd. 2010.

http://www.xmos.com/published/xtools_en

[USBAud10] Device Class Definition for Audio Devices (v1.0). 1998.

http://www.usb.org/developers/devclass_docs/audio10.pdf

[USBAud20] Device Class Definition for Audio Devices (v2.0). 2006.

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

[DFU11] Device Class Specification for Device Firmware Upgrade (v1.1). 2004.

http://www.usb.org/developers/devclass_docs/DFU_1.1.pdf

[USBMidi10] Universal Serial Bus Device Class Definition for MIDI Devices (v1.0).

1999. http://www.usb.org/developers/devclass_docs/midi10.pdf

[XS1HwCL] XS1-L Hardware Design Checklist. XMOS Ltd. 2010.

<http://www.xmos.com/published/xs1lcheck>

[Xud] XMOS USB Device (XUD) Layer Library. 2010.

<http://www.xmos.com/published/xuddg>



Copyright © 2011 XMOS Limited, All Rights Reserved.

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

XMOS and the XMOS logo are registered trademarks of XMOS Limited in the United Kingdom and other countries, and may not be used without written permission. All other trademarks are property of their respective owners. Where those designations appear in this book, and XMOS was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.
