USB Audio User Guide

Publication Date: 2023/3/16
Document Number: XM008854A

Document Number: XM008854A

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 xcore-200 (XS2) and xcore.ai (XS3) architectures, it supports USB Audio Class 2.0 and USB Audio Class 1.0, asynchronous mode (synchronous as an option) and sample rates up to 384kHz.

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

The XMOS USB Audio solution is deployed as a framework (see `lib_xua`) 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 user 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 (lib_xud)[2] and the XMOS USB Audio Library (lib_xua)[3]

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

[1] https://www.xmos.com/published/xmos-programming-guide
[2] https://github.com/xmos/lib_xud/releases/latest
[3] https://github.com/xmos/lib_xua/releases/latest

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 xcore.ai Series
Requirements	
Development Tools	xTIMEcomposer Development Tools v15.1 or later
USB	xCORE device with integrated USB phy (external phy not supported)
Audio	External audio DAC/ADC/CODECs (and required supporting componentry) supporting I2S/TDM
Boot/Storage	Compatible SPI/QSPI Flash device (or xCORE device with internal flash)
Licensing and Support	
Reference code provided without charge under license from XMOS. Please visit `http://www.xmos.com/support/contact` for support. Reference code is maintained by XMOS Limited.	

[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

This section describes the hardware development platforms supported by the XMOS USB Audio reference design software.

2.1 xcore.ai Multi-Channel Audio Board

The *XMOS xcore.ai Multichannel Audio Board* (XK-AUDIO-316-MC) is a complete hardware and software reference platform targeted at up to 32-channel USB audio applications, such as DJ decks, mixers and other musical instrument interfaces. The board can also be used to prototype products with reduced feature sets or HiFi style products.

The XK-AUDIO-316-MC is based around the XU316-1024-TQ128-C24 multicore microcontroller; a dual-tile xcore.ai device with an integrated High Speed USB 2.0 PHY and 16 logical cores delivering up to 2400MIPS of deterministic and responsive processing power.

Exploiting the flexible programmability of the xcore.ai architecture, the XK-AUDIO-316-MC supports a USB audio source, streaming 8 analogue input and 8 analogue output audio channels simultaneously - at up to 192kHz. It also supports digital input/output streams (S/PDIF and ADAT) and MIDI. Ideal for consumer and professional USB audio interfaces. The board can also be used for testing general purpose audio DSP activities - mixing, filtering, etc.

The guaranteed Hardware-ResponseTM times of xCORE technology always ensure lowest latency (round trip as low as 3ms), bit perfect audio streaming to and from the USB host

For full details regarding the hardware please refer to xcore.ai Multichannel Audio Platform Hardware Manual.

The XK-AUDIO-316-MC reference hardware has an associated firmware application that uses *lib_xua* to implement fully-featured and production ready USB Audio solution. Full details of this application can be found later in this document.

2.1.1 Hardware Features

The location of the various featurs of the xcore.ai Multichannel Audio Board (XK-AUDIO-316-MC) is shown in Figure 1.

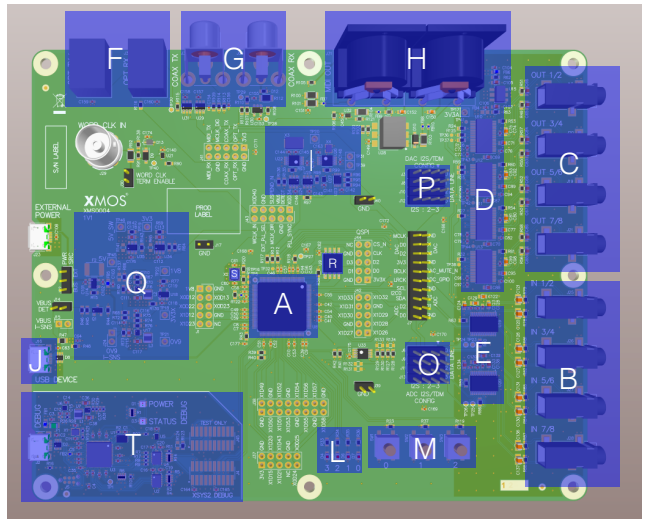

Figure 1: xcore.ai Multichannel Audio Board block diagram

It includes the following features:

- A: xcore.ai (XU316-1024-TQ128-C24) multicore microcontroller device
- B: 8 line level analog inputs (3.5mm stereo jacks)
- C: 8 line level analog outputs (3.5mm stereo jacks)
- D: 384kHz 24 bit audio DACs
- E: 192kHz 24 bit audio ADCs
- F: Optical connections for digital interface (e.g. S/PDIF and ADAT)
- G: Coaxial connections for digital interfaces (e.g. S/PDIF)
- H: MIDI in and out connections
- I: Flexible audio master clock generation
- J: USB 2.0 micro-B jacks
- L: 4 general purpose LEDs
- M: 3 general purpose buttons
- O: Flexible I2S/TDM input data routing

- P: Flexible I2S/TDM output data routing
- Q: Integrated power supply
- R: Quad-SPI boot ROM
- S: 24MHz Crystal
- T: Integrated XTAG4 debugger

2.1.2 Analogue Input & Output

A total of eight single-ended analog input channels are provided via 3.5mm stereo jacks. These inputs feed into a pair of quad-channel PCM1865 ADCs from Texas Instruments.

A total of eight single-ended analog output channels are provided. These are fed from four PCM5122 stereo DAC's from Texas instruments.

All ADC's and DAC's are configured via an I2C bus. Due to an clash of device addresses a I2C mux is used.

The four digital I2S/TDM input and output channels are mapped to the xCORE input/outputs through a header array. These jumpers allow channel selection when the ADCs/DACs are used in TDM mode.

2.1.3 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 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.

2.1.4 MIDI

MIDI input and output is provided on the board via standard 5-pin DIN connectors compliant to the MIDI specification. The signals are buffered using 5V line drivers and are then connected ports on the xCORE, via a 5V to 3.3V buffer. A 1-bit port is used for receive and a 4-bit port is used for transmit. A pullup resistor on the MIDI output ensures there is no MIDI output when the xCORE device is not actively driving the output.

2.1.5 Audio Clocking

In order to accommodate a multitude of clocking options a flexible clocking scheme is provided for the audio subsystem.

Three methods of generating an audio master clock are provided on the board:

- A Cirrus Logic CS2100-CP PLL device. The CS2100 features both a clock generator and clock multiplier/jitter reduced clock frequency synthesizer (clean up) and can generate a low jitter audio clock based on a synchronisation signal provided by the xCORE

- A Skyworks Si5351B PLL device. The Si5351 is an I2C configurable clock generator that is suited for replacing crystals, crystal oscillators, VCXOs, phase-locked loops (PLLs), and fanout buffers.
- xcore.ai devices are equipped with a secondary (or *application*) PLL which can be used to generate audio clocks.

Selecting between these methods is done via writing to bits 6 and 7 of PORT 8D on tile[0]. See §2.1.6.

The supplied software currently supports the xcore.ai secondary PLL or CS2100 device.

2.1.6 Control I/O

4 bits of PORT 8C are used to control external hardware on the board. This is described in Figure 2.

Figure 2: PORT 8C functionality

Bit(s)	Functionality	0	1
[0:3]	Unused		
4	Enable 3v3 power for digital (inverted)	Enabled	Disabled
5	Enable 3v3 power for analogue	Disabled	Enabled
6	PLL Select	CS2100	Si5351B
7	Master clock direction	Output	Input

To use the xCORE application PLL bit 7 should be set to 0. To use one of the external PLL's bit 7 should be set to 1.

2.1.7 LEDs, Buttons and Other IO

All programmable I/O on the board is configured for 3.3 volts.

Four green LED's and three push buttons are provided for general purpose user interfacing.

The LEDs are connected to PORT 4F and the buttons are connected to bits [0:2] of PORT 4E, both on tile 0. Bit 3 of this port is connected to the (currently unused) ADC interrupt line.

The board also includes support for an AES11 format Word Clock input via 75 ohm BNC. The software does not currently support any functionality related to this and it is provided for future expansion.

All spare I/O is brought out and made available on 0.1" headers for easy connection of expansion boards etc.

2.1.8 Power

The board is capable of acting as a USB2.0 self or bus powered device. If bus powered, the board takes power from the `USB DEVICE` connector (micro-B receptacle). If self powered, board takes power from `EXTERNAL POWER` input (micro-B receptacle).

A Power Source Select (marked `PWR SRC`) is used to select between bus and self-powered configuration.

To remain USB compliant the software should be properly configured for bus vs self powered operation

2.1.9 Debug

For convenience the board includes an on-board xTAG4 for debugging via JTAG/xSCOPE. This is accessed via the USB (micro-B) receptacle marked `DEBUG`.

2.2 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 XK-AUDIO-216-MC 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 XK-AUDIO-216-MC supports either USB or network audio source, streaming 8 analogue input and 8 analogue output audio channels simultaneously - at up to 192kHz.

For full details regarding the hardware please refer to xCORE-200 Multichannel Audio Platform Hardware Manual[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 §6.2.

2.2.1 Analogue Input & Output

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

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

2.2.2 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.

2.2.3 MIDI

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

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

2.2.4 Audio Clocking

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

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

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

2.2.5 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.

2.3 xCORE.ai Evaluation Kit

The *XMOS xCORE.ai Evaluation Kit* (XK-EVK-XU316) is an evaluation board for the xCORE.ai multi-core microcontroller from *XMOS*.

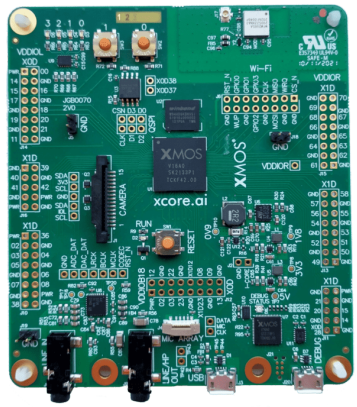

Figure 3: xCORE.ai Evaluation Kit

The XK-EVK-XU316 allows testing in multiple application scenarios and provides a good general software development board for simple tests and demos. The XK-EVK-XU316 comprises an *xCORE.ai* processor with a set of I/O devices and connectors arranged around it, as shown in :Figure 4.

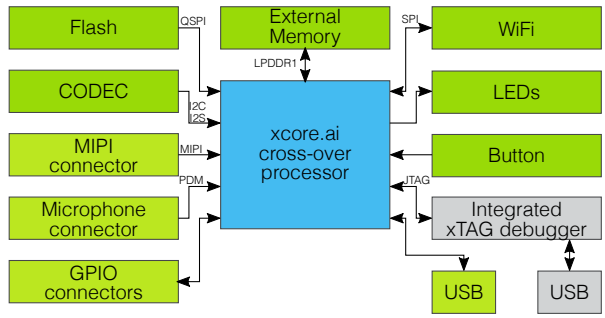

Figure 4: xCORE.ai Evaluation Kit block diagram

External hardware features board include, four general purpose LEDs, two general purpose push-button switches, a PDM microphone connector, audio codec with line-in and line-out jack, QSPI flash memory, LPDDR1 external memory 58 GPIO connections from tile 0 and 1, micro USB for power and host connection, MIPI connector for a MIPI camera, integrated *xTAG* debug adapter and a reset switch with LED to indicate running.

For full details regarding the hardware please refer to XK-EVK-XU316 xCORE.ai Evaluation Kit Manual[10].

The XK-EVK-XU316 hardware has an associated firmware application that uses `lib_xua` to implement an example USB Audio device. Full details of this application can be found later in this document.

The *xCORE.ai Evaluation Kit* is a general purpose evaluation platform and should be considered as an example rather than a fully fledged reference design.

2.3.1 Analogue Audio Input & Output

A stereo CODEC (TLV320AIC3204), connected to the xCORE.ai device via an I2S interface, provides analogue input/output functionality at line level.

The audio CODEC is are configured by the *xCORE.ai* device via an I2C bus.

2.3.2 Audio Clocking

xCORE.ai devices are equipped with a secondary (or *application*) PLL which is used to generate the audio clocks for the CODEC.

2.3.3 LEDs, Buttons and Other IO

Four green LED's and two push buttons are provided for general purpose user interfacing.

The LEDs are connected to PORT 4C and the buttons are connected to bits [0:1] of PORT 4D.

All spare I/O is brought out and made available on 0.1" headers for easy connection of expansion boards etc.

2.3.4 Power

The XK-EVK-XU316 requires a 5V power source that is normally provided through the micro-USB cable J3. The voltage is converted by on-board regulators to the 0V9, 1V8 and 3V3 supplies used by the components.

The board should therefore be configured to present itself as a bus powered device when connected to an active USB host.

2.3.5 Debug

For convenience the board includes an on-board xTAG4 for debugging via JTAG/xSCOPE. This is accessed via the USB (micro-B) receptacle marked **DEBUG**.

[10] https://www.xmos.ai/download/xcore.ai-explorer-board-v2.0-hardware-manual(5).pdf

3 Driver Support

IN THIS CHAPTER

The XMOS USB Audio Reference design includes support for USB Audio Class (UAC) versions 1.0 and 2.0. UAC 2.0 includes support for audio over high-speed USB (UAC 1.0 supports full-speed only) and other feature additions.

3.1 OS Support for UAC 1.0

Support for USB Audio Class 1.0 has been included in macOS and Windows for a number of years. Most Linux distributions also include support.

3.2 OS Support for UAC 2.0

Support for USB Audio Class 2.0 is only included in more modern versions of macOS and Windows:

- Since version 10.6.4 macOS natively supports USB Audio Class 2.0
- Since version 10, release 1809, Windows natively supports USB Audio Class 2.0

3.3 Thirds Party Windows Drivers

For some products it may be desirable to use a third-party driver for Windows. A number reasons exist as to why this may be desirable:

- In order to support UAC 2.0 on Windows versions earlier than 10
- The built-in Windows support is typically designed for consumer audio devices, not for professional audio devices
- The built in drivers support sound APIs such as WASAPI, DirectSound, MME, but not ASIO.

The XMOS USB Audio Reference design is tested against *Thesycon USB Audio Driver for Windows*. This includes the following feature-set/benefits:

- Available for Windows 10 and Windows 11 operating systems
- Designed for professional audio devices and consumer-style devices
- Supports ASIO for transparent and low-latency audio streaming
- Supports Windows sound APIs such as WASAPI, DirectSound, MME

- Supports high-end audio features such as bit-perfect PCM up to 768 kHz sampling rate, native DSD format (through ASIO) up to DSD1024
- Supports multiple clock sources such as S/PDIF, ADAT or WCLK inputs
- Supports MIDI 1.0 class, including MIDI port sharing
- Supports DFU (Device Firmware Upgrade) and comes with a GUI utility for firmware update
- Provides a private API for driver control and direct device communication (SDK available)
- Comes with a control panel application for driver status/control
- Optionally supports virtual channels (channels available at ASIO and Windows APIs but not implemented in the device)
- Optionally supports mixing and/or signal processing plugin in the kernel-mode driver
- Fully supports driver signing, branding and customization including driver installer (Customization will be done by Thesycon)
- Technical support and maintenance provided by Thesycon
- Custom features available on request

Many of the benefits listed above apply to both UAC1.0 and UAC2.0 and the Thesycon Driver supports both class versions. It should be noted, however, that XMOS only currently tests UAC1 with built-in drivers.

4 Quick Start

IN THIS CHAPTER

XMOS development boards are typically supplied with no firmware installed. The following steps explain how to install the latest firmware on the board and use it. Each step is explained in detail in the following sections.

1. Download the latest **USB Audio 2.0 Device Software** release from **http://xmos.com** ▶ **Applications** ▶ **USB & Multi-Channel Audio** and follow the *USB AUDIO SOFTWARE* link. Before you download the software review the licence and click **Accept** to initiate the download.

   (Section §4.1.)

2. If using a Windows host computer, download the **USB Audio Class 2.0 Evaluation Driver for Windows** from **http://xmos.com** ▶ **Applications** ▶ **USB & Multi-Channel Audio** and follow the *DRIVER SUPPORT* link and click on *Download*. Once downloaded, run the executable and install the driver.

   (Section §4.2.)

3. Download the **XMOS XTC Tools** from: `http://www.xmos.com/software-tools` and install.

   The firmware should be compiled using a specific version of the tools. Make sure that you download the correct version of the tools.

   (Section §4.3)

4. Compile the firmware relavant to the board you have .

   (Section: §4.4)

5. Connect the board to your development system using the xTAG supplied, and program the firmware onto the board.

   (Section §4.5)

6. Connect audio input and output devices, and play your audio.

(Section §4.7)

4.1 USB Audio 2.0 Reference Software

The latest USB Audio 2.0 Reference Design software is available free of charge from XMOS.

The first time you download the software you need to register at:

`http://www.xmos.com/`

To download the firmware:

1. Go to **http://xmos.com** ▶ **Applications** ▶ **USB & Multi-Channel Audio** and follow the *USB AUDIO SOFTWARE* link.
2. Review the licence agreement and click **Accept**.
3. Download and save the software when prompted.

The software is distributed as a zip archive containing pre-compiled binaries and source code that can be built using the *XMOS XTC Tools*.

Alternatively you can contact your local sales representative for further details:

`https://www.xmos.com/find-a-distributor/`

4.2 USB Audio Class 2.0 Evaluation Driver for Windows

Since version 10.6.4, macOS natively supports USB Audio Class 2.0 – no driver install is required.

Since version 10, release 1703, Windows natively supports USB Audio Class 2.0 – no driver install is required.

Earlier Window versions only provides support for USB Audio Class 1.0. To use a USB Audio Class 2.0 device under these Windows versions requires a third party driver.

Developers may also wish to use a third party driver for reasons including:

- ASIO support
- Advanced clocking options and controls
- Improved latency
- Native DSD (via ASIO)
- Branding customisation and custom control panels
- Large channel count devices
- Etc

XMOS therefore provides a free Windows USB Audio driver for evaluation and prototyping and a path to a more feature-rich multichannel production driver from our partner *Thesycon*.

The evaluation driver is available from:

http://www.xmos.com/published/usb-audio-class-20-evaluation-driver-windows

Further information about the evaluation and production drivers is available in the *USB Audio Class 2.0 Windows Driver Overview* document available from:

http://www.xmos.com/published/usb-audio-20-stereo-driver-windows-overview

4.3 XMOS XTC Development Tools

The *XMOS XTC Tools* provide everything you need to develop applications for *xcore multicore microcontrollers* and can be downloaded, free of charge, from: http://www.xmos.com/software-tools.

The *XMOS XTC Tools* make it easy to define real-time tasks as a parallel system. They come with standards compliant C and C++ compilers, language libraries, simulator, symbolic debugger, and runtime instrumentation and trace libraries. Multicore support offers features for task based parallelism and communication, accurate timing and I/O, and safe memory management. All components work off the real-time multicore functionality, giving a fully integrated approach.

The XTC tools are required by anyone developing or deploying applications on an *xcore* processor. The tools include:

- "Tile-level" toolchain (Compiler, assembler, etc)
- System libraries
- "Network-level" tools (Multi-tile mapper etc)
- XSIM simulator
- XGDB debugger
- Deployment tools

The tools as delivered are to be used within a command line environment, though may also be integrated with your preferred IDE[11].

The firmware must be compiled using a specific version of the tools. Make sure that you download the correct version of the tools. Older versions of tools are available from the *TOOLS ARCHIVE* section of http://www.xmos.com/software-tools

Information on using the tools, including installation, is provided in the XTC Tools Guide[12].

[11] https://www.xmos.ai/documentation/XM-014363-PC-7/html/tools-guide/install-configure/config-ide/index.html
[12] https://www.xmos.ai/documentation/XM-014363-PC-7/html/intro.html

4.4 Building the Firmware

For convenience the release zips provided from XMOS contain precompiled binary (xe) files.

From a command prompt with the XMOS tools available, follow these steps:

1. Unzip the package zip to a known location
2. Move into the relevant application directory (e.g. `app_usb_aud_xk_audio_316_mc`) and execute the command:

```
xmake all
```

The proceeding steps will build all of the available and supported build configurations for the application.

The main Makefile for the project is in the application directory (e.g. `app_usb_aud_xk_audio_316_mc`). This file specifies build options and dependencies.

This Makefile uses the common build infrastructure supplied with XMOS tools in `module_xmos_common`. This system includes the source files from the relevant modules and is documented within `module_xmos_common`. See :§5.1.1.

4.5 Running the Firmware

Typically during development the developer wishes to program the device's internal RAM directly via JTAG and run then execute this program.

To run one of the compiled binaries complete the following steps:

1. Connect the USB Audio board to your host computer.
2. Connect the xTAG to the USB Audio board and connect the it to your PC or Mac via a separate USB cable
3. Ensure any required external power jacks are connected

Finally, to run the binary on the target, execute a command similar to the following:

```
xrun path/to/binary.xe
```

The device should now present itself as a USB Audio Device on the connected host computer. It will continue to operate as a USB Audio Device until the target board is power cycled.

4.6 Writing the Application Binary to Flash

Optionally a binary can be programmed into the boot flash. To do this:

1. Connect the USB Audio board to your host computer.

2. Connect the xTAG to the USB Audio board and connect the it to your PC or Mac via a separate USB cable
3. Ensure any required external power jacks are connected

From a command prompt with the XMOS tools available, run the following command:

```
xflash path/to/binary.xe
```

Once flashed the target device will reboot and execute the binary. Power cycling the target board will cause the device to reboot the flashed binary.

If subsequently you wish to use `xrun` to program the device it is always advisable to erase the flash contents using the `erase-all` option to the `xflash` tool.

4.7 Playing Audio

1. Connect the board to any power supply provided (note, some boards will be USB bus powered)
2. Connect the board a host with driver support the USB Audio Class using a USB cable
3. Install the Windows USB Audio 2.0 demonstration driver, if required.
4. Connect your audio input/output devices to the connectors on the board e.g powered speakers
5. In your audio application, select the XMOS USB Audio device.
6. Start playing and recording.

4.8 Next Steps

Further information on using the board and the *XTC Tools* is available from:

xcore-200 Multichannel Audio Platform 2v0 Hardware Manual

https://www.xmos.ai/file/xcore-200-multichannel-audio-platform-hardware-manuals?vers

xcore.ai Multichannel Audio Platform 2v0 Hardware Manual

https://www.xmos.ai/file/xcore-ai-multichannel-audio-platform-hardware-manuals?vers

XMOS USB Device Library (lib_xud)

https://github.com/xmos/lib_xud/releases/latest

XMOS USB Audio Library (lib_xua)

https://github.com/xmos/lib_xua/releases/latest

XTC Tools User Guide

https://www.xmos.ai/view/Tools-15---Documentation

5 Programming Guide

IN THIS CHAPTER

The following sections provide a guide on how to program the USB Audio applications including information on project structure, build configurations and creating your own custom USB audio applications.

5.1 Project Structure

5.1.1 Build System

The *XMOS USB Audio Reference Design* software and associated libraries employ the *XMOS XCOMMON* build system. The *XCOMMON* build system is built on top of the GNU Makefile build system. The *XCOMMON* build system accelerates the development of xCORE applications. Instead of having to express dependencies explicitly in Makefiles, users should follow a particular folder structures and naming convention, from which dependencies are inferred automatically.

The *XCOMMON* build system depends on use of of the tool XMAKE[13] specifically. It cannot currently be used with a generic port of GNU Make.

5.1.2 Applications and Libraries

The `sw_usb_audio` GIT[14] repository includes multiple application directories that in turn contain Makefiles that build into executables. Typically you can expect to see one application directory per hardware platform. Applications and there respective hardware platforms are listed in Figure 5.

Figure 5: USB Audio Reference Applications

Application	Hardware platform
app_usb_aud_xk_316_mc	xcore.ai USB Audio 2.0 Multi-channel Audio Board
app_usb_aud_xk_216_mc	xcore-200 USB Audio 2.0 Multi-channel Audio Board
app_usb_aud_xk_evk_xu316	xcore.ai Evaluation Kit

The code is split into several modules (or *library*) directories, each their own GIT repository. The code for these libraries is included in the build by adding the library name to the `USED_MODULES` define in an application Makefile.

Each library has a `module_build_info` file that lists it's dependencies in `DEPENDENT_MODULES`. This allows dependency trees and nesting.

Most of the core code is contained in the *XMOS USB Audio Library* (`lib_xua`). A full list of core dependencies is shown in Figure 6.

Figure 6: Core dependencies of USB Audio

Library	Description
lib_xua	Common code for USB audio applications
lib_xud	Low level USB device library
lib_spdif	S/PDIF transmit and receive code
lib_adat	ADAT transmit and receive code
lib_mic_array	PDM microphone interface and decimator
lib_xassert	Lightweight assertions library

[13] https://www.xmos.ai/documentation/XM-014363-PC-7/html/tools-guide/tools-ref/cmd-line-tools/xmake-manual/xmake-manual.html
[14] https://git-scm.com

Some of these core dependencies will have their own dependencies, for example `lib_mic_array` depnds on `lib_xassert` (see above), `lib_logging` (a lightweight print library) and `lib_dsp` (a DSP library).

Applications may use additional dependencies to support the hardware platform or add features beyond core functionality. For example, the application for XK-AUDIO-316-MC uses the additional dependencies listed in Figure 7:

Figure 7: Example additonal dependencies

Library	Description
lib_i2c	I2C interface, used to configure DACs/ADCs etc

5.2 Build Configurations

Due to the flexibility of the reference design software there are a large number of build options. For example input and output channel counts, Audio Class version, interface types etc. A "build configuration" is a set of build options that combine to produce a binary with a certain feature set.

The following command builds all supported configurations:

```
xmake all
```

Build configurations are listed in the application Makefile with their associated options, a specific configuration can be built via the command line as follows:

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

Once build a corresponding binary for a configuration can be found in the following location:

```
<app name>/bin/<app name>_<config name>.xe
```

5.3 Configuration Naming

A naming scheme is employed in each application to link features to a build configuration/binary. Depending on the hardware interfaces available variations of the same basic scheme are used.

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")

Some example build options are listed in Figure 8.

For example, in this scheme, a configuration named `2xsx` would indicate Audio Class 2.0, MIDI disabled, S/PDIF output enabled and S/PDIF input disabled.

Some additional letters or numbers may also be used to denote things like channel counts etc. See comments in the application Makefile for details.

Figure 8: Example build options and naming

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

5.4 Quality & Testing

It is not possible for all build option permutations to be exhaustively tested. The *XMOS USB Audio Reference Design* software therefore defines three levels of quality:

- **Fully Tested** - the configuration is fully supported. A product based on it can be immediately put into to a production environment with high confidence. Quality assurance (QA) should cover any customised code/functionality.
- **Partially Tested** - the configuration is partially tested. A product based on it can be put into a production environment with medium confidence. Some additional QA is recommended.
- **Build Tested** - the configuration is guaranteed to build but has not been tested. Full QA is required.

Typically disabing a function should have no effect on QA. For example, disabling S/PDIF on a fully-tested configuration with it enabled should not effect its quality.

XMOS aims to provide fully tested configurations for popular device configurations and common customer requirements.

It is advised that full QA is applied to any product regardless of the quality level of the configuration it is based on.

Fully tested configurations can be found in the application Makefile. Partially and build tested configurations can be found in the `configs_partial.inc` and `configs_build.inc` files respectively. Using the command `xmake all` will only build fully tested configurations. Partially tested and build tested configurations can be accessed by setting the `PARTIAL_TEST_CONFIGS` and `BUILD_TEST_CONFIGS` variables respectively. For example:

```
xmake PARTIAL_TEST_CONFIGS=1 all
```

Pre-release (i.e. alpha, beta or RC) firmware should not be used as basis for a production device and may not be representative of the final release firmware. Additionally, some releases may include feaures of lesser quality level. For example a beta release may contain a feature still at alpha level quality. See application `README` for details of any such features.

Due to the similarities between the *xCORE-200* and *xCORE.ai* series feature sets, it is fully expected that all listed *xCORE-200* series configurations will operate as expected

on the *xCORE.ai* series and vice versa. It is therefore expected that a quality level of a configuration will migrate between XMOS device series.

5.5 A Typical USB Audio Application

This section provides a walk through of a typical USB Audio application. Where specific examples are required code is used from the application for *XK-AUDIO-316-MC* (`app_usb_aud_xk_316_mc`).

The applications in `sw_usb_audio` use the "Codeless Programming Model" as documented in `lib_xua`. Briefly, the `main()` function is used from `lib_xua` with build-time defines in the application configuring the framework provided by `lib_xua`. Various functions from `lib_xua` are then overridden to provide customisation. See `lib_xua` for full details.

Each application directory contains:

1. A `Makefile`
2. A `src` directory

The `src` directory is arranged into two directories:

1. A `core` directory containing source items that must be made available to the USB Audio framework
2. An `extensions` directory that includes extensions to the framework such as external device configuration etc

The `core` folder for each application contains:

1. A `.xn` file to describe the hardware platform the application will run on
2. An (optional) configuration header file to customised the framework provided by `lib_xua` named `xua_conf.h`

The *XCOMMON* build sytem autmatically locates `_conf.h` files in the source tree for all used `lib` dependencies.

5.5.1 Lib_xua Configuration

The `xua_conf.h` file contains all the build-time `#defines` required to tailor framework provided by `lib_xua` to the particular application at hand. Typically these over-ride default values in `xua_conf_default.h` in `lib_xua/api`.

Firstly in `app_usb_aud_xk_316_mc` the `xua_conf.h` file sets defines to determine overall capability. For this application most of the optional interfaces are disabled by default. This is because the applications provide a large number build configurations in the `Makefile` enabling various interfaces. For a product with a fixed specification this almost certainly would not be the case and setting in this file may be the preferred option.

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 S/PDIF output - Default is S/PDIF off */
#ifndef XUA_SPDIF_TX_EN
#define XUA_SPDIF_TX_EN     (0)
#endif

/* Enable/Disable S/PDIF input - Default is S/PDIF off */
#ifndef XUA_SPDIF_RX_EN
#define XUA_SPDIF_RX_EN     (0)
#endif

/* Enable/Disable ADAT output - Default is ADAT off */
#ifndef XUA_ADAT_TX_EN
#define XUA_ADAT_TX_EN      (0)
#endif

/* Enable/Disable ADAT input - Default is ADAT off */
#ifndef XUA_ADAT_RX_EN
#define XUA_ADAT_RX_EN      (0)
#endif

/* Enable/Disable Mixing core(s) - Default is on */
#ifndef MIXER
#define MIXER               (1)
#endif

/* Set the number of mixes to perform - Default is 0 i.e mixing disabled */
#ifndef MAX_MIX_COUNT
#define MAX_MIX_COUNT       (0)
#endif

/* Audio Class version - Default is 2.0 */
#ifndef AUDIO_CLASS
#define AUDIO_CLASS         (2)
#endif
```

Next, the file defines properties of the audio channels including counts and arrangements. By default the application provides 8 analogue channels for input and output.

The total number of channels exposed to the USB host (set via `NUM_USB_CHAN_OUT` and `NUM_USB_CHAN_IN`) are calculated based on the audio interfaces enabled. Again, this is due to the multiple build configurations in the application `Makefile` and likely to be hard-coded for a product.

```
/* Number of I2S channels to DACs*/
#ifndef I2S_CHANS_DAC
#define I2S_CHANS_DAC       (8)
#endif

/* Number of I2S channels from ADCs */
#ifndef I2S_CHANS_ADC
#define I2S_CHANS_ADC       (8)
#endif

/* Number of USB streaming channels - by default calculate by counting
  ↪ audio interfaces */
#ifndef NUM_USB_CHAN_IN
#define NUM_USB_CHAN_IN     (I2S_CHANS_ADC + 2*XUA_SPDIF_RX_EN + 8*
  ↪ XUA_ADAT_RX_EN)  /* Device to Host */
#endif

#ifndef NUM_USB_CHAN_OUT
#define NUM_USB_CHAN_OUT    (I2S_CHANS_DAC + 2*XUA_SPDIF_TX_EN + 8*
  ↪ XUA_ADAT_TX_EN)  /* Host to Device */
#endif

/*** Defines relating to channel arrangement/indices ***/
```

Channel indices/offsets are set based on the audio interfaces enabled. Channels are indexed from 0. Setting **SPDIF_TX_INDEX** to 0 would cause the S/PDIF channels to duplicate analogue channels 0 and 1. Note, the offset for analogue channels is always 0.

```
/* Channel index of S/PDIF Tx channels: separate channels after analogue
  ↪ channels (if they fit) */
#ifndef SPDIF_TX_INDEX
    #if (I2S_CHANS_DAC + 2*XUA_SPDIF_TX_EN) <= NUM_USB_CHAN_OUT
        #define SPDIF_TX_INDEX   (I2S_CHANS_DAC)
    #else
        #define SPDIF_TX_INDEX   (0)
    #endif
#endif

/* Channel index of S/PDIF Rx channels: separate channels after analogue
  ↪ channels */
#ifndef SPDIF_RX_INDEX
#define SPDIF_RX_INDEX     (I2S_CHANS_ADC)
#endif

/* Channel index of ADAT Tx channels: separate channels after S/PDIF
  ↪ channels (if they fit) */
#ifndef ADAT_TX_INDEX
    #if (I2S_CHANS_DAC + 2*XUA_SPDIF_TX_EN + 8*XUA_ADAT_TX_EN) <=
      ↪ NUM_USB_CHAN_OUT
        #define ADAT_TX_INDEX    (I2S_CHANS_DAC + 2*XUA_SPDIF_TX_EN)
    #else
        #define ADAT_TX_INDEX    (0)
    #endif
#endif

/* Channel index of ADAT Rx channels: separate channels after S/PDIF
  ↪ channels */
#ifndef ADAT_RX_INDEX
#define ADAT_RX_INDEX      (I2S_CHANS_ADC + 2*XUA_SPDIF_RX_EN)
#endif
```

The file then sets some frequency related defines for the audio master clocks and the maximum sample-rate for the device.

```
/* Master clock defines (in Hz) */
#ifndef MCLK_441
#define MCLK_441            (512*44100)    /* 44.1, 88.2 etc */
#endif

#ifndef MCLK_48
#define MCLK_48             (512*48000)    /* 48, 96 etc */
#endif

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

Due to the multi-tile nature of the *xCORE* architecture the framework needs to be informed as to which tile various interfaces should be placed on, for example USB, S/PDIF etc.

```
#define XUD_TILE            (0)
#define PLL_REF_TILE        (0)

#define AUDIO_IO_TILE       (1)
#define MIDI_TILE           (1)
```

The file also sets some defines for general USB ID's and strings. These are set for the XMOS reference design but vary per manufacturer:

```
#define VENDOR_ID           (0x20B1) /* XMOS VID */
#ifndef PID_AUDIO_2
#define PID_AUDIO_2         (0x0016)
#endif
#ifndef PID_AUDIO_1
#define PID_AUDIO_1         (0x0017)
#endif
#define PRODUCT_STR_A2      "XMOS xCORE.ai MC (UAC2.0)"
#define PRODUCT_STR_A1      "XMOS xCORE.ai MC (UAC1.0)"
```

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

5.5.2 User Functions

In addition to the `xua_conf.h` file, the application needs to provide implementations of some overridable user functions in `lib_xua` to provide custom functionality.

For `app_usb_aud_xk_316_mc` the implementations can be found in `src/extensions/audiohw.xc` and `src/extensions/audiostream.xc`

The two functions it overrides in `audiohw.xc` are `AudioHwInit()` and `AudioHwConfig()`. These are run from `lib_xua` on startup and sample-rate change respectively. Note, the default implementations in `lib_xua` are empty. These functions have parameters for sample frequency, sample depth, etc.

In the case of `app_usb_aud_xk_316_mc` these functions configure the external DAC's and ADC's via an I2C bus and configure the *xCORE* secondary PLL to generate the required master clock frequencies.

Due to the complexity of the hardware on the *XK-AUDIO-316-MC* the source code is not included here.

The application also overrides `UserAudioStreamStart()` and `UserAudioStreamStop()`. These are called from `lib_xua` when the audio stream to the device is started or stopped respectively. The applications uses these functions to enable/disable the LEDs on the board based on whether an audio stream is present (input or output).

```
#include <platform.h>

on tile[0]: out port p_leds = XS1_PORT_4F;

void UserAudioStreamStart(void)
{
    /* Turn all LEDs on */
    p_leds <: 0xF;
}

void UserAudioStreamStop(void)
{
    /* Turn all LEDs off */
    p_leds <: 0x0;
}
```

A media player application may choose to keep an audio stream open and simply send zero data when paused.

5.5.3 The Main Program

The `main()` function is the entry point to an application. In the *XMOS USB Audio Reference Design* software it is shared by all applications and is therefore part of the framework.

This section is largely informational as most developers should not need to modify the `main()` function. `main()` is located in `main.xc` in `lib_xua`, this file contains:

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

Full documentation can be found in `lib_xua`.

The first core run is a `usb_audio_core` task. This runs cores for the USB interface and buffering tasks for audio and endpoint buffering:

```
usb_audio_core(c_mix_out
#ifdef MIDI
    , c_midi
#endif
#ifdef IAP
    , c_iap
#ifdef IAP_EA_NATIVE_TRANS
    , c_ea_data
#endif
#endif
#if (MIXER)
    , c_mix_ctl
#endif
    , c_clk_int, c_clk_ctl, dfuInterface
#if (XUA_SYNCMODE == XUA_SYNCMODE_SYNC)
    , i_pll_ref
#endif
    VENDOR_REQUESTS_PARAMS_

);
```

This task runs various cores including one for the USB interfacing core (`XUD_Main()`):

```
            XUD_Main(c_xud_out, ENDPOINT_COUNT_OUT, c_xud_in,
              ↪ ENDPOINT_COUNT_IN,
                c_sof, epTypeTableOut, epTypeTableIn, usbSpeed, xudPwrCfg);
        }

        {
            unsigned x;
            thread_speed();

            /* Attach mclk count port to mclk clock-block (for feedback) */
            //set_port_clock(p_for_mclk_count, clk_audio_mclk);
#if(AUDIO_IO_TILE != XUD_TILE)
            set_clock_src(clk_audio_mclk_usb, p_mclk_in_usb);
            set_port_clock(p_for_mclk_count, clk_audio_mclk_usb);
            start_clock(clk_audio_mclk_usb);
#else
            /* Clock port from same clock-block as I2S */
            /* TODO remove asm() */
            asm("ldw %0, dp[clk_audio_mclk]":"=r"(x));
            asm("setclk res[%0], %1"::"r"(p_for_mclk_count), "r"(x));
#endif
            /* Endpoint & audio buffering cores */
            XUA_Buffer(c_xud_out[ENDPOINT_NUMBER_OUT_AUDIO],/* Audio Out*/
#if (NUM_USB_CHAN_IN > 0)

                c_xud_in[ENDPOINT_NUMBER_IN_AUDIO],         /* Audio In */
#endif
#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
#if (XUA_SPDIF_RX_EN || XUA_ADAT_RX_EN)
                /* 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
#if (HID_CONTROLS)
                , c_xud_in[ENDPOINT_NUMBER_IN_HID]
#endif
                , c_mix_out
#if (XUA_SYNCMODE == XUA_SYNCMODE_SYNC)
                , i_pll_ref
#endif
            );
            //:
        }

        /* Endpoint 0 Core */
        {
            thread_speed();
            XUA_Endpoint0( c_xud_out[0], c_xud_in[0], c_aud_ctl, c_mix_ctl,
              ↪  c_clk_ctl, c_EANativeTransport_ctrl, dfuInterface
              ↪ VENDOR_REQUESTS_PARAMS_);
        }

        //:
    }
}
#endif /* XUA_USB_EN */
```


```
#if (XUA_SPDIF_TX_EN) && (SPDIF_TX_TILE != AUDIO_IO_TILE)
void SpdifTxWrapper(chanend c_spdif_tx)
{
    unsigned portId;
    //configure_clock_src(clk, p_mclk);

    // TODO could share clock block here..
    // NOTE, Assuming SPDIF tile == USB tile here..
    asm("ldw %0, dp[p_mclk_in_usb]":"=r"(portId));
    asm("setclk res[%0], %1"::"r"(clk_mst_spd), "r"(portId));
    configure_out_port_no_ready(p_spdif_tx, clk_mst_spd, 0);
    set_clock_fall_delay(clk_mst_spd, 7);
    start_clock(clk_mst_spd);

    while(1)
    {
        spdif_tx(p_spdif_tx, c_spdif_tx);
    }
}
#endif

void usb_audio_io(chanend ?c_aud_in,
#if (XUA_SPDIF_TX_EN) && (SPDIF_TX_TILE != AUDIO_IO_TILE)
    chanend c_spdif_tx,
#endif
#if (MIXER)
    chanend c_mix_ctl,
#endif
    streaming chanend ?c_spdif_rx,
    chanend ?c_adat_rx,
    chanend ?c_clk_ctl,
    chanend ?c_clk_int
#if (XUD_TILE != 0)  && (AUDIO_IO_TILE == 0) && (XUA_DFU_EN == 1)
    , server interface i_dfu ?dfuInterface
#endif
#if (XUA_NUM_PDM_MICS > 0)
#if (PDM_TILE == AUDIO_IO_TILE)
    , streaming chanend c_ds_output[2]
#endif
    , chanend c_pdm_pcm
#endif
#if (XUA_SPDIF_RX_EN || XUA_ADAT_RX_EN)
    , client interface pll_ref_if i_pll_ref
#endif
)
{
#if (MIXER)
    chan c_mix_out;
#endif

#if (XUA_SPDIF_RX_EN || XUA_ADAT_RX_EN)
    chan c_dig_rx;
#else
    #define c_dig_rx null
#endif

#if (XUA_NUM_PDM_MICS > 0) && (PDM_TILE == AUDIO_IO_TILE)
    /* Configure clocks ports - sharing mclk port with I2S */
    xua_pdm_mic_config(p_mclk_in, p_pdm_clk, p_pdm_mics, clk_pdm);
#endif

#if (XUA_SPDIF_TX_EN) && (SPDIF_TX_TILE == AUDIO_IO_TILE)
    chan c_spdif_tx;

    /* Setup S/PDIF tx port - note this is done before par since sharing
      ↪ clock-block/port */
    spdif_tx_port_config(p_spdif_tx, clk_audio_mclk, p_mclk_in, 7);
#endif
```


```
/* Main for USB Audio Applications */
int main()
{
#if !XUA_USB_EN
    #define c_mix_out null
#else
    chan c_mix_out;
#endif

#ifdef MIDI
    chan c_midi;
#endif
#ifdef IAP
    chan c_iap;
#ifdef IAP_EA_NATIVE_TRANS
    chan c_ea_data;
#endif
#endif

#if (MIXER)
    chan c_mix_ctl;
#endif

#if (XUA_SPDIF_RX_EN)
    streaming chan c_spdif_rx;
#else
#define c_spdif_rx null
#endif

#if (XUA_ADAT_RX_EN)
    chan c_adat_rx;
#else
#define c_adat_rx null
#endif

#if (XUA_SPDIF_TX_EN) //&& (SPDIF_TX_TILE != AUDIO_IO_TILE)
    chan c_spdif_tx;
#endif

#if (XUA_SPDIF_RX_EN || XUA_ADAT_RX_EN)
    chan c_clk_ctl;
    chan c_clk_int;
#else
#define c_clk_int null
#define c_clk_ctl null
#endif

#if (XUA_DFU_EN == 1)
    interface i_dfu dfuInterface;
#else
    #define dfuInterface null
#endif

#if (XUA_NUM_PDM_MICS > 0)
    chan c_pdm_pcm;
    streaming chan c_ds_output[2];
#ifdef MIC_PROCESSING_USE_INTERFACE
    interface mic_process_if i_mic_process;
#endif
#endif

#if ((XUA_SYNCMODE == XUA_SYNCMODE_SYNC) || XUA_SPDIF_RX_EN ||
  ↪ XUA_ADAT_RX_EN)
    interface pll_ref_if i_pll_ref;
#endif

    USER_MAIN_DECLARATIONS

    par
```


```
#if XUA_USB_EN
#if (XUD_TILE != 0 ) && (AUDIO_IO_TILE != 0) && (XUA_DFU_EN == 1)
        /* Run flash code on its own - hope it gets combined */
        //#warning Running DFU flash code on its own
        on stdcore[0]: DFUHandler(dfuInterface, null);
#endif
#endif

#ifndef PDM_RECORD
#if (XUA_NUM_PDM_MICS > 0)
#if (PDM_TILE != AUDIO_IO_TILE)
        /* PDM Mics running on a separate to AudioHub */
        on stdcore[PDM_TILE]:
        {
            xua_pdm_mic_config(p_pdm_mclk, p_pdm_clk, p_pdm_mics, clk_pdm);
            xua_pdm_mic(c_ds_output, p_pdm_mics);
        }
#endif

#ifdef MIC_PROCESSING_USE_INTERFACE
        on stdcore[PDM_TILE].core[0]: XUA_PdmBuffer(c_ds_output, c_pdm_pcm,
          ↪  i_mic_process);
#else
        on stdcore[PDM_TILE].core[0]: XUA_PdmBuffer(c_ds_output, c_pdm_pcm)
          ↪ ;
#endif /*MIC_PROCESSING_USE_INTERFACE*/
#endif /*XUA_NUM_PDM_MICS > 0*/
#endif /*PDM_RECORD*/
    }

    return 0;
}

#endif
```

The specification of the channel arrays connecting to this driver are described in XM-005512-PC.

The channels connected to **`XUD_Main()`** are passed to the **`XUA_Buffer()`** function which implements audio buffering and also buffering for other Endpoints.

```
XUA_Buffer(c_xud_out[ENDPOINT_NUMBER_OUT_AUDIO],/* Audio Out*/
#if (NUM_USB_CHAN_IN > 0)

    c_xud_in[ENDPOINT_NUMBER_IN_AUDIO],         /* Audio In */
#endif
#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
#if (XUA_SPDIF_RX_EN || XUA_ADAT_RX_EN)
    /* 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
#if (HID_CONTROLS)
    , c_xud_in[ENDPOINT_NUMBER_IN_HID]
#endif
    , c_mix_out
#if (XUA_SYNCMODE == XUA_SYNCMODE_SYNC)
    , i_pll_ref
#endif
);
```

A channel connects this buffering task to the audio driver which controls the I2S output. It also forwards and receives audio samples from other interfaces e.g. S/PDIF, ADAT, as required:

```
    usb_audio_io(c_mix_out
#if (XUA_SPDIF_TX_EN) && (SPDIF_TX_TILE != AUDIO_IO_TILE)
        , c_spdif_tx
#endif
#if (MIXER)
        , c_mix_ctl
#endif
        , c_spdif_rx, c_adat_rx, c_clk_ctl, c_clk_int
#if (XUD_TILE != 0) && (AUDIO_IO_TILE == 0) && (XUA_DFU_EN == 1)
        , dfuInterface
#endif
#if (XUA_NUM_PDM_MICS > 0)
#if (PDM_TILE == AUDIO_IO_TILE)
        , c_ds_output
#endif
        , c_pdm_pcm
#endif
#if (XUA_SPDIF_RX_EN || XUA_ADAT_RX_EN)
        , i_pll_ref
#endif
    );
}
```

Finally, other task are create for various interfaces, for example, if MIDI is enabled a core is required to drive the MIDI input and output.

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

5.6 Adding Custom Code

The flexibility of the *XMOS USB Audio Reference Design* software is such that you can modify the reference applications to change the feature set or add extra functionality. Any part of the software can be altered since full source code is supplied.

The reference designs have been verified against a variety of host OS types at 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.

Developers are encouraged to use a version control system, i.e. *GIT*, to track changes to the codebase, however, this is beyond the scope of this document.

The general steps to producing a custom codebase are as follows:

1. Make a copy of the application directory (e.g. `app_usb_aud_xk_316_mc` or `app_usb_aud_xk_216_mc`) you wish to base your code on, to a separate directory with a different name.
2. Make a copy of any dependencies 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 re-flash 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 `xua_conf.h` with the specific defines you wish to set.
3. Add any custom code in other files you need.
4. Update `main.xc` to add any custom tasks

Whilst a developer may directly change the code in `main.xc` to add custom tasks this may not always be desirable. Doing this may make taking updates from *XMOS* non-trivial (the same can be said for any custom modifications to any core libraries). Since adding tasks is considered a reasonably common customisation defines `USER_MAIN_CORES` and `USER_MAIN_DECLARATIONS` are made available.

An example usage is shown in `app_usb_aud_xk_316_mc/src/extensions/user_main.h` In reality the developer must weigh up the pain of using these defines versus the pain of merging updates from *XMOS*.

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

5.6.1 Example: Changing Output Format

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

To do this you need to alter the main audio driver loop in `xua_audiohub.xc`. After the alteration you need to re-test the functionality.

Hint, a naive approach would simply include right-shifting the audio data by 7 bits before it is output to the port. This would of course lose LSB data depending on the sample-depth.

5.6.2 Example: Adding DSP to the Output Stream

To add some DSP requires an extra core of computation. Depending on the *xCORE* device being used you may have to disable some existing functionality to free up a core (e.g. disable S/PDIF). There are many ways that DSP processing can be added, the steps below outline one approach:

1. Remove some functionality using the defines in XM-005512-PC to free up a core as required.

2. Add another core to do the DSP. This core will probably have a single XC channel. This channel can be used to send and receive audio samples from the `XUA_AudioHub()` task. A benefit of modifying samples here is that samples from all inputs are collected into one place at this point. Optionally, a second channel could be used to accept control messages that affect the DSP. This could be from Endpoint 0 or some other task with user input - a core handling button presses, for example.

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

6 USB Audio Applications

IN THIS CHAPTER

The reference applications supplied in `sw_usb_audio` use the framework provided in `lib_xua` and provide qualified configurations of the framework which support, and are validated, on an accompanying reference hardware platform.

These reference design applications customise and extended this framework to provide the required functionality. This document will now go on to detail how each of the provided applications customise and extend the framework

The applications contained in this repo use `lib_xua` in a "code-less" manner. That it, they use the `main()` function from `lib_xua` and customise the code-base as required using build time defines and by providing implementations to the various required functions in order to support their hardware.

Please see `lib_xua` documentation for full details.

6.1 The xcore.ai 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.ai-series dual tile device. The related code can be found in `app_usb_aud_xk_316_mc`.

The design supports upto 8 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. It also supports S/PDIF, ADAT and MIDI input and output aswell as the mixing functionalty of `lib_xua`.

The design uses the following tasks:

- XMOS USB Device Driver (XUD)
- Endpoint 0
- Endpoint Buffer
- Decoupler
- AudioHub Driver
- Mixer
- S/PDIF Transmitter

- S/PDIF Receiver
- ADAT Receiver
- Clockgen
- MIDI

The software layout of the USB Audio 2.0 Reference Design running on the *xcore.ai* device is shown in Figure 9.

Each circle depicts a task running in a single core concurrently with the other tasks. The lines show the communication between each task.

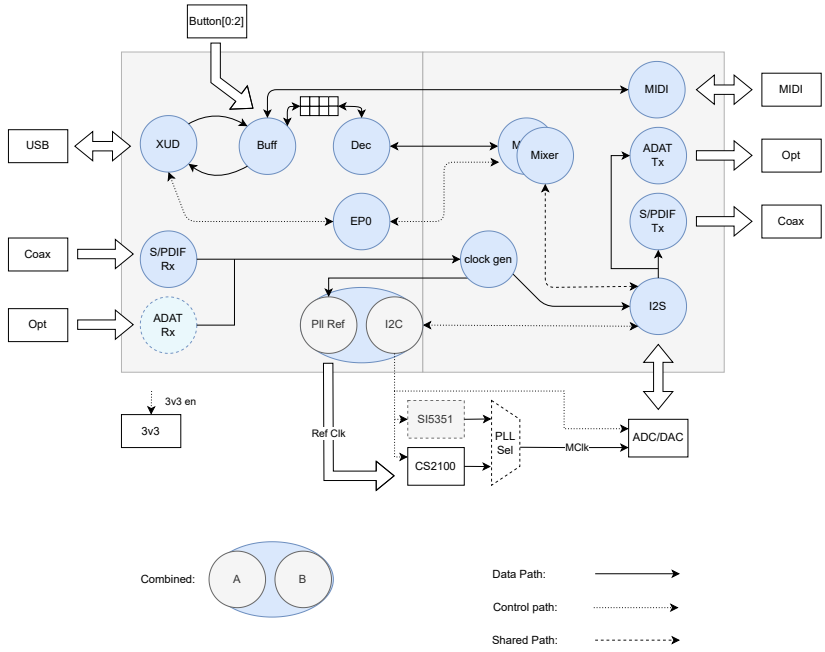

Figure 9: xcore.ai Multichannel Audio System/Core Diagram

6.1.1 Clocking and Clock Selection

As well as the secondary (application) PLL of the *xcore.ai* device the board includes two options for master clock generation:

- A Cirrus Logic CS2100 fractional-N clock multiplier allowing the master clock to be generated from a xCORE derived reference.
- A Skyworks Si5351A-B-GT CMOS clock generator.

The master clock source is chosen by driving two control signals as shown below:

Control Signal EXT_PLL_SEL	MCLK_DIR	Master Clock Source
0	0	Cirrus CS2100
1	0	Skyworks SI5351A-B-GT
X	1	xcore.ai secondary (application) PLL

Each of the sources have potential benefits, some of which are discussed below:

- The Cirrus CS2100 simplifies generating a master clock locked to an external clock (such as S/PDIF in or word clock in).
  - It multiplies up the PLL_SYNC signal which is generated by the xcore.ai device based on the desired external source (so S/PDIF in frame signal or word clock in).
- The Si5351A-B-GT offers very low jitter performance at a relatively lower cost than the CS2100. Locking to an external source is more difficult.
- The xcore.ai application PLL is obviously the lowest cost and significantly lowest power solution, however its jitter performance can not match the Si5351A which may be important in demanding applications. Locking to an external clock is possible but involves more complicated firmware and more MIPS.

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

Figure 10: 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 11: Master Clock Frequency Select

Value	Frequency
0	24.576MHz
1	22.579MHz

6.1.2 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 lib lib_i2c[15].

[15] http://www.github.com/xmos/lib_i2c

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

6.1.3 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.

6.1.4 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.

6.1.5 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 §5.2 for details and general information on build configuration naming scheme.

These fully validated build configurations are enumerated in the supplied Makefile

The build configuration naming scheme employed in the makefile is shown in Figure 12.

e.g. A build configuration named 2AMi10o10xsxxxx would signify: Audio class 2.0 running in asynchronous mode. *xCORE* is I2S master. Input and output enabled (10 channels each), no MIDI, S/PDIF input, no S/PDIF output, no ADAT or DSD.

In addition to this some terms may be appended onto a build configuration name to signify additional options. For example, *tdm* may be appended to the build configuration name to indicate the I2S mode employed.

6.2 The xcore-200 Multi-Channel Audio 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_xk_216_mc`.

Feature	Option 1	Option 2
Audio Class	1	2
USB Sync Mode	async: A	sync: S
I2S Role	slave: S	master: M
Input	enabled: i (channel count)	disabled: x
Output	enabled: i (channel count)	disabled: x
MIDI	enabled: m	disabled: x
S/PDIF input	enabled: s	disabled: x
S/PDIF input	enabled: s	disabled: x
ADAT input	enabled: a	disabled: x
ADAT output	enabled: a	disabled: x
DSD output	enabled: d	disabled: x

Figure 12: Build config naming scheme

The design supports upto 8 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. It also supports S/PDIF, ADAT and MIDI input and output aswell as the mixing functionalty of `lib_xua`.

The design uses the following tasks:

- XMOS USB Device Driver (XUD)
- Endpoint 0
- Endpoint Buffer
- Decoupler
- AudioHub Driver
- Mixer
- S/PDIF Transmitter
- S/PDIF Receiver
- ADAT Receiver
- Clockgen
- MIDI

The software layout of the USB Audio 2.0 Reference Design running on the *xCORE.ai* device is shown in Figure 13.

Each circle depicts a task running in a single core concurrently with the other tasks. The lines show the communication between each task.

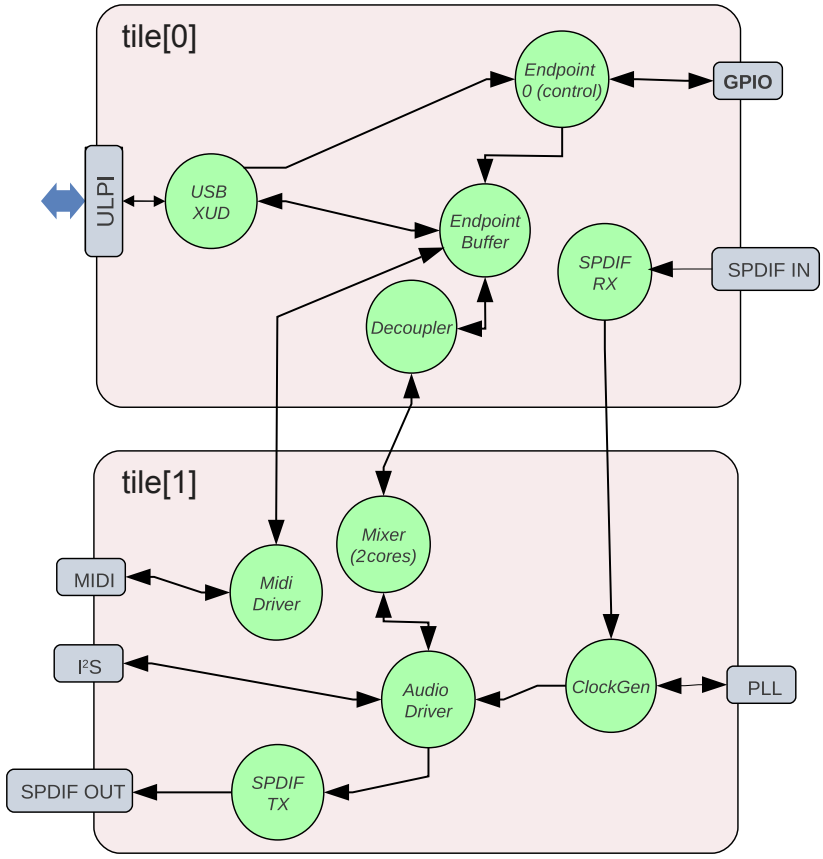

Figure 13: xcore-200 Multichannel Audio System/Core Diagram

6.2.1 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*:

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

Figure 14: Master Clock Source Selection

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

Figure 15: Master Clock Frequency Select

Value	Frequency
0	24.576MHz
1	22.579MHz

6.2.2 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 lib_i2c[16].

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

6.2.3 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.

6.2.4 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.

[16] http://www.github.com/xmos/lib_i2c

6.2.5 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 §5.2 for details and general information on build configuation naming scheme.

These fully validated build configurations are enumerated in the supplied Makefile.

In practise, due to the similarities between the *xcore-200* and *xCORE.ai* series feature set, it is fully expected that all listed *xcore-200* series configurations will operate as expected on the *xCORE.ai* series and vice versa.

The build configuration naming scheme employed in the makefile is shown in Figure 16.

Figure 16: Build config naming scheme

Feature	Option 1	Option 2
Audio Class	1	2
USB Sync Mode	async: A	sync: S
I2S Role	slave: S	master: M
Input	enabled: i (channel count)	disabled: x
Output	enabled: i (channel count)	disabled: x
MIDI	enabled: m	disabled: x
S/PDIF input	enabled: s	disabled: x
S/PDIF input	enabled: s	disabled: x
ADAT input	enabled: a	disabled: x
ADAT output	enabled: a	disabled: x
DSD output	enabled: d	disabled: x

e.g. A build configuration named 2AMi10o10xsxxxx would signify: Audio class 2.0 running in asynchronous mode. *xCORE* is I2S master. Input and output enabled (10 channels each), no MIDI, S/PDIF input, no S/PDIF output, no ADAT or DSD.

In addition to this some terms may be appended onto a build configuration name to signify additional options. For example, *tdm* may be appended to the build configuration name to indicate the I2S mode employed.

7 API

IN THIS CHAPTER

7.1 Configuration Defines

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

An application should override these defines in an optional `xua_conf.h` file or in the `Makefile` for a relevant build configuration.

This section documents commonly used defines, for full listings and documentation see the `lib_xua`.

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

Macro	PLL_REF_TILE
Description	Location (tile) of reference signal to CS2100. Default: AUDIO_IO_TILE

Macro	SPDIF_TX_TILE
Description	Location (tile) of SPDIF Tx. 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	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)

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

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	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)

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	XUA_SPDIF_TX_EN
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	XUA_SPDIF_RX_EN
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	XUA_ADAT_RX_EN
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 XUA_ADAT_RX_EN is true)

7.1.5.4 PDM Microphones

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

7.1.5.5 DFU

Macro	XUA_DFU_EN
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	Defines whether XMOS device runs as master (i.e. drives LR and Bit clocks) 0: XMOS is I2S master. 1: CODEC is I2s master. Default: 0 (XMOS is master)

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: “XMOS xCORE (UAC2.0)”

Macro	PRODUCT_STR_A1
Description	Product string for Audio Class 1.0 mode. Default: “XMOS xCORE (UAC1.0)”

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 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)

7.1.8 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

7.1.9 Power

Macro	XUA_POWERMODE
Description	Report as self or bus powered device. This affects descriptors and XUD usage and is important for USB compliance Default: XUA_POWERMODE_BUS

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.

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

Continued on next page

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)`

Continued on next page

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.

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*

What is the maximum channel count the device can support?

The maximum channel count of a device is a function of sample-rate and sample-depth. A standard high-speed USB Isochronous endpoint can handle a 1024 byte packet every microframe (125uS).

It follows then that at 192kHz the device/hosts expects 24 samples per frame (192000/8000). When using Asynchronous mode we must allow for +/- one sample, so 25 samples per microframe in this case.

Assuming 4 byte (32 bit) sample size, the bus expects ((192000/8000)+1) * 4 = 100 bytes per channel per microframe. Dividing the maximum packet size by this value yields the theoretical maximum channel count at the given frequency, that is 1024/100 = 10.24. Clearly this must be rounded down to 10 whole channels.