sw_usb_audio: USB Audio reference designs

Publication Date: 2025/10/14
Document Number: XM-008854-UG v9.2.0

XMOS

IN THIS DOCUMENT

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.

The reader should also familiarise themselves with the XMOS USB device library (lib_xud) and the XMOS USB audio library (lib_xua)

Note

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

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
	USB Audio Class 2.0
	USB Firmware Upgrade (DFU) 1.1
	USB MIDI Device Class 1.0
Audio	I²S/TDM
	S/PDIF (receive may be limited to 96kHz depending on external hardware)
	ADAT
	Direct Stream Digital (DSD)
	PDM Microphones
	MIDI

Supported sample frequencies

16kHz to 384kHz[1]

Supported devices

XMOS Devices	*xcore-200* Series
	xcore.ai Series

Requirements

Development Tools	XTC Development Tools (see readme for required version)
USB	*xcore* device with integrated USB phy
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

Reference code provided without charge under license from *XMOS*.

[1] Not all features may be supported at all sample frequencies, simultaneously or on all devices.

2 USB Audio hardware platforms

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

Board	*xcore* device	Analog channels	Digital Rx/Tx & MIDI
XK_EVK_XU316	*xcore.ai*	2 in + 2 out	N/A
XK_AUDIO_316_MC_AB	*xcore.ai*	8 in + 8 out	Supported
XK_AUDIO_216_MC_AB	*xcore-200*	8 in + 8 out	Supported

Each of the platforms supported has a Board Support Package (BSP), the code for which can be be found in lib_board_support. The code in `lib_board_support` abstracts away all of the hardware setup including enabling external hardware blocks and DAC and ADC configuration and provides a translation layer from the common API supported by lib_xua for initialising and configuring hardware on a sample rate or stream format change.

Detailed feature sets for the each of the supported boards can be found in the documentation for lib_board_support.

3 Driver support

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 Third party Windows drivers

For some products it may be desirable to use a third-party driver for Windows. A number of 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

Note

Many of the benefits listed above apply to both UAC1.0 and UAC2.0 and the Thesycon Driver supports both class versions.

4 Quick start

Warning

XMOS development boards are typically supplied with no firmware pre-programmed.

The following steps explain how to program the USB Audio software onto a development board and use it as a USB Audio device. Each step is explained in detail in the following sections.

1. Download the latest USB Audio 2.0 Device software release from the XMOS USB & Multichannel Audio webpage, using the *DOWNLOAD SOFTWARE* link.
   - Before downloading the software, review the licence and click **Accept** to initiate the download.

   (Section *USB Audio reference software*.)
2. If using a Windows host computer, download the *USB Audio Class 2.0 Evaluation Driver for Windows*.
   - On the XMOS USB & Multichannel Audio developer webpage, follow the *DRIVER SUPPORT* link, and click on *DOWNLOAD EVALUATION DRIVER*. Once downloaded, run the executable and install the driver.

   (Section *USB Audio Class (UAC) 2.0 evaluation driver for Windows*.)
3. Download and install the XMOS XTC Tools
   - The required XTC Tools version for compiling USB Audio applications can be found in the README file supplied in the software package. Be sure to download the correct version of the tools.

   (Section *XMOS XTC development tools*)
4. Compile the software relevant to the available board.

   (Section: *Building the firmware*)
5. Connect the board to the host machine using two connections: one for the USB Audio device and another for the debug/programming interface, and program the compiled binary onto the board.

   (Section *Running the firmware*)
6. Connect audio input and output devices, and play audio.

   (Section *Playing/recording audio*)

4.1 USB Audio reference software

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

When downloading the software for the first time, the user needs to register at http://www.xmos.com/.

To download the software:

1. On the XMOS USB & Multichannel Audio webpage, follow the *DOWNLOAD 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*.

4.2 USB Audio Class (UAC) 2.0 evaluation driver for Windows

Note

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

Note

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 (UAC) 1.0. To use a UAC 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 - such as synchronisation to an external S/PDIF or ADAT clock
- 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 partner *Thesycon*.

The evaluation driver is available via the USB Audio Driver Support webpage.

4.3 XMOS XTC development tools

The *XMOS XTC tools* provide everything required to develop applications for *xcore multicore microcontrollers* and can be downloaded, free of charge, from XMOS XTC tools. Installation instructions can be found here. Be sure to pay attention to the section Installation of required third-party 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* device. 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 VS Code graphical code editor.

4.4 Building the firmware

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

Applications are compiled using XCommon CMake which is a CMake based build system.

> **Note**
>
> See *Build system* for more details.

Each board is supported by a dedicated application located in its own directory. The boards and their corresponding applications are listed in Table 2.

Table 2: Boards and their applications

Board	Application
XK-EVK-XU316	app_usb_aud_xk_evk_xu316
XK-AUDIO-216-MC	app_usb_aud_xk_audio_216_mc
XK-AUDIO-316-MC	app_usb_aud_xk_audio_316_mc

The primary configuration for applications is in *CMakeLists.txt*. It is present in each application directory (e.g. **app_usb_aud_xk_audio_316_mc**). This file specifies build configs, sources, build options and dependencies.

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

1. Unzip the package zip to a known location
2. From the relevant application directory (e.g. **app_usb_aud_xk_audio_316_mc**), execute the commands:

```
cmake -G "Unix Makefiles" -B build
xmake -j -C build
```

These steps will configure and build all of the available and supported build configurations for the application.

4.5 Running the firmware

To support different feature sets and products, multiple build configurations are provided. Each configuration produces a distinct binary. Table 3 lists the recommended build configurations for initial evaluation.

Table 3: Applications and suggested build configuration for quick start

Application	Suggested build config	Description
app_usb_aud_xk_evk_xu316	2AMi10o10xssxxx	UAC 2.0, 10 ch in/out, 8 analogue channels in/out, S/PDIF in/out
app_usb_aud_xk_audio_216_mc	2AMi10o10xssxxx	UAC 2.0, 10 ch in/out, 8 analogue channels in/out, S/PDIF in/out
app_usb_aud_xk_audio_316_mc	2AMi2o2xxxxxx	UAC 2.0, 2 ch in/out, 2 analogue channels in/out

During development, it is common to load the application binary directly into the device's internal RAM via JTAG and execute it immediately.

Boards require two USB connections, one for the USB Audio device and another for a debug interface. Depending on the board being used, the debug interface will either use an integrated or xTAG.

To run one of the compiled binaries, follow these steps:

1. Connect the USB Audio board to the host computer with a USB cable (typically labelled `USB` or `USB DEVICE`)
2. Connect the xTAG to the USB Audio board, and then connect the xTAG to the host computer using a separate USB cable. *Note: Some boards have an integrated xTAG.*
3. Ensure that any required external power supplies are connected.

Once everything is connected, run the binary on the target device using a command like the following:

```
xrun ./bin/2AMi10o10xssxxx/app_usb_aud_xk_316_mc_2AMi10o10xssxxx.xe
```

The device should now appear as a USB Audio Device on the host computer. It will continue to function as such until the board is power cycled.

4.6 Writing the application binary to flash

If desired, the application binary can be programmed into the device's boot flash. To do so:

1. Connect the USB Audio board to the host computer with a USB cable (typically labelled `USB` or `USB DEVICE`)
2. Connect the xTAG to the USB Audio board, then connect the xTAG to the host computer using a separate USB cable. *Note: Some boards include an integrated xTAG.*
3. Ensure any required external power supplies are connected.

Once everything is connected, flash the binary on the target device using a command like the following:

```
xflash ./bin/2AMi10o10xssxxx/app_usb_aud_xk_316_mc_2AMi10o10xssxxx.xe
```

After flashing, the device will automatically reboot and begin executing the application. Subsequent power cycles will cause the device to boot and run the flashed binary.

4.7 Playing/recording audio

To play and record audio using the USB Audio board, follow these steps:

1. Connect the board to a power supply, if required. *Note: Some boards are powered via USB and do not require an external supply.*
2. Connect the board to a host system using a USB cable. Ensure the host supports USB Audio Class.
3. Install the Windows USB Audio 2.0 demonstration driver, if required.
4. Connect audio input/output devices to the appropriate connectors on the board (e.g., powered speakers, MP3 player). For multichannel boards use input/output 1/2.
5. In your audio application, select the *XMOS* USB Audio device.
6. Begin audio playback and/or recording.

5 USB Audio programming guide

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

5.1 Project structure

Build system

The *XMOS USB Audio Reference Design* software and associated libraries employ the *XCommon CMake* build system. The XCommon CMake build system uses CMake to configure and generate the build environment which can then be built using xmake. As part of configuring the build environment, if there are any missing dependencies, *XCommon CMake* fetches them using **`git`**.

> **Note**
>
> All required dependencies are included in the *sw_usb_audio* zip download.

Applications and libraries

The sw_usb_audio GIT repository includes multiple application directories. Each application directory contains a **`CMakeLists.txt`** file which describes the build configs for that application. The format of the **`CMakeLists.txt`** is described here *XCommon CMake* uses the **`CMakeLists.txt`** to generate Makefiles that can be compiled using **`xmake`** into executables. Typically, there's one application directory per hardware platform. Applications and their respective hardware platforms are listed in Table 4.

Table 4: 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 applications depend on several modules (or *libraries*), each of which have their own GIT repository. The immediate dependency libraries for the applications are specified by setting the **`APP_DEPENDENT_MODULES`** variable in the deps.cmake file. **`deps.cmake`** lists the common dependencies for all the applications and is included in each application's **`CMakeLists.txt`**.

The dependency list specified in the **`deps.cmake`** can be extended to add new dependencies. Refer to the *XCommon CMake* Dependency Format documentation for more information about adding dependencies.

A shared file containing common dependencies ensures a consistent set of dependencies between all of the applications.

Each library has a **`lib_build_info.cmake`** which lists the library source, compile flags and dependencies. The library dependencies are specified in the **`LIB_DEPENDENT_MODULES`** variable in the **`lib_build_info.cmake`**. This allows de-

pendency trees and nesting. *XCommon CMake* builds up a tree which is traversed depth-first, and populates the sandbox, fetching any missing dependencies by cloning them from github.

Most of the core code is contained in the XMOS USB Audio Library (lib_xua). A full list of core dependencies is shown in Table 5.

Table 5: 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

Note

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_xcore_math** (a DSP library).

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 build configurations are listed in the application **CMakeLists.txt** file. The build config names are appended to the **APP_COMPILER_FLAGS** variable to list the options for the compiler to use when compiling all source files for the given build configuration (APP_COMPILER_FLAGS_<build config>). For example:

```
set(APP_COMPILER_FLAGS_2AMi10o10xssxxx ${SW_USB_AUDIO_FLAGS}
                                       -DXUA_SPDIF_TX_EN=1
                                       -DXUA_SPDIF_RX_EN=1)
```

specifies that the compiler flags used when compiling the **2AMi10o10xssxxx** build config are everything defined in the **SW_USB_AUDIO_FLAGS** variable plus two extra compiler options for enabling S/PDIF transmit and receive.

To configure the build configurations, run the **cmake** command from the application (e.g. **app_usb_aud_xk_audio_316_mc**) directory:

```
cmake -G "Unix Makefiles" -B build
```

This will create a directory called **build** within the application directory. The output displayed on stdout for the **cmake** command will contain the list of all the build configurations for that application. For example,

```
-- Configuring application: app_usb_aud_xk_evk_xu316
-- Found build configs:
```

(continues on next page)

(continued from previous page)

```
-- 1AMi2o2xxxxxx
-- 2AMi2o2xxxxxx
```

The `cmake` command generates the *Makefile* for compiling the different build configurations. The *Makefile* is created in the `build` directory.

The next step is to run the `xmake` command which executes the commands in the Makefile to build the executables corresponding to the build configs. To build all supported configurations for a given application, from the application directory (e.g. `app_usb_aud_xk_audio_316_mc`), run:

```
xmake -C build
```

This will run the `xmake` command in the `build` directory. The built executables are stored in the `<app name>/bin/<config name>` directories. For example, the `app_usb_aud_xk_316_mc/bin/2AMi8o8xxxxxx` directory contains the `app_usb_aud_xk_316_mc_2ASi8o8xxxxxx.xe` executable. Note how the name of the executable is set to <app_name>_<config_name>.xe:

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

To build a specific build configuration, after running the `cmake` command, run `xmake` with the build config specified:

```
xmake -C build <build config>
```

For example:

```
xmake -C build 2AMi10o10xssxxx
```

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 Table 6.

Table 6: Example build options and naming

Build Option Name	Options	Denoted by
Audio Class Version	1 or 2	1 or 2
USB synchronisation type	Asynchronous or Synchronous	A or S
Device I2S role	Master or Slave	M or S
USB IN channels		i<number>
USB OUT channels		o<number>
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
ADAT Input	on or off	a or x
ADAT Output	on or off	a or x
DSD	on or off	d or x

For example, in this scheme, a configuration named `2AMi8o8msxxax` would indicate Audio Class 2.0, USB asynchronous mode, *xcore* is I²S master, 8 USB IN channels, 8 USB OUT channels, MIDI enabled, S/PDIF input enabled, S/PDIF output disabled, ADAT input disabled, ADAT output enabled and DSD disabled.

See comments in the application *CMakeLists.txt* for details.

5.4 Quality & testing

It is not practical 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.

Note

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

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

Note

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 CMakeLists.txt. Partially and build tested configurations can be found in the `configs_partial.cmake` and `configs_build.cmake` files respectively.

Running `cmake -G "Unix Makefiles" -B build` will only configure the fully tested configurations and following this up with the `xmake -C build` command will build only these.

To configure and build the partially tested configs in addition to the fully tested ones, run cmake with the `PARTIAL_TESTED_CONFIGS` variable set to 1:

```
cmake -G "Unix Makefiles" -B build -DPARTIAL_TESTED_CONFIGS=1
```

Following this with the `xmake -C build` command will build both fully and partially tested configs.

Similarly to also build the build tested configs along with the fully tested ones, run cmake with `BUILD_TESTED_CONFIGS` set to 1, followed by the `xmake` command:

```
cmake -G "Unix Makefiles" -B build -DBUILD_TESTED_CONFIGS=1
```

Note that setting `BUILD_TESTED_CONFIGS` to 1 internally also set the `PARTIAL_TESTED_CONFIGS` to 1. So running `cmake` with `BUILD_TESTED_CONFIGS` set to 1 will configure the fully tested, partially tested and build-only configs and following this up with an `xmake -C build` will build all the 3 types of configs.

Note

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 features 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.

Note

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`).

> **Note**
>
> 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:

- A `CMakeLists.txt`
- A `src` directory

The `src` directory is arranged into two directories:

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

The `core` folder for each application contains:

- A `.xn` file to describe the hardware platform the application will run on
- An optional header file to customise the framework provided by `lib_xua` named `xua_conf.h`

`lib_xua` configuration

The `xua_conf.h` file contains all the build-time `#defines` required to tailor the framework provided by `lib_xua` to the particular application at hand. Typically these override 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 `CMakeLists.txt` 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 `CMakeLists.txt`.

```
/* 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
```

(continues on next page)

(continued from previous page)

```
/* 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 `CMakeLists.txt` 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
    #define ADAT_TX_INDEX    (I2S_CHANS_DAC + 2*XUA_SPDIF_TX_EN)
#endif
```

(continues on next page)

(continued from previous page)

```
/* 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

/* Minumum sample frequency device runs at */
#ifndef MIN_FREQ
#define MIN_FREQ           (44100)
#endif

/* Maximum sample 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 XUA_XUD_TILE_NUM           (0)
#define XUA_PLL_REF_TILE_NUM       (0)

#define XUA_AUDIO_IO_TILE_NUM      (1)
#define XUA_MIDI_TILE_NUM          (1)
```

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

```
#define VENDOR_ID          (0x20B1) /* XMOS VID */
#ifndef PID_AUDIO_2
#if MIDI
    #define PID_AUDIO_2        (0x0020) /* Enumerate with a different PID since the
                                         * interface numbering is different between MIDI
                                         * enabled vs disabled and the Thesycon driver doesnt
                                         * support different interfae layouts with the same PID */
#else
    #define PID_AUDIO_2        (0x0016)
#endif
#endif
#ifndef PID_AUDIO_1
#define PID_AUDIO_1        (0x0017)
#endif

#ifndef DFU_PID
#if (AUDIO_CLASS == 1)
#define DFU_PID             (0xD000 + PID_AUDIO_1)
#else
#define DFU_PID             (0xD000 + PID_AUDIO_2)
#endif
#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 *Configuration defines*.

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 DACs and ADCs 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 `UserAudioStreamState()`. This function is called from `lib_xua` when the audio stream to the device is started or stopped.

The application uses this function to enable/disable the on board LEDs based on whether an audio stream is active (input or output).

```
// Copyright 2022-2025 XMOS LIMITED.
// This Software is subject to the terms of the XMOS Public Licence: Version 1.
#include <platform.h>

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

void UserAudioStreamState(int inputActive, int outputActive)
{
    if(inputActive || outputActive)
    {
        /* Turn all LEDs on */
        p_leds <: 0xF;
    } else
    {
        /* Turn all LEDs off */
        p_leds <: 0x0;
    }
}
```

Note

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

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 items in the `par` include running tasks for the Endpoint 0 implementaton and buffering tasks for audio and endpoint buffering:

```
            {
                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(SECOND_MCLK_REQUIRED)
                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
                /* XUA_AUDIO_IO_TILE_NUM == XUA_XUD_TILE_NUM */
                /* 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));
                /* This clock block is started in audiohub in case we first need to connect other logic
                   e.g. digital Tx to it before starting */
#endif
                /* Endpoint & audio buffering cores - buffers all EP's other than 0 */
                XUA_Buffer(
#if (NUM_USB_CHAN_OUT > 0)
                            c_xud_out[ENDPOINT_NUMBER_OUT_AUDIO],       /* Audio Out*/
#endif
#if (NUM_USB_CHAN_IN > 0)
                            c_xud_in[ENDPOINT_NUMBER_IN_AUDIO],         /* Audio In */
#endif
#if (NUM_USB_CHAN_OUT > 0) && ((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 (XUA_HID_ENABLED)
                            , c_xud_in[ENDPOINT_NUMBER_IN_HID]
#endif
                            , c_mix_out
#if (XUA_SYNCMODE == XUA_SYNCMODE_SYNC)
                            , c_audio_rate_change
    #if (!XUA_USE_SW_PLL)
                            , i_pll_ref
    #else
                            , c_sw_pll
    #endif
#endif
                        );
                //:
            }
#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, dfuInterface VENDOR_
↪REQUESTS_PARAMS_);
            }
```

It also runs a task for the USB interfacing thread (`XUD_Main()`):

```
                XUD_Main(c_xud_out, ENDPOINT_COUNT_OUT, c_xud_in, ENDPOINT_COUNT_IN,
```

The specification of the channel arrays connecting to this driver are described in the documentation for lib_xud.

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(
#if (NUM_USB_CHAN_OUT > 0)
                            c_xud_out[ENDPOINT_NUMBER_OUT_AUDIO],       /* Audio Out*/
#endif
#if (NUM_USB_CHAN_IN > 0)
                            c_xud_in[ENDPOINT_NUMBER_IN_AUDIO],         /* Audio In */
#endif
#if (NUM_USB_CHAN_OUT > 0) && ((NUM_USB_CHAN_IN == 0) || defined(UAC_FORCE_FEEDBACK_EP))
                            c_xud_in[ENDPOINT_NUMBER_IN_FEEDBACK],      /* Audio FB */
#endif
#ifdef MIDI
```

(continues on next page)

(continued from previous page)

```
                            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 (XUA_HID_ENABLED)
                            , c_xud_in[ENDPOINT_NUMBER_IN_HID]
#endif
                            , c_mix_out
#if (XUA_SYNCMODE == XUA_SYNCMODE_SYNC)
                            , c_audio_rate_change
    #if (!XUA_USE_SW_PLL)
                            , i_pll_ref
    #else
                            , c_sw_pll
    #endif
#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(
#if (NUM_USB_CHAN_OUT > 0) || (NUM_USB_CHAN_IN > 0) || XUA_HID_ENABLED || defined(MIDI)
                /* Connect audio system to XUA_Buffer(); */
                c_mix_out
#else
                /* Connect to XUA_Endpoint0() */
                c_aud_ctl
#endif
#if (XUA_SPDIF_TX_EN) && (XUA_SPDIF_TX_TILE_NUM != XUA_AUDIO_IO_TILE_NUM)
                , c_spdif_tx
#endif
#if (MIXER)
                , c_mix_ctl
#endif
                , c_spdif_rx, c_adat_rx, c_clk_ctl, c_clk_int
#if (XUA_XUD_TILE_NUM != 0) && (XUA_AUDIO_IO_TILE_NUM == 0) && (XUA_DFU_EN == 1)
                , dfuInterface
#endif
#if (XUA_NUM_PDM_MICS > 0)
                , c_pdm_pcm
#endif
#if (XUA_SPDIF_RX_EN || XUA_ADAT_RX_EN)
                , i_pll_ref
#endif
#if (XUA_SYNCMODE == XUA_SYNCMODE_SYNC)
                , c_audio_rate_change
#endif
#if ((XUA_SPDIF_RX_EN || XUA_ADAT_RX_EN) && XUA_USE_SW_PLL)
                , p_for_mclk_count_audio
                , c_sw_pll
#endif
            );
        }
```

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

```
        on tile[XUA_MIDI_TILE_NUM]:
        {
            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 the reference applications can be modified to change the feature set or add extra functionality. Any part of the software can be altered since full source code is supplied.

Note

The reference designs have been verified against a variety of host operating systems at different samples rates. Modifications to the code may invalidate the results of this verification and fully retesting the resulting software is strongly recommended.

Note

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 reference application directory (e.g. `app_usb_aud_xk_316_mc` or `app_usb_aud_xk_216_mc`) to a separate directory with a different name. Modify the new application to suit the custom requirements. For example:
   - Provide the `.xn` file for the target hardware platform by setting the `APP_HW_TARGET` in the application's `CMakeLists.txt`.
   - Update `xua_conf.h` with specific defines for the custom application.
   - Add any other custom code in the files as needed.
   - Update the `main.xc` to add any custom tasks.
2. Make a copy of any dependencies that require modification (in most cases, this step is unnecessary). Update the custom application's `CMakeLists.txt` to use these new modules.
3. After making appropriate changes to the code, rebuild and re-flash the device for testing.

Note

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 reasonably common, customisation defines `USER_MAIN_TASKS`, `USER_MAIN_DECLARATIONS` and `USER_MAIN_GLOBALS` are made available. Alternatively "optional" headers file can be used to add custom tasks and declarations, these should be named as follows and placed in the application source tree: `xua_conf_tasks.h`, `xua_conf_globals.h` and `xua_conf_declarations.h`

An example usage is shown in `app_usb_aud_xk_316_mc/src/core/xua_conf_tasks.h` and `app_usb_aud_xk_316_mc/src/core/xua_conf_globals.h` In reality the developer must weigh up the inconvenience of using these defines or header files versus the inconvenience of merging updates from *XMOS* into a modified codebase.

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

Example: Changing output format

Customising the digital output format may be required, for example, to support a CODEC that expects sample data right-justified with respect to the word clock.

To achieve this, alter the main audio driver loop in `xua_audiohub.xc`. After making the alteration, re-test the functionality to ensure proper operation.

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.

Example: Adding DSP to the output stream

To add some DSP requires an extra thread of computation. Depending on the *xcore* device being used, some existing functionality might need to be disabled to free up a thread (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 *Configuration defines* to free up a thread as required.

2. Add another thread 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 thread handling button presses, for example.

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

6 USB Audio applications

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 extend this framework to provide the required functionality. This document will now examine in 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 is, 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.

Refer to *lib_xua* <https://www.xmos.com/file/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 *XK_AUDIO_316_MC* hardware described in *USB Audio hardware platforms* 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 192 kHz (assuming the use of I²S). This can be further increased by utilising TDM. It also supports S/PDIF, ADAT and MIDI input and output as well 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 Transmitter
- ADAT Receiver
- Clockgen
- MIDI

The software layout of the USB Audio 2.0 Reference Design running on the *xcore.ai* device is shown in Fig. 1.

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

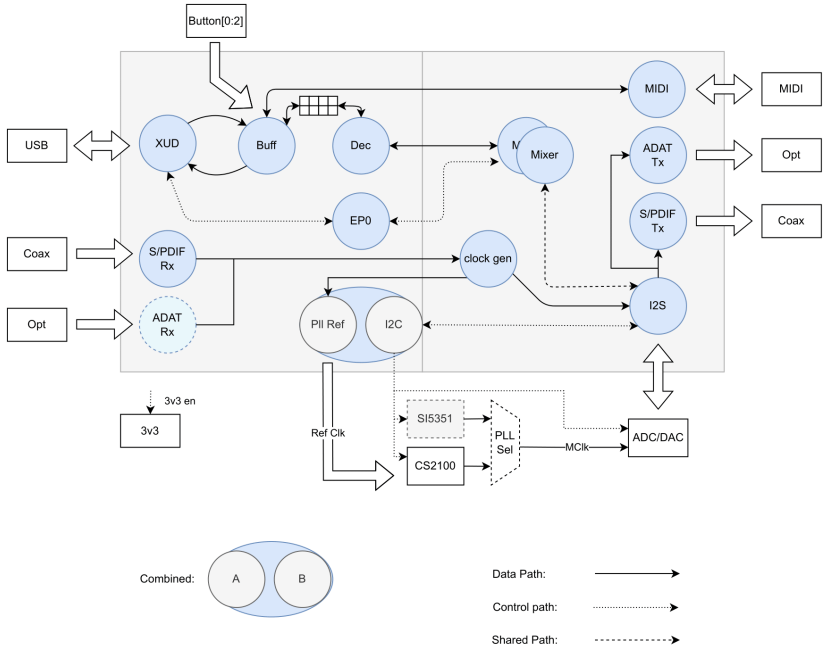

Fig. 1: *xcore.ai* multichannel audio system/task diagram

Audio hardware

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 clock.
- A Skyworks Si5351A-B-GT CMOS clock generator.

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

Control Signal		Master Clock Source
EXT_PLL_SEL	MCLK_DIR	
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* secondary 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*:

Table 7: 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*:

Table 8: Master Clock Frequency Select

Value	Frequency
0	24.576MHz
1	22.579MHz

DAC and ADC The board is equipped with four PCM5122 stereo DACs from Texas instruments and two quad-channel PCM1865 ADCs from Texas Instruments, giving 8 channels of analogue output and 8 channels of analogue input. Configuration of both the DAC and the ADC takes place over I²C.

Configuring audio hardware

All of the external audio hardware is configured using lib_board_support.

> **Note**
>
> `lib_board_support` has the I²C library (lib_i2c) in its dependency list.

The hardware targeted is the *XMOS XU316 Multichannel Audio board* (*XK-AUDIO-316-MC*). The *lib_board_support* functions `xk_audio_316_mc_ab_board_setup()`, `xk_audio_316_mc_ab_i2c_master()`, `xk_audio_316_mc_ab_AudioHwInit()` and `xk_audio_316_mc_ab_AudioHwConfig()` are called at various points during initialisation and runtime to start the I²C master, to initialise and configure the audio hardware.

The audio hardware configuration is set in the `config` structure of type `xk_audio_316_mc_ab_config_t` which is passed to the `xk_audio_316_mc_ab_board_setup()`, `xk_audio_316_mc_ab_AudioHwInit()` and `xk_audio_316_mc_ab_AudioHwConfig()` functions.

```
static xk_audio_316_mc_ab_config_t config =
{
    // clk_mode
    (XUA_SYNCMODE == XUA_SYNCMODE_SYNC || XUA_SPDIF_RX_EN || XUA_ADAT_RX_EN)
    ? ( XUA_USE_SW_PLL
        ? CLK_PLL : CLK_CS2100 )
    : CLK_FIXED,

    // dac_is_clk_master
    CODEC_MASTER,

    // default_mclk
    (DEFAULT_FREQ % 22050 == 0) ? MCLK_441 : MCLK_48,

    // pll_sync_freq
    PLL_SYNC_FREQ,

    // pcm_format
    XUA_PCM_FORMAT,

    // i2s_n_bits
    XUA_I2S_N_BITS,

    // i2s_chans_per_frame
    I2S_CHANS_PER_FRAME
};
```

`xk_audio_316_mc_ab_board_setup()` function is called from the wrapper function `board_setup()` as part of the application's initialisation process. It performs the required port operations to enable the audio hardware on the platform.

`xk_audio_316_mc_ab_i2c_master()` function is called after `board_setup()` during initialisation and it starts the I²C master task. This is required to allow the audio hardware to be configured over I²C, remotely from the other tile, due to the IO arrangement of the *XK-AUDIO-316-MC* board.

```
on tile[0]: {
                board_setup();
                xk_audio_316_mc_ab_i2c_master(i2c);
            }


on tile[1]: {
                unsafe
                {
                    i_i2c_client = i2c[0];
                }
            }
```

The *AudioHwInit()* function is implemented to make a call to the `lib_board_support` function `xk_audio_316_mc_ab_AudioHwInit()` to power up and initialise the audio hardware ready for a configuration.

The *AudioHwConfig()* function configures the audio hardware post initialisation. It is typically called each time a sample rate or stream format change occurs. It is implmented to make a call to the `lib_board_support` function `xk_audio_316_mc_ab_AudioHwConfig()`.

For further details on the hardware platform and the functions available for configuring it refer to lib_board_support documentation.

Validated build options

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

The design has only been fully validated against the build options as set in the application as distributed in the *CMakeLists.txt*. See *Build configurations* for details and general information on build configuration naming scheme.

These fully validated build configurations are enumerated in the supplied *CMakeLists.txt*.

The build configuration naming scheme employed in the CMakeLists.txt is shown in Table 9.

Table 9: 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. The *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 *XK_AUDIO_216_MC_AB* hardware described in *USB Audio hardware platforms* and is implemented on an *xcore-200* series dual tile device. The related code can be found in `app_usb_aud_xk_216_mc`.

The design supports upto 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 as well 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 Fig. 2.

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

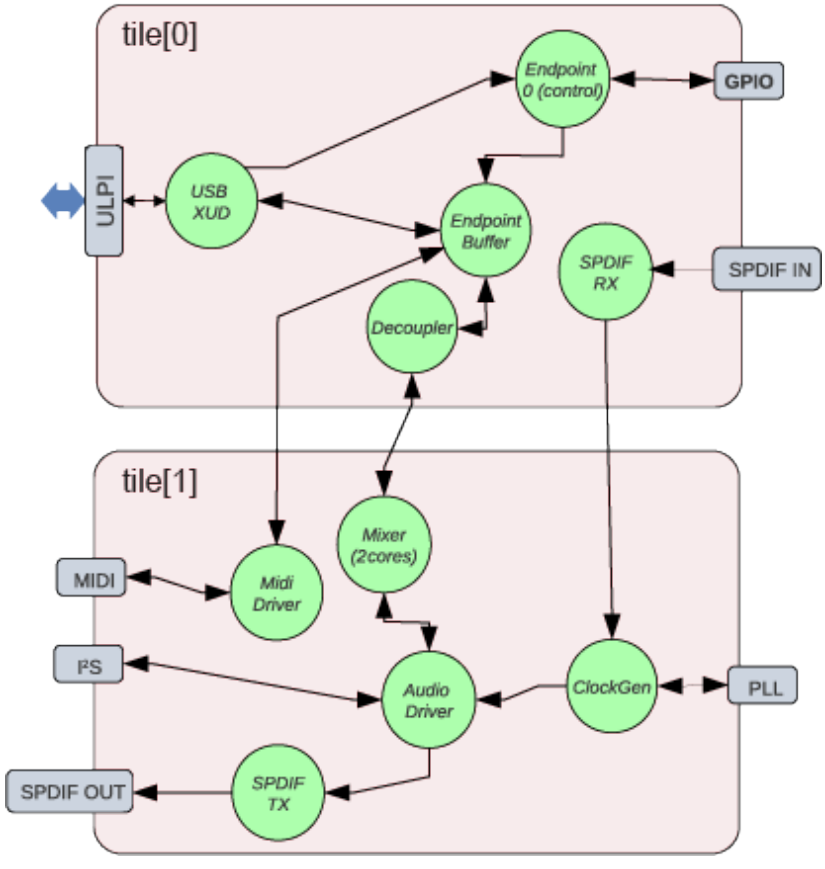

Fig. 2: *xcore-200* Multichannel Audio system/task diagram

The `app_usb_aud_xk_216_mc` application uses the functions provided in `lib_board_support` for master clock generation and audio hardware configuration. The functions `xk_audio_216_mc_ab_AudioHwInit()` and `xk_audio_216_mc_ab_AudioHwConfig()` are called at various points during initialisation and runtime to initialise and configure the audio hardware.

For further details on the hardware platform and the functions available for configuring it refer to lib_board_support documentation.

Audio hardware

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 clock.

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

Table 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*:

Table 11: Master Clock Frequency Select

Value	Frequency
0	24.576MHz
1	22.579MHz

DAC and ADC 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 the ADC takes place over I²C.

Configuring audio hardware

All of the external audio hardware is configured using `lib_board_support`.

> **Note**
>
> `lib_board_support` has the I²C library (lib_i2c) in its dependency list.

The hardware targeted is the *XMOS XU216 Multichannel Audio board* (*XK-AUDIO-216-MC*). The functions `xk_audio_216_mc_ab_AudioHwInit()` and `xk_audio_216_mc_ab_AudioHwConfig()` are called at various points during initialisation and runtime to initialise and configure the audio hardware.

The audio hardware configuration is set in the `config` structure of type `xk_audio_216_mc_ab_config_t` which is passed to the `xk_audio_216_mc_ab_AudioHwInit()` and `xk_audio_216_mc_ab_AudioHwConfig()` functions.

```
static const xk_audio_216_mc_ab_config_t config = {
    // clk_mode
    CLK_MODE,

    // codec_is_clk_master
    CODEC_MASTER,

    // usb_sel
    (USB_SEL_A ? AUD_216_USB_A : AUD_216_USB_B),

    // pcm_format
    XUA_PCM_FORMAT,

    // pll_sync_freq
    PLL_SYNC_FREQ
};
```

The *AudioHwInit()* function is implemented to make a call to the `lib_board_support` function `xk_audio_216_mc_ab_AudioHwInit()` to power up and initialise the audio hardware ready for a configuration.

The *AudioHwConfig()* function configures the audio hardware post initialisation. It is called each time a sample rate or stream format change occurs. It is implemented to make a call to the `lib_board_support` function `xk_audio_216_mc_ab_AudioHwConfig()`.

For further details on the hardware platform and the functions available for configuring it refer to lib_board_support documentation.

Validated build options

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

The design has only been fully validated against the build options as set in the application as distributed in the CMakeLists.txt. See *Build configurations* for details and general information on build configuation naming scheme.

These fully validated build configurations are enumerated in the supplied CMakeLists.txt.

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 CMakeLists.txt is shown in Table 12.

Table 12: 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. The *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 USB Audio API reference

7.1 Configuration defines

An application using the USB audio framework provided by `lib_xua` needs to be configured via defines. 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 `CMakeLists.txt` for the relevant build configuration.

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

Code location (tile)

XUA_AUDIO_IO_TILE_NUM

Location (tile) of audio I/O.

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

XUA_XUD_TILE_NUM

Location (tile) of audio I/O.

The tile number to run lib_xud tasks on.

XUA_MIDI_TILE_NUM

Location (tile) of MIDI I/O.

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

XUA_PLL_REF_TILE_NUM

Location (tile) of reference signal to CS2100.

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

XUA_SPDIF_TX_TILE_NUM

Location (tile) of SPDIF Tx.

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

Channel counts

NUM_USB_CHAN_OUT

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

NUM_USB_CHAN_IN

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

I2S_CHANS_DAC

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

I2S_CHANS_ADC

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

DSD_CHANS_DAC

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

Frequencies and clocks

MAX_FREQ

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

MIN_FREQ

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

MCLK_441

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

MCLK_48

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

Audio Class

AUDIO_CLASS

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

System feature configuration

MIDI

MIDI

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

MIDI_RX_PORT_WIDTH

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

S/PDIF

XUA_SPDIF_TX_EN

Enables SPDIF Tx. Default: 0 (Disabled)

SPDIF_TX_INDEX

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

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

XUA_SPDIF_RX_EN

Enables SPDIF Rx. Default: 0 (Disabled)

SPDIF_RX_INDEX

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

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

ADAT

XUA_ADAT_TX_EN

Enables ADAT Tx. Default: 0 (Disabled)

ADAT_TX_INDEX

Defines which output channels (8) should be output on ADAT. Note, Output channels indexed from 0.

Default: 0 (i.e. channels [0:7])

XUA_ADAT_RX_EN

Enables ADAT Rx. Default: 0 (Disabled)

ADAT_RX_INDEX

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

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

PDM Microphones

XUA_NUM_PDM_MICS

Number of PDM microphones in the design.

Default: 0

DFU

XUA_DFU_EN

Enable DFU functionality.

Default: 1 (Enabled)

HID

HID_CONTROLS

Enable HID playback controls functionality.

1 for enabled, 0 for disabled.

Default 0 (Disabled)

CODEC Interface

CODEC_MASTER

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

0: XMOS is I2S master. 1: CODEC is I2s master.

Default: 0 (XMOS is master)

USB device configuration

VENDOR_STR

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

Default: "XMOS"

VENDOR_ID

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

Default: 0x20B1 (XMOS)

PRODUCT_STR

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

Default: Undefined

PRODUCT_STR_A2

Product string for Audio Class 2.0 mode.

Default: "XMOS xCORE (UAC2.0)"

PRODUCT_STR_A1

Product string for Audio Class 1.0 mode.

Default: "XMOS xCORE (UAC1.0)"

PID_AUDIO_1

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

Default: 0x0003

PID_AUDIO_2

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

Default: 0x0002

BCD_DEVICE

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

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

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

Volume control

OUTPUT_VOLUME_CONTROL

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

Default: 1 (Enabled)

INPUT_VOLUME_CONTROL

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

Default: 1 (Enabled)

Mixing parameters

MIXER

Enable "mixer" core.

Default: 0 (Disabled)

MAX_MIX_COUNT

Number of seperate mixes to perform.

Default: 8 if MIXER enabled, else 0

MIX_INPUTS

Number of channels input into the mixer.

Note, total number of mixer nodes is MIX_INPUTS * MAX_MIX_COUNT

Default: 18

Power

XUA_POWERMODE

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

Default: XUA_POWERMODE_BUS

XUA_CHAN_BUFF_CTRL

Enable power saving feature in XUA_Buffer_Decouple()

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

XUA_LOW_POWER_NON_STREAMING

Enable power saving when device is enumerated but audio in not currently streaming.

If set to 1 then transitions to ALT interface 0 (streaming stopped) will cause AudioHub to cease looping and no-longer driver the I2S/TDM lines. In addition, the callback AudioHwShutdown() is called which allows the user to run any specific code (eg. CODEC power-down and/or disable master clock) to further reduce system power in this state. AudioHwInit() and AudioHwConfig() will always be called again prior to USB audio streaming. The transition to the low power state will only occur when *both* input *and* output interfaces are not streaming. As soon as either input or output streaming starts then audiohub is restarted.

If set to zero or undefined (default behaviour) then AudioHub will always continue looping even when audio streaming stops. This behaviour may be preferable in applications where frequent initialisation of the mixed signal hardware is undesirable, where other parts of the system rely on I2S clocks being conitinuously available or in MI (musical instrument) applications where functions such as mixer need to continuously operate regardless of USB streaming state.

7.2 User function definitions

The following functions can be optionally defined by an application to override default (empty) implementations in `lib_xua`.

External audio hardware configuration

The functions listed below should be implemented to configure external audio hardware.

void **AudioHwInit**(void)

User audio hardware initialisation code.

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

void **AudioHwConfig**(
unsigned samFreq,unsigned mClk,unsigned dsdMode,unsigned sampRes_DAC,unsigned sampRes_ADC,
)

User audio hardware configuration code.

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

Parameters

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

void **AudioHwConfig_Mute**(void)

User code mute audio hardware.

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

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

void **AudioHwConfig_UnMute**(void)

User code to un-mute audio hardware.

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

void **AudioHwShutdown**(void)

User audio hardware de-initialisation code.

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

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

Audio streaming notification

The functions listed below can be useful for mute lines, indication LEDs etc.

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

User stream start code.

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

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

HID controls

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

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

Get the data for the next HID Report.

Parameters

- **id** – **[in]** The HID Report ID (see 5.6, 6.2.2.7, 8.1 and 8.2 of the USB Device Class Definition for HID 1.11) Set to zero if the application provides only one HID Report which does not include a Report ID
- **hidData** – **[out]** The HID data If using Report IDs, this function places the Report ID in the first element; otherwise the first element holds the first byte of HID event data.

Return values

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

Returns

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

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 *Configuration defines*

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 192 kHz 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.