



# AN02014: Integrating DSP into the XMOS USB reference design

Publication Date: 2025/4/28

Document Number: XM-015104-AN v2.0.0

## IN THIS DOCUMENT

1	Introduction . . . . .	1
2	Getting Started . . . . .	1
3	Application Overview . . . . .	3
4	Implementing <i>UserBufferManagement</i> . . . . .	4
5	Generating the DSP pipeline . . . . .	5
6	References . . . . .	6
7	Support . . . . .	6

## 1 Introduction

**Note:** Some software components in this tool flow are prototypes and will be updated in Version 2 of the library. The underlying Digital Signal Processing (DSP) blocks are however fully functional. Future updates will enhance the features and flexibility of the design tool.

This application note explains how to integrate custom DSP into the XMOS USB Audio Reference Design ([sw\\_usb\\_audio](#)) by utilising the XMOS Audio DSP solution ([lib\\_audio\\_dsp](#)). An application showcasing this integration accompanies this document and the steps taken to create that application are described below.

The XMOS audio DSP solution offers a low-code approach to designing custom audio DSP through a Python library, which generates a multithreaded, pipelined DSP process for the xcore. The XMOS USB Audio Reference Design serves as a highly configurable audio IO platform. By following the process outlined in this document, it is possible to create highly specific audio DSP applications with less than 100 lines of additional embedded source code.

Both of the repositories discussed above contain detailed documentation on their use which should be consulted when modifying this application to any specific product. The scope of this note is limited to a single configuration of the XMOS USB Audio Reference Design and a simple DSP pipeline. It explains the steps required to add DSP to any configuration of the XMOS USB Audio Reference Design.

This app note is part one of two that discusses adding DSP to a USB application. Part two, AN02015, covers adding run-time control and includes DSP pipeline that showcases more of the DSP library's features.

## 2 Getting Started

### 2.1 Requirements

Before running this application note ensure the following applications are installed on your system:

- [XTC 15.3.1](#)



- ▶ [CMake](#) >= 3.21.0
- ▶ [Python](#) 3.12
- ▶ [Graphviz](#), ensuring the `dot` executable available on your PATH.

The following hardware is required:

- ▶ [XK-AUDIO-316-MC-AB](#)
- ▶ 2 Micro-USB cables
- ▶ Optional, 1 to 4 3.5 mm stereo audio cables.

## 2.2 Running the example

First, connect the [XK-AUDIO-316-MC-AB](#) to your computer with both the “DEBUG” and “USB DEVICE” Micro-USB ports as shown in [Fig. 1](#).

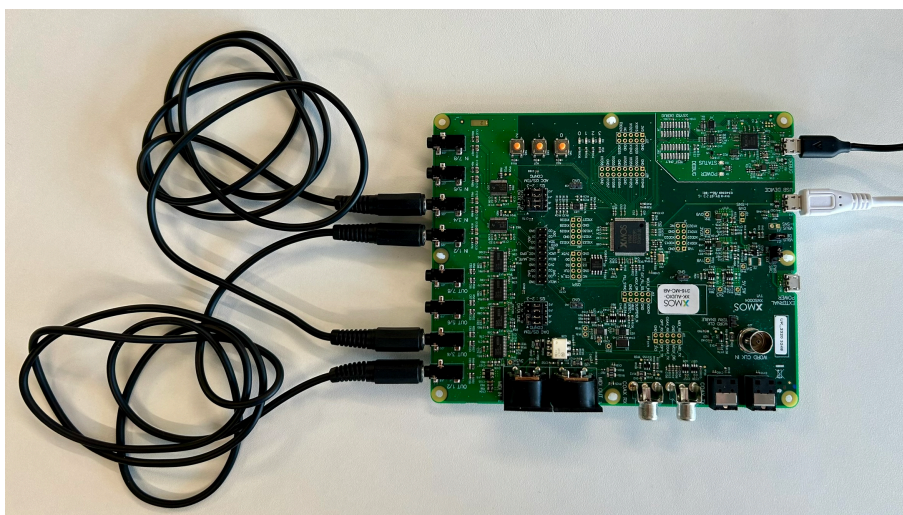


Fig. 1: XK-AUDIO-316-MC-AB with USB cables and 3.5 mm jack cables connected.

Once connected follow these steps:

1. Open a terminal and activate the [XTC](#) environment. Optionally, create a [Python virtual environment](#) and activate it.
2. Get the source code for this app note from <https://www.xmos.com/application-notes/>
3. Navigate to the root directory of this app note and install the [Python](#) requirements:

```
pip install -Ur requirements.txt

# for users in China, a pip mirror will give faster downloads
pip install -Ur requirements.txt -i https://pypi.tuna.tsinghua.edu.cn/simple
```

4. Start the [Jupyter](#) notebook from the `app_an02014` directory. [Jupyter](#) Notebook is an interactive Python editor which was installed via the pip command in the previous step.

```
cd app_an02014
jupyter notebook
```

5. If this does not automatically open a browser window, then copy the url from the output of *jupyter* that starts with *http://127.0.0.1* and navigate to it in your web browser.
6. Open *dsp.ipynb* on the web interface by double clicking on the file name.
7. Execute all the cells in the notebook by selecting “Run all cells” from the “Run” menu.

This final step will display a diagram that represents the provided simple DSP pipeline. It will then generate the xcore source code, build the 8-channel application, and run it on the connected device. The device will appear on the connected computer as an 8-input, 8-output USB audio device named “XMOS xCORE.ai MC (UAC2.0)”. The provided DSP design simply reduces the input signal by 6 dB in both directions. When audio is played through the USB device, the signal will be processed by the DSP and then output through the 3.5 mm jacks on the board.

The DSP pipeline in *dsp.ipynb* provides a template that can be adapted to other needs and can form the base for the user's own specific application requirements. The notebook can be updated and rerun to try different designs and iterate rapidly to find the best solution.

### 3 Application Overview

The XMOS USB Audio 2.0 Reference Design, [sw\\_usb\\_audio](#), offers a versatile infrastructure for transmitting audio between various audio interfaces, including USB, I<sup>2</sup>S, ADAT, and SPDIF. It's highly adaptable, supporting diverse combinations of input and output interfaces. Although the given configurations will transfer audio between interfaces without modification (excluding its built-in mixer module), it does offer callback hooks for application code to intercept the data flow. The pertinent callback, *UserBufferManagement*, is invoked once per sample period with the most recent data from each interface.

The core functionality of *sw\_usb\_audio* is provided by [lib\\_xua](#); this is also the case for the application associated with this note. [lib\\_xua](#) provides the ability to configure which features are enabled and which tiles different features run on. The ports available in the end product will significantly effect the choice of tile for each feature. Consult the [sw\\_usb\\_audio](#) documents for more details. For this example application the chosen values for *XUD\_TILE* and *AUDIO\_IO\_TILE* are 0 and 1 respectively. This leads to the software structure shown in [Fig. 2](#).

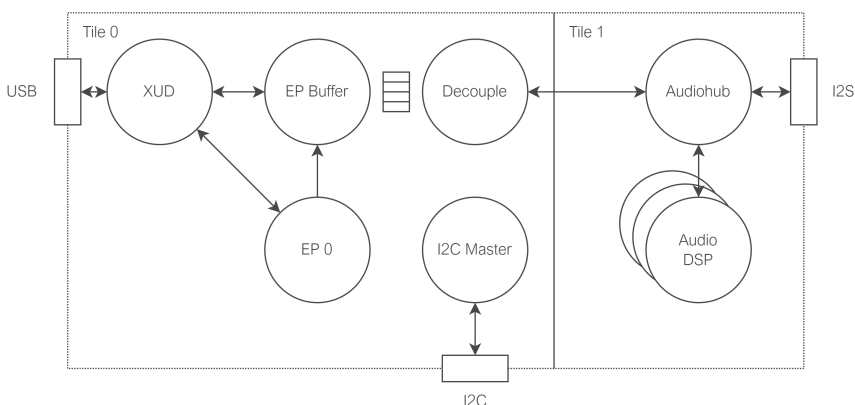


Fig. 2: System thread diagram

The *UserBufferManagement* callback is executed on the audiohub thread on tile 1. This is the only thread in use on that tile, so there are 7 threads left for new functionality. It is

important to understand the thread usage of the tile that will execute the DSP in order to know how many threads of DSP the pipeline design may use.

To reduce the complexity of this application, there is one provided build configuration and support for a single board. This has facilitated the removal of most files present in [sw\\_usb\\_audio](#). However, the correlation between the remaining source files and those found in [sw\\_usb\\_audio](#) should be evident. All changes discussed in this document can also be applied to a full [sw\\_usb\\_audio](#) based design to integrate the generated pipeline.

This application has been configured to operate at a single sample rate of 48 kHz (using [lib\\_xua](#) configuration macros). The DSP pipeline must be designed for a fixed sample rate to allow tuning parameters to be determined. [lib\\_audio\\_dsp](#) supports rates other than 48 kHz, but the rate set when designing the DSP must align with the rate set for the rest of the application to get the expected performance.

## 4 Implementing UserBufferManagement

This application note includes a modified version of the *UserBufferManagement* function normally found in [sw\\_usb\\_audio](#). It has the following prototype:

```
void UserBufferManagement(unsigned* sampsFromUsbToAudio, unsigned* sampsFromAudioToUsb);
```

The two arguments contain the input samples and must be filled with the output samples. The first argument takes channels from the USB host as an input and outputs to the other interfaces. The second argument outputs to the USB host and takes channels from the other audio interfaces as an input.

To pass audio through the DSP pipeline two functions must be called. These are defined in *adsp\_pipeline.h* from [lib\\_audio\\_dsp](#):

```
static inline void adsp_pipeline_source(adsp_pipeline_t *adsp, int32_t **data);
static inline void adsp_pipeline_sink(adsp_pipeline_t *adsp, int32_t **data);
```

The first passes samples to the pipeline, and the second reads processed samples from the pipeline. Both of these use channels stored within an instance of *adsp\_pipeline\_t* that must be initialised by functions from the generated pipeline. Both *adsp\_pipeline\_source* and *adsp\_pipeline\_sink* block on a channel until the DSP threads are available to process the sample exchange. This can lead to issues if the generated DSP cannot meet the real time requirements of the system.

The *data* parameter expects an array containing frames of samples for each input and output channel. For this application the frame size will be 1; therefore, we can construct both data parameters by initialising a new array of pointers that reference the correct elements in *sampsFromUsbToAudio* and *sampsFromAudioToUsb*. [Fig. 3](#) shows how the *dsp\_input* and *dsp\_output* arrays are constructed in this application.

It is important to note that the size of *sampsFromUsbToAudio* and *sampsFromAudioToUsb* depend on the application configuration of [lib\\_xua](#). In this application there are 8 USB OUT channels and 8 ADC channels, totalling 16 DSP inputs. There are also 8 USB IN channels and 8 DAC channels, totalling 16 channels of DSP outputs. The provided application will adapt to different I<sup>2</sup>S and USB configurations but will need updating when other [lib\\_xua](#) interfaces are enabled.

It is also important to note that the channel indices in *dsp\_input* and *dsp\_output* will be used later when defining the DSP pipeline.

Once constructed, the *dsp\_input* can be passed into the “adsp\_pipeline\_source” function, and the *dsp\_output* can be passed to the “adsp\_pipeline\_sink” function. An example implementation of *UserBufferManagement* can be found in *app\_dsp.c* from the application provided with this note. This includes an example *UserBufferManagement* for frame sizes greater than 1, where additional buffering is required before calling “adsp\_pipeline\_source” and “adsp\_pipeline\_sink”.

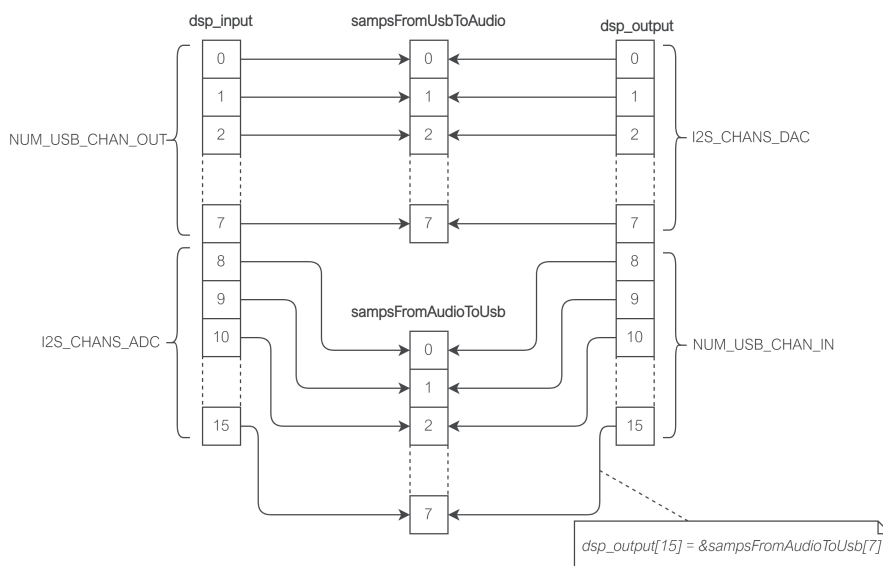


Fig. 3: Mapping the DSP input/output arrays to the *UserBufferManagement* arguments

## 5 Generating the DSP pipeline

Adding DSP to the project requires an initial DSP design, which is best done in a [Jupyter](#) notebook. This requires [Python](#) and some [Python](#) packages, which the application specifies in *requirements.txt*. This application's source directory contains the notebook used at *app\_an02014/dsp.ipynb*. The [Jupyter documentation](#) covers the details of creating, modifying, and executing a [Jupyter](#) notebook.

Upon opening *dsp.ipynb*, you will find a simple DSP design that processes the 16 pipeline inputs. The design of a DSP pipeline is covered thoroughly in the user guide associated with [lib\\_audio\\_dsp](#). The indices of the pipeline inputs match the indices of the channels in the *dsp\_input* array discussed in the [Implementing UserBufferManagement](#). The outputs of the pipeline, set when *pipeline.set\_outputs()* is called, have indices that align with the *dsp\_output* array.

After defining the DSP pipeline the notebook will proceed to generate the xcore source code:

```
generate_dsp_main(pipeline, out_dir="src/generated_dsp")
```

This function takes the *pipeline* and generates the source code in the provided *out\_dir*, relative to the parent folder of *dsp.ipynb*. Consequently, this action creates the following files:

```
app_an02014/src/generated_dsp
  adsp_generated_auto.c
  adsp_generated_auto.h
  adsp_instance_id_auto.h
```

To include these files in the build, the *CMakeLists.txt* has been updated. No change was required to add the C files as [XCommon CMake](#) will automatically find them. In order for the application to include the header files, *APP\_INCLUDES* has been appended with *src/generated\_dsp* (see *app\_an02014/CMakeLists.txt* for this change).

With the generated files included in the build it is possible to start the DSP threads in the application. The following function is defined in *app\_dsp.c* and called in *user\_main.h* on tile 1.

```
void dsp_thread(void) {  
    // Initialise the DSP instance and enter the generated DSP main function.  
    // This will never return.  
    m_dsp = adsp_auto_pipeline_init();  
    adsp_auto_pipeline_main(m_dsp);  
}
```

With these changes, the application can be run on the board. The supplied [Jupyter](#) notebook will automatically do this as the final step of execution.

The custom DSP application is now ready for the development of a more complex DSP pipeline, such as the example described in AN02015.

## 6 References

- ▶ [sw\\_usb\\_audio](#)
- ▶ [lib\\_audio\\_dsp](#)
- ▶ [lib\\_xua](#)
- ▶ [XTC](#)
- ▶ [XK-AUDIO-316-MC-AB](#)
- ▶ [XCommon CMake](#)
- ▶ [Jupyter](#)
- ▶ [Graphviz](#)

## 7 Support

For all support issues please visit <http://www.xmos.com/support>



Copyright © 2025, All Rights Reserved.

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

XMOS, XCORE, VocalFusion and the XMOS logo are registered trademarks of XMOS Ltd. in the United Kingdom and other countries and may not be used without written permission. Company and product names mentioned in this document are the trademarks or registered trademarks of their respective owners.

