

AN02031: Live Streaming Sound Card Example

Publication Date: 2025/4/3

Document Number: XM-015221-AN v1.0.0

IN THIS DOCUMENT

| | | |
|---|---|---|
| 1 | Introduction | 1 |
| 2 | Setup | 3 |
| 3 | Running the GUI | 5 |
| 4 | Saving, Loading and Code Generation | 9 |
| 5 | Building the App from Source | 9 |
| 6 | Updating the GUI or Modifying the DSP | 9 |

1 Introduction

This application note demonstrates how to use `lib_audio_dsp` to create a live streaming sound card application. The device captures sound from both a USB source and a microphone, outputting through USB and analog outputs. The sound is processed through a series of Digital Signal Processing (DSP) stages, which can be tuned on the host machine. The application includes a Graphical User Interface (GUI) that allow these adjustments.

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.

The sound card features the following DSP stages:

- ▶ reverb
- ▶ denoising
- ▶ ducking
- ▶ and EQ.

It could be used for applications such as:

- ▶ Live streaming to social media
- ▶ Podcasting
- ▶ Gaming
- ▶ General purpose headset

The application has the following inputs and outputs:

- ▶ Stereo analogue input microphone
- ▶ Stereo input over USB
- ▶ Stereo analogue output for headphones/speakers
- ▶ Stereo output over USB for livestreaming/recording.

The DSP pipeline is shown below:

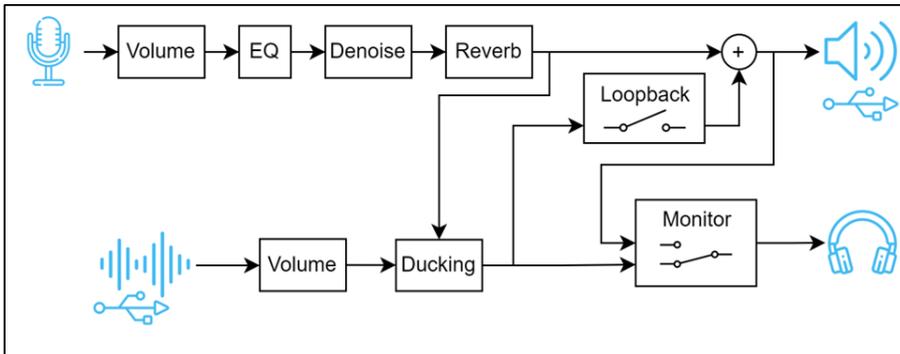


Fig. 1: DSP Pipeline

The audio performance of the demo is:

- ▶ 48 kHz sampling rate
- ▶ 6 sample latency (0.125 ms), excluding ADC & DAC filter (56 samples/ 1.15 ms latency including filters)

2 Setup

This section guides the user through installing all the necessary components for setting up the application, from both a software and hardware point of view.

2.1 Prerequisites

The following are required before setting up the application:

- ▶ Python 3.10 (recommended) or above.
- ▶ Windows PC.
- ▶ [XMOS Multichannel Audio Evaluation Kit \(XMOS MCAB\)](#).

2.2 Hardware Setup

Connect the XMOS MCAB to the host computer using the **USB DEVICE** port. For flashing the board, also connect the **DEBUG** port. The **DEBUG** port can be disconnected once the device has been flashed.

Connect a microphone to the **IN 1/2** jack, and headphones to the **OUT 1/2** jack.

Note: The microphone input is a stereo TRS jack. This means a balanced mic should not be used, as the stereo inputs are summed, and for a balanced signal this results in total signal cancellation.

Note: A low-impedance microphone or one with a powered preamplifier is recommended to ensure a strong signal for processing. Otherwise, the signal may be too weak for proper handling. Alternatively, users can manually add a fixed gain stage near the beginning of the pipeline.

See the [Fig. 2](#) for an example hardware setup.



Fig. 2: USB Multichannel Audio Evaluation Kit Setup

2.3 Software Setup

A compiled binary is provided, and the device can be flashed from a XMOS XTC command prompt using:

```
xf1ash app_an02031\bin\app_an02031.exe
```

To build from source, refer to the section [Saving, Loading and Code Generation](#).

2.4 Installing the *libusb* Driver on Windows

The first time the device is used on Windows, the *libusb* driver must be installed. This is done using the third-party tool *Zadig*.

These steps are only required once and they must be executed while the firmware is running on the device.

1. Open *Zadig* and select *XMOS Control (Interface 3)* from the list of devices. If the device is not present, ensure *Options -> List All Devices* is checked.
2. Select *libusb-win32* from the list of drivers in the right hand spin box as shown in [Fig. 3](#).
3. Click the *Install Driver* button and wait for the installation to complete.

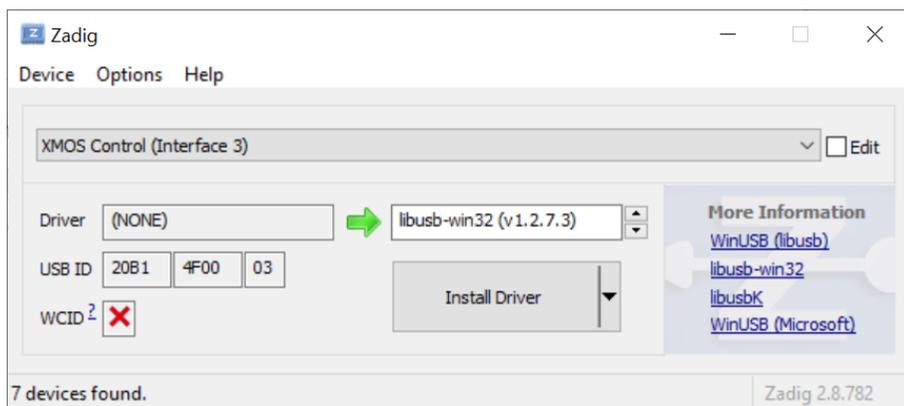


Fig. 3: Selecting the *libusb-win32* driver in Zadig for the Control Interface

3 Running the GUI

3.1 Initial Use

In Windows Explorer, double click on `tuning.cmd`. This will initialise the Python virtual environment and install requirements. Once this is done it will open the GUI.

Note: Customers in China may wish to use `tuning_cn.cmd` instead. This will use <https://pypi.tuna.tsinghua.edu.cn/simple> for retrieving Python packages instead of <https://pypi.python.org/simple>, which may give increased download speeds.

The GUI has 4 tabs described below.

3.2 End Customer Interface

These are the controls exposed to the final customer. In this implementation, USB control commands are used over EP0. The same control logic could alternatively be implemented via GPIO.

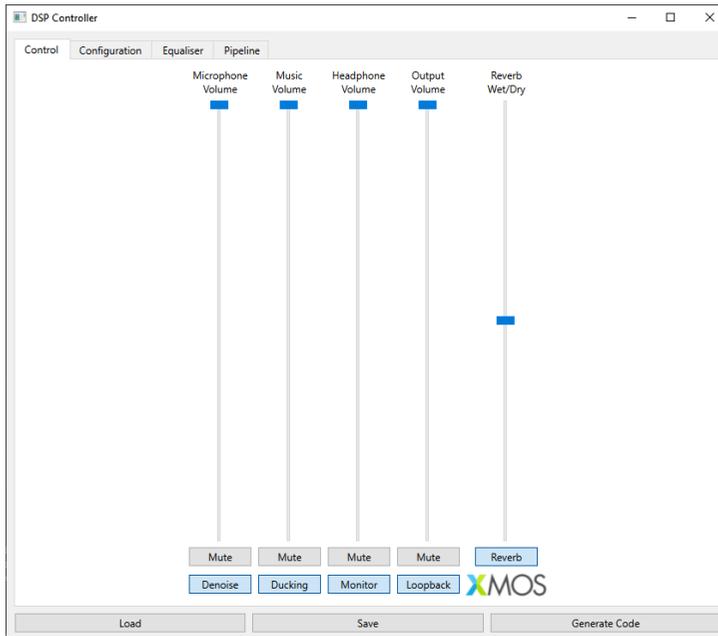


Fig. 4: GUI Control Tab

The controls perform the following functions:

- ▶ Volume sliders: control the volume of the input/output with optional mute.
- ▶ Reverb slider: control wet/dry mix of the reverb.
- ▶ Reverb button: enable/disable the reverb.
- ▶ Denoise button: enable noise suppression on the microphone.
- ▶ Ducking button: reduce the volume of the music when there is signal on the microphone.
- ▶ Monitor button: when on, send the same output to the USB and headphone outputs. When off, do not send the microphone output to the headphones.
- ▶ Loopback button: when on, send the USB input back to the USB output. Otherwise just send the microphone signal over USB.

3.3 Algorithm Tuning Parameters

This tab exposes the lower level tuning parameters to a DSP product engineer for tuning before final production.

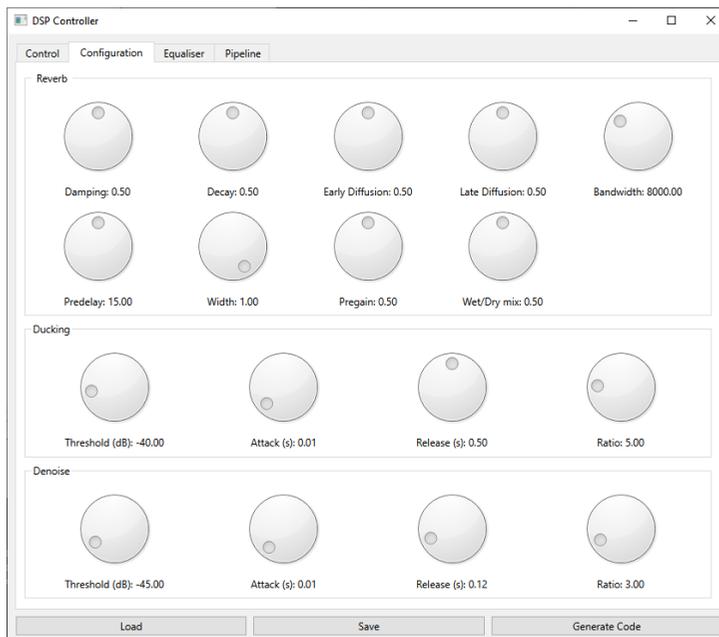


Fig. 5: GUI Configuration Tab

3.4 Parametric EQ

This tab shows the EQ being applied to the microphone signal. The type and parameters of each biquad can be set, and the frequency response of the cascade is calculated from it.

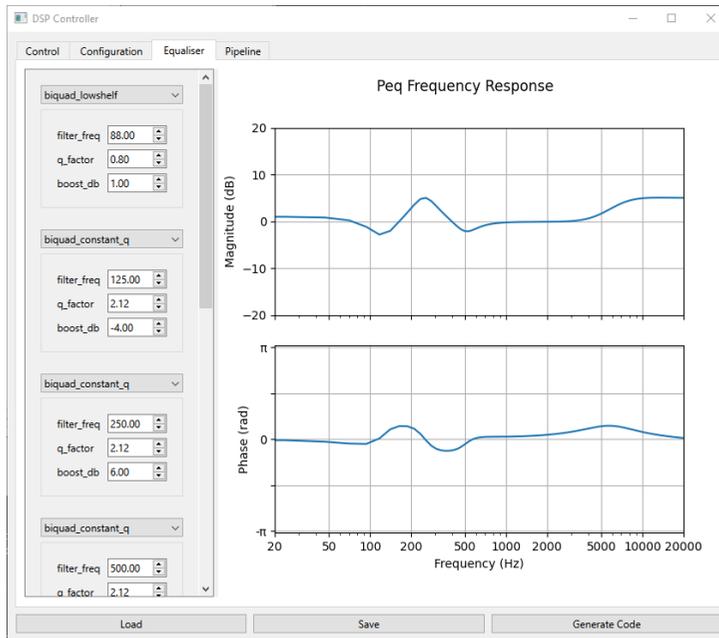


Fig. 6: GUI Equaliser Tab

3.5 Pipeline Graphic

This tab contains an image of the DSP pipeline.

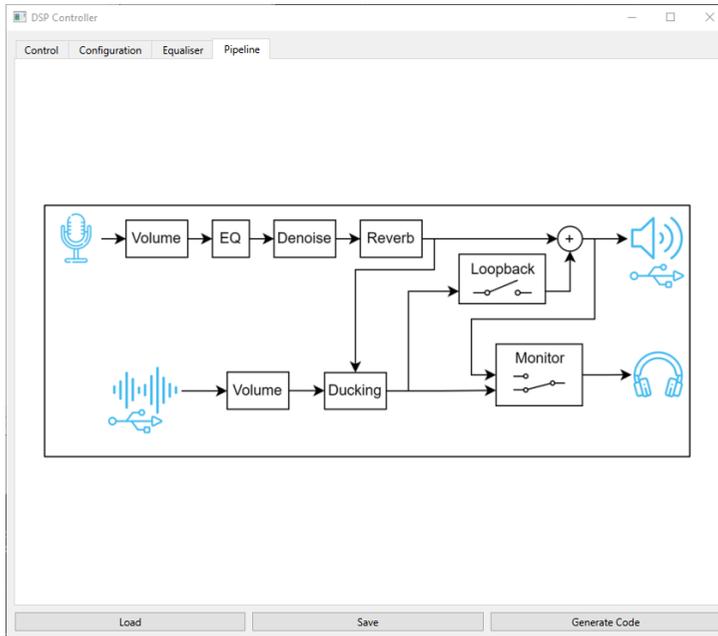


Fig. 7: GUI Pipeline Tab

4 Saving, Loading and Code Generation

Once the parameters have been tuned to the desired values, they can be saved as a JSON file by clicking the “Save” button. Other sets of parameters can be loaded from the JSON file by using the “Load” button. During loading, the saved checksum in the JSON file is compared against the checksum generated from the pipeline defined in `pipeline.py`. If they do not match, a warning is given to indicate that the pipelines differ. The loaded JSON file may still be compatible with the GUI if the exposed tuning parameters defined in `params.py` have remained the same, and the DSP stages they interact with still exist. If the loaded JSON file is not compatible with the current version of the GUI an error is given. This usually means that `pipeline.py` and `params.py` have been changed, and a different version of the GUI needs to be used.

Once the parameters are fully configured, the autogenerated C should be updated by clicking the “Generate Code” button in the GUI. By default this selects the folder `app_an02031\src\generated_dsp`. The application can then be rebuilt by following the steps in [Building the App from Source](#).

Alternatively the code can be generated from the command line, see the steps in [Updating the DSP Pipeline](#). This may be useful for regenerating the code as part of a CMake step.

5 Building the App from Source

The app is supplied with a full sandbox and compiled app. Modifications may require rebuilding, which can be done as described below.

In an XTC Tools 15.3 prompt, navigate to the app note directory, then create and activate a Python virtual environment, and install `lib_audio_dsp`:

```
python -m venv .venv
.venv\Scripts\activate
pip install -e lib_audio_dsp/python
```

Then run Xcommon Cmake to initialise the sandbox:

```
cd an02031
cmake -G "Unix Makefiles" -B build
```

To build the app, run:

```
cmake --build build
```

Finally, to run it:

```
xrun app_an02031\bin\app_an02031.xe
# Or, to permanently flash it
xflash app_an02031\bin\app_an02031.xe
```

6 Updating the GUI or Modifying the DSP

This application note provides a fixed DSP pipeline and a GUI designed to interact with it. Users may wish to modify or extend the provided DSP pipeline and correspondingly update the GUI. The following sections detail the procedures for accomplishing these tasks.

6.1 Updating the DSP Pipeline

Should it be necessary, the first step involves updating the DSP pipeline. The pipeline is defined in the file `tuning_utility\pipeline.py` and utilizes the `lib_audio_dsp` Python library to generate a pipeline. The library includes its own documentation, which can be consulted to implement any required modifications.

Prior to generating sources, it may be necessary to update the parameter model, which is defined in `tuning_utility\params.py`. This file defines a class named `Params`



that contains the tuning data for the named stages in the pipeline. Initially, “tuning_gui.py” will invoke `pipeline()` to create an instance of `Pipeline`. Subsequently, it will instantiate `Params` and invoke the function `update_from_tuning_params` from “pipeline.py” to adjust the pipeline with the tuning parameters.

When updating the pipeline, it is likely the checksum will change. This may give errors when trying to reuse a `tuning.json` file. Removing or renaming the default `tuning.json` should resolve most issues. Note that if the pipeline is different, but the `params.py` is the same, the new pipeline may be compatible with the GUI despite the pipeline difference.

To regenerate the sources for the updated pipeline, create a Python virtual environment and install the required dependencies specified in “requirements.txt” using pip. Then, execute the following command to generate the code for the default JSON file:

```
python tuning_gui.py --code-gen
```

To use a different JSON file, the path to the file can be passed in:

```
python tuning_gui.py --code-gen path/to/tuning.json
```

This command will update the generated DSP sources. Subsequently, use CMake to compile the new application in the usual manner.

6.2 Updating the GUI

This section addresses the addition or removal of widgets in the GUI. The GUI is a straightforward Python application utilizing PySide, a Python binding for the Qt framework. The GUI is defined in “tuning_gui.py”.

Within “tuning_gui.py”, the `DSPWindow` class is defined. This class is a `QMainWindow` to which all widgets are added. The `DSPWindow` class contains a `state` attribute that tracks the current parameter values. All widgets must update this state through one of its update methods. To ensure the widget remains current when the value it controls may be updated elsewhere, it should connect to the `STATE_CHANGED` signal. This signal is emitted whenever any parameters change.



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.

