lib_audio_dsp: Audio DSP Library

Publication Date: 2025/4/19
Document Number: XM-015103-UG v1.3.0

XMOS

IN THIS DOCUMENT

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.

lib_audio_dsp is a DSP library for the XMOS xcore architecture. It facilitates the creation of multithreaded audio DSP pipelines that efficiently utilise the xcore architecture.

The library is built around a set of DSP function blocks, referred to in the documentation as *Stages*, which have a consistent API and can be combined to create many different designs.

A tool for easily combining stages into a custom DSP pipeline is provided. DSP pipeline parameters can be adjusted and tuned on the fly via a PC based tuning interface, and utilities for hardware controls are also provided.

lib_audio_dsp includes common signal processing functions optimised for the xcore, such as:

- biquads and FIR filters.
- compressors, limiters, noise gates and envelope detectors.
- adders, subtractors, gains, volume controls and mixers.
- delays and reverb.

These can be combined together to make complex audio pipelines for many different applications, such as home audio, music production, voice processing, and AI feature extraction.

This document covers the following topics:

1. *Tool User Guide*: A beginner's guide to installing and using the DSP design and generation Python library.
2. *Design Guide*: Advanced guidance on designing and debugging generated DSP pipelines.
3. *DSP Components*: List of all DSP components and details on the backend implementation.
4. *Run-Time Control User Guide*: Basic guide to add time control to a DSP application.
5. *API Reference*: References to DSP components, control and integration and high-level tool desing API.

The subsequent sections provide comprehensive insights into the functionalities and applications of lib_audio_dsp, detailing how to leverage its features for efficient audio signal processing.

For example appliations, see the Application Notes related to lib_audio_dsp on the XMOS website.

1 Tool User Guide

This section introduces the *audio_dsp* Python library, and how to use it to generate multithreaded DSP pipelines for the xcore.

The following sections provide guidance on preparing the environment for the library, building a pipeline, and exploring the API documentation.

Note: For a quick start, the Application Notes related to lib_audio_dsp on the XMOS website come preconfigured with a complete application and sandbox.

1.1 Setup

This section describes the requirements and the steps to run a basic pipeline. This document lists the necessary steps for both Windows and Linux/macOS. This section uses the *app_simple_audio_dsp_integration* example found within this repository. The steps will be broadly similar for any user-created project.

Note: Copying multiple lines into the console may not work as expected on Windows. To avoid issues, copy and execute each line individually.

Hardware Requirements

- xcore.ai evaluation board (XK-EVK-XU316 or XK-316-AUDIO-MC-AB)
- xTag debugger and cable
- 2x Micro USB cable (one for power supply and one for the xTag)

Software Requirements

- XTC tools: 15.3.1.
- Graphviz: this software must be installed and the `dot` executable must be on the system path.
- Python: 3.12 or later.
- CMAKE: 3.21 or later.

Additionally, on Windows the following is required:

- ninja-build

Setup Steps

Note: All the steps below are executed from the sandbox folder created in the second step.

---

1. Prepare the development environment

   On Windows:

   1. Open the Command Prompt or other terminal application of choice

2. Activate the XTC environment:

```
call "C:\Program Files\XMOS\XTC\15.3.1\SetEnv.bat"
```

On Linux and macOS:

1. Open a terminal
2. Activate the XTC environment using *SetEnv*

```
source /path/to/xtc/tools/SetEnv
```

2. Create a sandbox folder with the command below:

```
mkdir lib_audio_dsp_sandbox
```

3. Clone the library inside *lib_audio_dsp_sandbox* using SSH (if you have shared your keys with Github) or HTTPS:

```
cd lib_audio_dsp_sandbox

# with SSH
git clone git@github.com:xmos/lib_audio_dsp.git

# without SSH
git clone https://github.com/xmos/lib_audio_dsp.git
```

For troubleshooting SSH issues, please see this Github guide.

4. Get the lib_audio_dsp library dependencies inside *lib_audio_dsp_sandbox*. This step can take several minutes.

On Windows:

```
cd lib_audio_dsp/examples/app_simple_audio_dsp_integration
cmake -B build -G Ninja
cd ../../..
```

On Linux and macOS:

```
cd lib_audio_dsp/examples/app_simple_audio_dsp_integration
cmake -B build
cd ../../..
```

5. Create a Python virtualenv inside *lib_audio_dsp_sandbox*, and install lib_audio_dsp and it's requirements.

Note: Make sure to use the same Python version as the the recommended in the *Software Requirements* section.

On Windows:

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

On Linux and macOS:

```
python3 -m venv .venv
source .venv/bin/activate
pip install -e ./lib_audio_dsp/python
```

6. Connect an XCORE-AI-EXPLORER using both USB ports

7. The examples are presented as a Jupyter notebook for interactive development. Install Juptyer notebooks into the Python virtual environment with the command:

```
pip install notebook==7.2.1
```

8. Open the notebook by running from *lib_audio_dsp_sandbox* the following command:

```
jupyter notebook lib_audio_dsp/examples/app_simple_audio_dsp_integration/dsp_design.ipynb
```

   If a blank screen appears or nothing opens, then copy the link starting with http://127.0.0.1/ from the terminal into the browser. The top level Jupyter notebook page should open, as can be seein in Fig. 1.

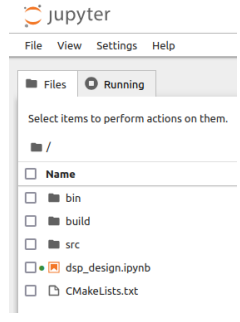

Fig. 1: Top-level page of the Jupyter Notebook

9. Run all the cells from the browser. From the menu at the top of the page click *Run -> Run all cells* (Fig. 2). This creates the pipeline and builds the app. Wait for all the cells to finish.

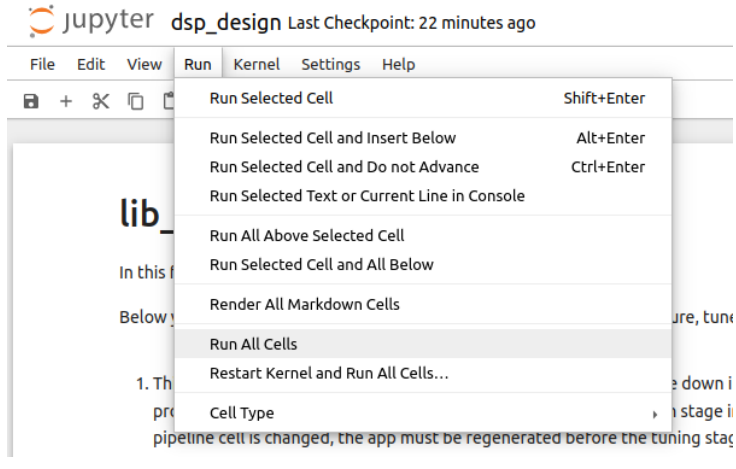

Fig. 2: Run menu of the Jupyter Notebook

Once finished, the setup phase is complete. The notebook should look like as in the example on Fig. 3.

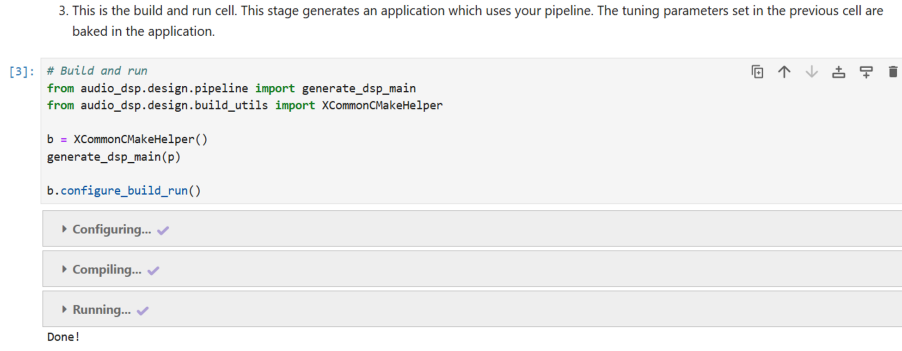

Fig. 3: Run Success of the Jupyter Notebook

If there are any configuration or compilation errors, they will be displayed in the notebook in the *Build and run* cell, as in the example on Fig. 4.

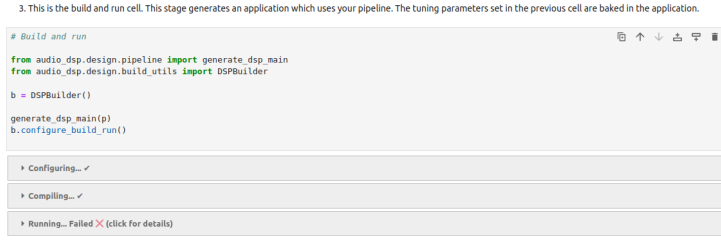

Fig. 4: Run Error of the Jupyter Notebook

Once the setup phase is complete, the user can proceed to either creating a custom pipeline, mapping the pipeline to audio input or output sources, or deploying the pipeline to the xcore. The *Using the Tool* section describes how to achieve this.

Running a notebook after the first installation

If running the notebook after the initial configuration, the following steps are required:

1. Configure the settings below, using the instructions in the *Setup Steps* section:
   - Enable the XTC tools: the installation can be tested by running the command `xrun --version` from the terminal. If the command is not found, the XTC tools are not installed correctly.
   - From your sandbox, enable the Python Virtual Environment and check the path is set:

   On Windows:

```
call .venv/Scripts/activate.bat
echo %VIRTUAL_ENV%
```

   On Linux and macOS:

```
source .venv/bin/activate
echo $VIRTUAL_ENV
```

2. From the `lib_audio_dsp_sandbox` folder, open the notebook by running:

```
jupyter notebook lib_audio_dsp/examples/app_simple_audio_dsp_integration/dsp_design.ipynb
```

1.2 Using the Tool

In this section the basic operation of the tools provided by lib_audio_dsp is described.

This document takes the user through three scenarios, illustrated by way of the included example *app_simple_audio_dsp_integration*, which may be found in the *examples* directory in *lib_audio_dsp*.

These scenarios are:

- Creating a pipeline
- Tuning and simulating a pipeline
- Deploying pipeline code onto the xcore.

The steps in this guide should be executed in a Jupyter Notebook.

Creating a Pipeline

A simple yet useful DSP pipeline that could be made is a bass and treble control with output limiter. In this design the product will stream real time audio boosting or suppressing the treble and bass and then limiting the output amplitude to protect the output device.

The DSP pipeline will perform the processes shown in Fig. 5.

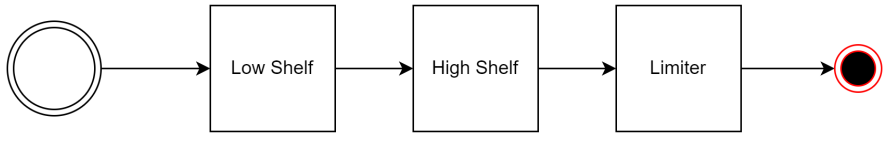

Fig. 5: The target pipeline

The first step is to create an instance of the `Pipeline` class. This is the top level class which will be used to create and tune the pipeline. On creation the number of inputs and sample rate must be specified.

```
from audio_dsp.design.pipeline import Pipeline

pipeline, inputs = Pipeline.begin(
    1,          # Number of pipeline inputs.
    fs=48000    # Sample rate.
)
```

The `Pipeline` object can now be used to add DSP stages. For high shelf and low shelf use *Biquad* and for the limiter use *LimiterPeak*. For a full list of available DSP stages, see the *DSP Stages List*.

```
from audio_dsp.design.pipeline import Pipeline
from audio_dsp.stages import *

p, inputs = Pipeline.begin(1, fs=48000)
```

(continues on next page)

(continued from previous page)

```
# i is a list of pipeline inputs. "lowshelf" is a label for this instance of Biquad.
# The new variable x is the output of the lowshelf Biquad
x = p.stage(Biquad, inputs, "lowshelf")

# The output of lowshelf "x" is passed as the input to the
# highshelf. The variable x is reassigned to the outputs of the new Biquad.
x = p.stage(Biquad, x, "highshelf")

# Connect highshelf to the limiter. Labels are optional, however they are required
# if the stage will be tuned later.
x = p.stage(LimiterPeak, x)

# Finally connect to the output of the pipeline.
p.set_outputs(x)

p.draw()
```

Fig. 6 demonstrates the output of the Jupyter Notebook when the above snippet was executed. The Jupyter Notebook will illustrate the designed pipeline. For information on creating more complex pipeline topologies, see *Designing Complex Pipelines*.

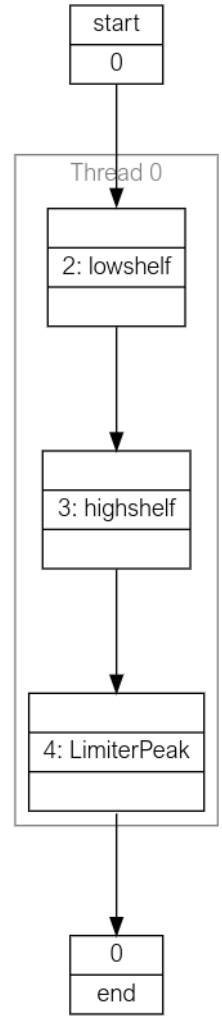

Fig. 6: Generated pipeline diagram

Tuning and simulating a pipeline

Each stage contains a number of designer methods which can be identified as they have the `make_` prefix. These can be used to configure the stages. The stages also provide a `plot_frequency_response()` method which shows the magnitude and phase response of the stage with its current configuration Fig. 7. The two biquads created above will have a flat frequency response until they are tuned. The code below shows how to use the designer methods to convert them into the low shelf and high shelf that is desired. The individual stages are accessed using the labels that were assigned to them when the stage was added to the pipeline.

```
# Make a low shelf with a centre frequency of 200 Hz, q of 0.7 and gain of +6 dB
p["lowshelf"].make_lowshelf(200, 0.7, 6)
p["lowshelf"].plot_frequency_response()

# Make a high shelf with a centre frequency of 4000 Hz, q of 0.7 and gain of +6 dB
p["highshelf"].make_highshelf(4000, 0.7, 6)
p["highshelf"].plot_frequency_response()
```

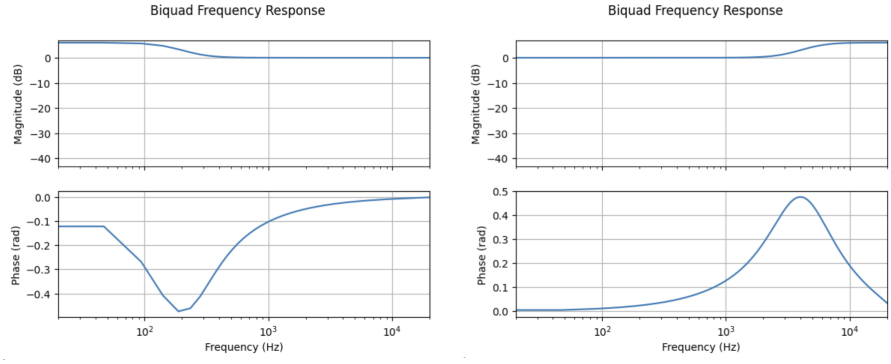

Fig. 7: Frequency response of the biquads (low shelf left, high shelf right)

For this tutorial the default settings for the limiter will provide adequate performance.

Code Generation

With an initial pipeline complete, it is time to generate the xcore source code and run it on a device. The code can be generated using the `generate_dsp_main()` function.

```
from audio_dsp.design.pipeline import generate_dsp_main
generate_dsp_main(p)
```

The reference application should then provide instructions for compiling the application and running it on the target device.

Note: Application Note AN02014 discusses integrating a DSP pipeline into the XMOS USB Reference Design.

The `generate_dsp_main()` function will cause the tuned DSP pipeline to run on the xcore device, where it can be used to stream audio. The next step is to iterate on the design and tune it to perfection. One option is to repeat the steps described above, regenerating the code with new tuning values until the performance requirements are satisfied.

Designing Complex Pipelines

The audio dsp library is not limited to the simple linear pipelines shown above. Stages can scale to take an arbitrary number of inputs, and the outputs of each stage can be split and joined arbitrarily.

When creating a new DSP pipeline, the initialiser returns the pipeline input channels as an instance of `StageOutputList`, a list-like container of `StageOutput`. When adding a new stage to the pipeline, a `StageOutputList` is used to pass the stage inputs. The stage initialiser returns a new instance of `StageOutputList` containing its outputs.

To select specific channels from a `StageOutputList` to pass to another stage, standard Python indexing can be used. Channels from multiple instances of `StageOutputList` can be combined by using the `+` operator.

The below shows an example of how this could work with a pipeline with 7 inputs.

```
 # start with 7 input channels
 p, inputs = Pipeline.begin(7, fs=48000)

# pass the first 2 inputs to a 2-channel Biquad
i0 = p.stage(Biquad, i[0:2])

# pass the third input (index 2) to a 1-channel biquad
i1 = p.stage(Biquad, i[2])

# pass the inputs at index 3, 5, and 6 to a 3 channel biquad
i2 = p.stage(Biquad, i[3, 5, 6])

# pass all of i0 and i1, as well as the first channel in i2
# to create a 4 channel biquad
i3 = p.stage(Biquad, i0 + i1 + i2[0])

# The pipeline output has 6 channels:
# - all four i3 channels
# - the 2nd and 3rd channel from i2
p.set_outputs(i3 + i2[1:])
```

In order to split a signal path, a *Fork* stage should be used. This takes a count parameter that specifies how many times to duplicate each input to the `Fork`. The code block below shows how the signal chain can be forked:

```
p, inputs = Pipeline.begin(1, fs=48000)

# fork the input to create a 2 channel signal
x = p.stage(Fork, inputs, count=2)

# fork again to create a 4 channel signal
x = p.stage(Fork, x, count=2)

# there are now 4 channels in the pipeline output
p.set_outputs(x)
```

As the pipeline grows it may end up consuming more MIPS than are available on a single xcore thread. The pipeline design interface allows adding additional threads using the `next_thread()` method of the `Pipeline` instance. Each thread in the pipeline represents an xcore hardware thread. Do not add more threads than are available in your application. The maximum number of threads that should be used, if available, is five. This limitation is due to the architecture of the xcore processor.

```
# thread 0
i = p.stage(Biquad, i)

# thread 1
p.next_thread()
i = p.stage(Biquad, i)

# thread 2
p.next_thread()
i = p.stage(Biquad, i)
```

2 Design Guide

This section will cover the details of how the xcore DSP pipeline is generated from the Python description. This should enable the reader to debug their pipeline when issues arise and understand the resource usage of a generated DSP pipeline. This is an advanced guide intended for users who wish for a deeper understanding of the generated DSP pipeline that they have created. The accompanying tool and component guides should be consulted for the basic process of using this tool.

2.1 Summary of the xcore architecture

A basic understanding of the xcore architecture is required in order to understand the consequences of various design choices that can be made in the DSP pipeline.

An xcore application will consist of 1 or more xcore chips connected together via a communication fabric (the XLink). Each xcore chip contains 2 or more tiles; a tile is an independent processor with its own memory. A tile cannot read or write the memory of another tile. Each tile contains 8 logical cores; a logical core is an independent thread of execution that will run some application code. Each tile also has 32 chanends available for allocation; connecting 2 chanends forms a channel, which allows for synchronous communication between any 2 logical cores in the system (even between tiles or packages).

In its default configuration, an xcore.ai chip will operate at 600MHz; this means that each tile executes instructions at a rate of 600MIPS. This is shared between the 8 logical cores by multiplexing the execution across 5 time slots. Each thread can consume at most 1 time slot per scheduler cycle. The consequence of this is that for applications with up to 5 threads, each thread operates at 120MIPS (600/5). If there are over 5 threads then this number can be reduced down to 75MIPS (600/8). If any of the threads modify their priority mode then this can reduce the available MIPS even further; high-priority threads are always guaranteed a slot in the scheduler on each cycle.

Term	Definition
xcore.ai	A chip containing 2 or more tiles.
Tile	A single processor with some memory.
Logical Core	1 of the 8 threads available in each tile.
Chanend	The physical hardware used by a logical core to create a channel. There are 32 available per tile.
Channel	The bidirectional communication pathway that is created when 2 chanends are connected.

For more information about the xcore architecture, consult The XMOS XS3 Architecture and the data sheet for your package.

2.2 The Architecture of the Generated Pipeline

Fig. 8 shows the relationship between the classes in an application with a generated DSP pipeline. A class in this context refers to a C struct and the functions that operate on it.

The application package contains *Audio Source*, *Audio Sink* and *Control* classes. The *Audio Source* and *Audio Sink* are responsible for producing and consuming audio at the rate required by the DSP pipeline. The *Control* is responsible for implementing any application specific dynamic control of the DSP pipeline; this is optional and will only be present

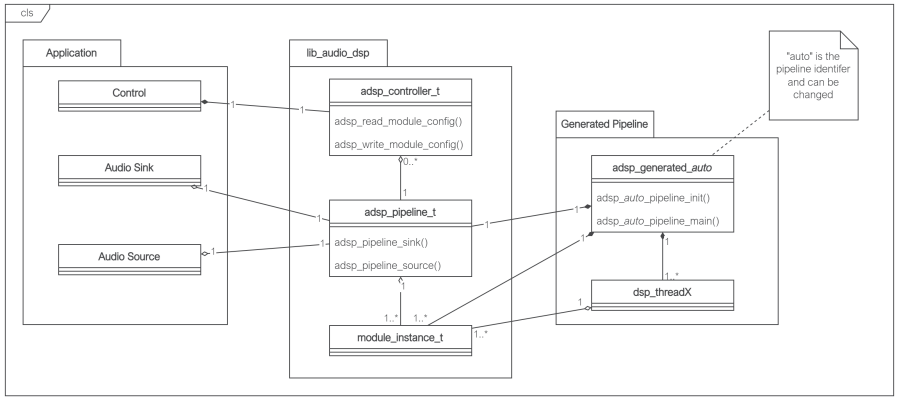

Fig. 8: Class diagram of a lib_audio_dsp application

where run time control is used. These are in the *Application* block as they will be unique for each application. *Audio Source*, *Audio Sink*, and *Control* make use of the classes in lib_audio_dsp; all make use of a pointer to a shared **`adsp_pipeline_t`** (as shown by the aggregation relationships (hollow diamond) in Fig. 8). lib_audio_dsp presents a thread safe API, allowing *Audio Source*, *Audio Sink* and *Control* to exist on separate threads if desired. However, they must all exist on the same tile in order to access the shared **`adsp_pipeline_t`**.

The lib_audio_dsp repository represents the classes from this library. These APIs are documented fully in the *Integration and Control* section.

The *Generated Pipeline* package represents the classes and objects which will be generated from the user's specified DSP pipeline design. Fig. 8 shows that **`adsp_generated_auto`** is composed of (filled diamond) the **`adsp_pipeline_t`** and multiple **`module_instance_t`**. Therefore, the generated pipeline is responsible for allocating the memory for all the stages in the pipeline and also initialising each stage. The generated pipeline also creates multiple threads (labelled **`dsp_threadX`** in Fig. 8), each of which will have been uniquely generated for the DSP pipeline that has been designed. The generated pipeline will always require at least 1 thread to run the DSP on; it is not possible to generate a DSP pipeline that can be executed inline on an existing thread. It is also not possible to split the DSP threads across more than 1 tile, because all threads access a shared **`adsp_pipeline_t`** object.

To summarise, the generated DSP pipeline will consume the number of threads specified in the design (at least 1). At least one other thread on the same tile must be available to exchange audio with the DSP pipeline.

2.3 Resource usage of the Generated Pipeline

The resources that are consumed by the generated DSP pipeline are threads, chanends, and memory. Each DSP thread also has a finite number of instructions per sample that are available for DSP. It is the responsibility of the DSP designer to ensure that this limit is not exceeded on any of the threads.

Chanend Usage

The following snippet of Python shows a DSP design; the pipeline diagram for the snippet is shown in Fig. 9. This design splits 4 DSP stages amongst 3 threads. Threads 0 and 1 operate on the pipeline inputs in parallel. Thread 2 receives its inputs from threads 0 and 1. The pipeline output comes from thread 1.

The generated DSP threads and the APIs for exchanging inputs with the pipeline all use channels to communicate audio.

```
from audio_dsp.design.pipeline import Pipeline
from audio_dsp.stages import *

p, edge = Pipeline.begin(4)

# thread 0
e0 = p.stage(Bypass, edge[0], "a")

# thread 1
p.next_thread()
e1 = p.stage(Bypass, edge[1:], "b")
e1 = p.stage(Bypass, e1, "c")

# thread 2
p.next_thread()
e = p.stage(Bypass, e0 + e1, "d")

p.set_outputs(e)
```

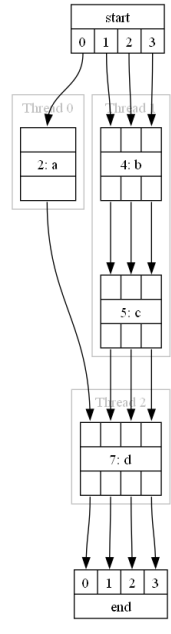

Fig. 9: Output of *Pipeline.draw()* for the example pipeline

Fig. 10 shows how the chanends are allocated for this design. A channel (2 chanends) is allocated for every connection from one thread to another. Thread 2 receives data from thread 0 and 1, therefore it has 2 input channels. It only outputs to 1 thread (end) so has 1 output channel.

If multiple data channels are passed from 1 thread to another (e.g. 3 channels from thread 1 to 2) this still only consumes a single xcore channel (2 chanends) as all the data channels are sent over the same xcore channel.

For a simple linear pipeline, the chanend usage will be $2 * num_dsp_threads + 2$. For pipelines with parallel threads the usage will be higher, as shown in Fig. 10 where 10 chanends (5 channels) are used for 3 DSP threads.

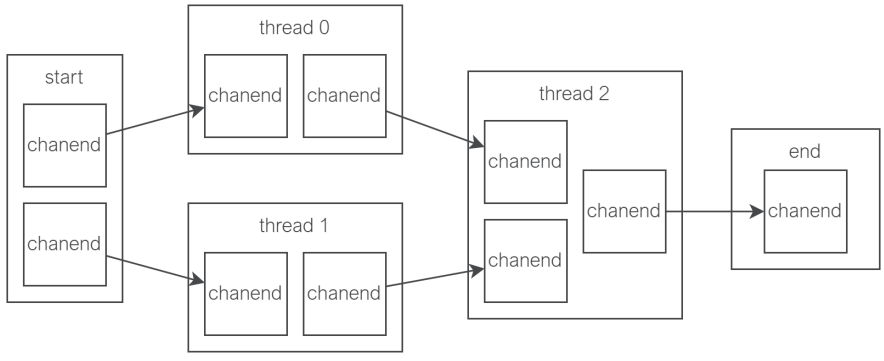

Fig. 10: Chanend usage for the example pipeline

Thread Usage

Thread usage of the DSP pipeline is discussed in the sections above. Understanding the thread usage of your application is a manual process. The application designer must have an understanding of how many threads are in use in their application as well as in the DSP pipeline to ensure that the limit of 8 is not exceeded. If this limit is exceeded the xcore will trap when the application attempts to fork a ninth thread.

Memory Usage

All memory used in the generated DSP pipeline is statically allocated and therefore known at compile time. The Python design API cannot assist in understanding the memory usage of your application. The memory report which is displayed when compiling the application must be consulted to see the memory used. This value will include the generated DSP pipeline as well as any other application code that is running on the tile.

MIPS Usage

In order to operate in a real time audio system it is critical that each thread in the DSP pipeline can complete execution in less time than the sample period (or frame period if the frame size is greater than 1). It is this constraint that requires the DSP to be split into pipelined threads. If a thread is overloaded, the DSP pipeline will consume and produce samples at a slower rate than expected. This could cause the source and sink threads to block and miss timing. The current version of lib_audio_dsp provides only limited support for measuring the MIPS usage of each thread.

Each thread measures the total number of system ticks (periods of the system clock, by default a 100MHz clock) that pass while it is doing work and stores the maximum value that has occured since boot. This measurement can be used to get an estimate of the threads' MIPS utilisations. To access this value, the function `adsp_auto_print_thread_max_ticks()` ("auto" may be replaced with a custom pipeline identifier if specified) is generated along with the other generated pipeline functions. Calling this function on the same tile as the pipeline will print the measured value. Printing is implemented with `printf`, so the output will only be visible when connected to the device with `xrun` or `xgdb`.

The number of available ticks on each thread depends on the frame size and sample rate of the data. For example, given that the system clock runs by default at 100MHz, if the sample rate is 48000 Hz and frame size is 1 then the available ticks will be $1 * 100e6/48000 = 2083 ticks$. Below is an example output from `adsp_auto_print_thread_max_ticks()` for a pipeline with 4 threads:

```
DSP Thread Ticks:
0:     1800
1:     181
2:     67
3:     93
```

The number that is displayed is the worst case that has happened since boot. This is not necessarily the absolute worst case as some stages have data dependent execution time. Therefore, it is recommended to play an audio signal through the pipeline with varying amplitude and frequencies before measuring the thread MIPS.

2.4 Troubleshooting resource issues

Tile exceeds memory limit

Memory available check will report "FAILED" during linking. The *Memory Usage* section describes how memory is allocated in the DSP pipeline. Recommended steps:

1. Remove all stages from the pipeline.
2. Add them back one at a time and take note of the memory usage of each stage.
3. Consult the documentation for the problematic stages and see if its memory usage is configuration dependent.

Moving stages between threads will not impact the memory usage as all threads are on the same tile.

Tile exceeds available chanends

If a tile attempts to allocate too many chanends it will raise an illegal resource exception and cease execution. This can be detected easily with xgdb or xrun as it will print the following message:

```
Unhandled exception: ILLEGAL_RESOURCE
```

The *Chanend Usage* section describes how chanends are used within the DSP pipeline. Resolving this problem will require either redesigning the DSP or the application that runs on the same tile to use fewer chanends.

Exchanging audio with the DSP pipeline blocks for too long

`adsp_pipeline_sink` or `adsp_pipeline_source` will block until data is available. The *MIPS Usage* section describes how to ensure the DSP pipeline meets timing. Identifying this particular issue will depend on the rest of the application. The result could be either dropped samples that are audible in the output or a complete application crash.

Tile exceeds available threads

If a tile attempts to fork too many threads it will raise an illegal resource exception and cease execution. This can be detected easily with xgdb or xrun as it will print the following message:

```
Unhandled exception: ILLEGAL_RESOURCE
```

The *Thread Usage* section describes how threads are used within the DSP pipeline. Resolving this problem will require either redesigning the DSP or the application that runs on the same tile to use fewer threads.

3 DSP Components

lib_audio_dsp provides many common signal processing functions optimised for xcore. These can be combined together to make complex audio pipelines for many different applications, such as home audio, music production, voice processing, and AI feature extraction.

The library is split into 2 levels of API: DSP stages and DSP modules. Both APIs provide similar DSP functionality, but are suited to different use cases.

The higher-level APIs are called *DSP Stages*. These stages are designed to work with the Python DSP pipeline tool. This tool allows developers to quickly and easily create, test, and deploy DSP pipelines without needing to write a lot of code. By using DSP stages, the user can build complex audio processing workflows in a short amount of time, making it ideal for rapid prototyping and development.

The lower-level APIs are called *DSP Modules*. They are meant to be used as an API directly in cases where the Python DSP pipeline tool is not used. These modules can be useful when integrating DSP functionality into an existing system, or as a starting point for creating bespoke DSP functions.

3.1 DSP Stages List

This a list of all the supported stages that can be used with the DSP pipeline tool:

- *Biquad Stages*

  Biquad Stages can be used for basic audio filters.

  - *Biquad*
  - *BiquadSlew*

- *Cascaded Biquads Stages*

  Cascaded biquads Stages consist of several biquad filters connected together in series.

  - *CascadedBiquads*
  - *CascadedBiquads16*

- *Compressor Stages*

  Compressor stages allow for control of the dynamic range of the signal, such as reducing the level of loud sounds.

  - *CompressorRMS*

- *Compressor Sidechain Stages*

  Sidechain compressor Stages use the envelope of one input to control the level of a different input.

  - *CompressorSidechain*
  - *CompressorSidechainStereo*

- *Envelope Detector Stages*

  Envelope detector Stages measure how the average or peak amplitude of a signal varies over time.

  - *EnvelopeDetectorPeak*
  - *EnvelopeDetectorRMS*

- *FIR Stages*

  Finite impulse response (FIR) filter Stages allow the use of arbitrary filters with a finite number of taps.

  - *FirDirect*

- *Graphic EQ Stages*

  Graphic EQs allow frequency response adjustments at fixed center frequencies.

  - *GraphicEq10b*

- *Limiter Stages*

  Limiter Stages allow the amplitude of the signal to be restricted based on its envelope.

  - *LimiterRMS*
  - *LimiterPeak*
  - *HardLimiterPeak*
  - *Clipper*

- *Noise Gate Stages*

  Noise gate Stages remove quiet signals from the audio output.

  - *NoiseGate*

- *Noise Suppressor Expander Stages*

  Noise suppressor and expander Stages control the behaviour of quiet signals, typically by tring to reduce the audibility of noise in the signal.

  - *NoiseSuppressorExpander*

- *Reverb Stages*

  Reverb Stages emulate the natural reverberance of rooms.

  - *ReverbRoom*
  - *ReverbRoomStereo*
  - *ReverbPlateStereo*

- *Signal Chain Stages*

  Signal chain stages allow for the control of signal flow through the pipeline. This includes stages for combining and splitting signals, basic gain components, and delays.

  - *Bypass*
  - *Fork*
  - *Mixer*
  - *Adder*
  - *Subtractor*
  - *FixedGain*
  - *VolumeControl*
  - *Switch*
  - *SwitchSlew*
  - *SwitchStereo*
  - *Delay*
  - *Crossfader*

- *CrossfaderStereo*

3.2 DSP Modules List

This a list of all the modules that can be used independently without the DSP pipeline tool:

- *Biquad Filters*
  - *Single Biquad*
  - *Single Slewing Biquad*
  - *Cascaded Biquads*
  - *Cascaded Biquads 16*
- *Dynamic Range Control*
  - *Peak Envelope Detector*
  - *RMS Envelope Detector*
  - *Clipper*
  - *Peak Limiter*
  - *Hard Peak Limiter*
  - *RMS Limiter*
  - *RMS Compressor*
  - *Sidechain RMS Compressor*
  - *Stereo Sidechain RMS Compressor*
  - *Noise Gate*
  - *Noise Suppressor/Expander*
- *Graphic Equalisers*
  - *10 Band Graphic Equaliser*
- *Finite Impulse Response Filters*
  - *FIR Direct*
  - *Block Time Domain FIR*
  - *Block Frequency Domain FIR*
- *Reverb*
  - *Reverb Room*
  - *Reverb Room Stereo*
  - *Reverb Plate Stereo*
- *Signal Chain Components*
  - *Adder*
  - *Subtractor*
  - *Fixed Gain*
  - *Mixer*
  - *Volume Control*
  - *Delay*
  - *Switch with slew*

- *Crossfader*

3.3 Library Q Format

Note: For fixed point Q formats this document uses the format QM.N, where M is the number of bits before the decimal point (excluding the sign bit), and N is the number of bits after the decimal point. For an int32 number, M+N=31.

By default, the signal processing in the audio pipeline is carried out at 32 bit fixed point precision in Q4.27 format. Assuming a 24 bit input signal in Q0.24 format, this gives 4 bits of internal headroom in the audio pipeline.

Most modules in this library assume that the signal is in a specific global Q format. This format is defined by the `Q_SIG` macro. An additional macro for the signal exponent, `SIG_EXP` is defined, where `SIG_EXP = - Q_SIG`.

`Q_SIG`

Default Q format

`SIG_EXP`

Default signal exponent

To ensure optimal headroom and noise floor, the user should ensure that signals are in the correct Q format before processing. Either the input Q format can be converted to `Q_SIG`, or `Q_SIG` can be changed to the desired value.

Note: Not using the DSP pipeline tool means that Q formats will not automatically be managed, and the user should take care to ensure they have the correct values for optimum performance and signal level.

For example, for more precision, the pipeline can be configured to run with no headroom in Q0.31 format, but this would require manual headroom management (e.g. reducing the signal level before a boost to avoid clipping).

To convert between `Q_SIG` and Q0.31 in a safe and optimised way, the APIs below are provided.

int32_t **`adsp_from_q31`**(int32_t input)

Convert from Q0.31 to Q_SIG.

Parameters

- **`input`** – Input in Q0.31 format

Returns

int32_t Output in Q_SIG format

int32_t **`adsp_to_q31`**(int32_t input)

Convert from Q_SIG to Q0.31.

Parameters

- **`input`** – Input in Q_SIG format

Returns
int32_t Output in Q0.31 format

3.4 Precision

Note: For fixed point Q formats this document uses the format QM.N, where M is the number of bits before the decimal point (excluding the sign bit), and N is the number of bits after the decimal point. For an int32 number, M+N=31.

By default, the signal processing in the audio pipeline is carried out at 32 bit fixed point precision in Q4.27 format. Assuming a 24 bit input signal in Q0.24 format, this gives 4 bits of internal headroom in the audio pipeline, which is equivalent to 24 dB. The output of the audio pipeline will be clipped back to Q0.24 before returning. For more precision, the pipeline can be configured to run with no headroom in Q0.31 format, but this requires manual headroom management. More information on setting the Q format can be found in the *Library Q Format* section.

DSP algorithms are implemented either on the XS3 CPU or VPU (vector processing unit).

CPU algorithms are typically implemented as 32-bit x 32-bit operations into 64-bit results and accumulators, before rounding back to 32-bit outputs.

The VPU allows for 8 simultaneous operations, with a small cost in precision. VPU algorithms are typically implemented as 32-bit x 32-bit operations into 34-bit results and 40-bit accumulators, before rounding back to 32-bit outputs.

3.5 Latency

The latency of the DSP pipeline is dependent on the number of threads. By default, the DSP pipeline is configured for one sample of latency per thread. All current DSP modules have zero inbuilt latency (except where specified e.g. delay stages). For pipelines that fit on a single thread, this means the total pipeline latency is 1 sample.

The pipeline can also be configured to use a higher frame size. This increases latency, but can reduce compute for simple functions. For a pipeline consisting of just biquads:

- Frame size = 1, latency = 1 sample, compute = 25 biquads per thread @ 48kHz.
- Frame size = 8, latency = 8 samples, compute = 60 biquads per thread @ 48kHz.

4 Run-Time Control User Guide

For many applications, the ability to update the DSP configuration at run time will be required. A simple example would be a volume control where the end product will update the volume setting based on user input. This DSP library has been designed with use cases like this in mind and the generated DSP pipeline provides an interface for writing and reading the configuration of each stage.

This document details how to use this interface to extend a DSP application with run-time control of the audio processing. For a complete example of an application that updates the DSP configuration based on user input refer to application note AN02015.

4.1 Defining a Controllable Pipeline

This section will walk through adding control to a basic DSP pipeline. The following code snippet describes a simple DSP process with a volume control and a limiter. In the end application the volume can be set by the application. This code snippet will generate the pipeline diagram shown in Fig. 11.

```
from audio_dsp.design.pipeline import Pipeline
from audio_dsp.stages import *

p, edge = Pipeline.begin(4)
edge = p.stage(VolumeControl, edge, "volume")
edge = p.stage(LimiterRMS, edge)
p.set_outputs(edge)
```

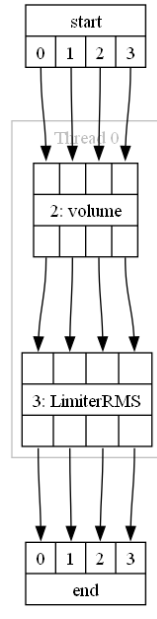

Fig. 11: The example pipeline diagram

In this example the tuning methods on the stages in the pipeline are not called which means the code that is generated will intialise the stages with their default configuration values.

A point of interest in this example is that the `label` argument to the pipeline `stage` method is set, but only for the volume control stage. The label for the volume control in this example is `volume`. After generating the source code for this pipeline, a file will

be created in the specified directory named `adsp_instance_id_auto.h` (assuming that the pipeline identifier has been left as its default value of "auto"). The contents of the generated file are shown below:

```
// Copyright 2024-2025 XMOS LIMITED.
// This Software is subject to the terms of the XMOS Public Licence: Version 1.
#pragma once

#define thread0_stage_index               (1)
#define volume_stage_index               (2)
#define auto_thread_stage_indices  { thread0_stage_index }
```

In this file the macro `volume_stage_index` is defined. The value of this macro can be used by the control interface to find the volume control stage and process control commands. The benefit of this to an application author is that this header file can be included in the application and the value of `volume_stage_index` will always be correct, even when the pipeline is redesigned.

4.2 Writing the Configuration of a Stage

Each stage type has a set of controllable parameters that can be read or written. A description of each parameter along with its type and name can be found in the *DSP Stages* section in the DSP components document. For volume control, there is a command named `CMD_VOLUME_CONTROL_TARGET_GAIN` that can be updated at run time to set the volume. This command is defined in the generated header file `cmds.h` which will be placed into the build directory at `src.autogen/common/cmds.h`. `cmds.h` contains all the command IDs for all the stage types that CMake found.

It is also possible to see the available control parameters, along with the values they will be set to, while designing the pipeline in Python. This can be done using the `get_config` method of the stage as shown below.

```
config = p["volume"].get_config()
print(config)
```

This will print this dictionary of parameters:

```
{'target_gain': 134217728, 'slew_shift': 7, 'mute_state': 0}
```

This dictionary does not contain `CMD_VOLUME_CONTROL_TARGET_GAIN`, but is does contain "target_gain". The final command name is constructed as `CMD_{STAGE_TYPE}_{PARAMETER}` where stage type and parameter should be replaced with the correct values for each, capitalised. All stages of the same type (e.g. `VolumeControl`) will have the same set of parameters.

The format and type of the control parameters for each stage are chosen to optimise processing time on the DSP thread. For example, `CMD_VOLUME_CONTROL_TARGET_GAIN` is not a floating point value in decibels, but rather a linear fixed point value. For this example we can use the convenience function `adsp_dB_to_gain()` which is defined in `control/signal_chain.h`.

In order to send a control command, the API defined in `stages/adsp_control.h` is used. This API is documented in the *API Reference*, in the *Control* section. Complete the following steps:

1. Create a thread that will be updating the DSP configuration. This thread must be on the same tile as the DSP.

2. Create a new `adsp_controller_t` from the `adsp_pipeline_t` that was initialised for the generated pipeline. If multiple threads will be attempting control, each thread must have a unique instance of `adsp_controller_t` to ensure thread safety.

3. Initialise a new `adsp_stage_control_cmd_t`, specifying the instance ID (`volume_stage_index`), the command ID (`CMD_VOLUME_CONTROL_TARGET_GAIN`), and payload length (`sizeof(int32_t)`).

4. Create the command payload; this will be an `int32_t` containing the computed gain. Update the command payload pointer to reference the payload.

5. Call `adsp_write_module_config` until it returns `ADSP_CONTROL_SUCCESS`. There may be in-progress write or read commands which have been issued but not completed when starting the new command. In this scenario the `adsp_write_module_config` will return `ADSP_CONTROL_BUSY` which means that the attempt to write had no effect and should be attempted again.

A full example of a control thread that does this is shown below.

```
#include <xcore/parallel.h>
#include "cmds.h"
#include "adsp_generated_auto.h"
#include "adsp_instance_id_auto.h"
#include "dsp/signal_chain.h"
#include "control/signal_chain.h"
#include "stages/adsp_control.h"
#include "stages/adsp_pipeline.h"

void control_thread(adsp_controller_t* control) {
  // convert desired value to parameter type
  float desired_vol_db = -6;
  int32_t desired_vol_raw = adsp_dB_to_gain(desired_vol_db);

  adsp_stage_control_cmd_t command = {
    .instance_id = volume_stage_index,
    .cmd_id = CMD_VOLUME_CONTROL_TARGET_GAIN,
    .payload_len = sizeof(desired_vol_raw),
    .payload = &desired_vol_raw
  };

  // try write until success
  while(ADSP_CONTROL_SUCCESS != adsp_write_module_config(control, &command));

  // DONE!
}

void audio_source_sink(adsp_pipeline_t* p) {
  // sends and receives audio to the pipeline
}

void dsp_main(void) {
  adsp_pipeline_t* dsp = adsp_auto_pipeline_init();

  // created a controller instance for each thread.
  adsp_controller_t control;
  adsp_controller_init(&control, dsp);

  PAR_FUNCS(
    PFUNC(audio_source_sink, dsp),
    PFUNC(control_thread, &control),
    PFUNC(adsp_auto_pipeline_main, dsp)
  );
}
```

4.3 Reading the Configuration of a Stage

In some cases it makes sense to read back the configuration of the stage. Some stages have dynamic values that are updated as the audio is processed and can be read back to the control thread. Volume control is an example of this as it will smoothly adjust the gain towards `CMD_VOLUME_CONTROL_TARGET_GAIN`; the current value of the gain which is actually being applied can be read by reading from the parameter `CMD_VOLUME_CONTROL_GAIN`. The API for reading is largely the same as writing, except the control API will write to the payload buffer.

This code example shows how to read the current `CMD_VOLUME_CONTROL_GAIN` parameter from the “volume” stage that is created in the example above.

```
int32_t read_volume_gain(adsp_controller_t* control) {
  int32_t gain_raw;

  adsp_stage_control_cmd_t command = {
    .instance_id = volume_stage_index,
    .cmd_id = CMD_VOLUME_CONTROL_GAIN,
    .payload_len = sizeof(gain_raw),
    .payload = &gain_raw
  };

  // try write until success
  while(ADSP_CONTROL_SUCCESS != adsp_read_module_config(control, &command));

  return gain_raw;
}
```

4.4 Control Interface Details

This section provides a brief overview of how the control interface works.

Each stage that is included in the generated DSP pipeline has its own state which it will maintain as it processes audio. It also has a structure that contains its configuration parameters. Finally, it has a control state variable which is used to communicate between the DSP and control threads. Threads that wish to read or write to the configuration of a stage use the control API that is discussed above.

For a write command, the controlling thread will check that a command is not ongoing by querying the control state of the stage. If the stage is not processing a control command then the control thread will update the configuration struct for the stage and write to the control state variable that new parameters are available. When the DSP thread next gets an opportunity the stage will see that the parameters have been updated and update its internal state to match. When this is complete the control state variable will be cleared.

For a read command the process is similar. The control thread requests a read by updating the control state variable. The stage will see this and update the configuration struct with the latest value and notify the control thread, via the control state variable, that it has completed the request.

The control API ensures thread safety through the use of the `adsp_controller_t` struct. As long as each thread uses a unique instance of `adsp_controller_t` then the control APIs will return `ADSP_CONTROL_BUSY` if a command that was initialised by another `adsp_controller_t` is ongoing.

5 API Reference

This section provides a comprehensive guide to the DSP components, their integration into the DSP pipeline, and the run-time control mechanisms. It also includes an overview of high-level pipeline design principles.

5.1 DSP Stages

DSP stages are high level blocks for use in the Python DSP pipeline tool. Each Stage has a Python and C implementation, allowing pipelines to be rapidly prototyped in Python before being easily deployed to hardware in C. The audio performance of both implementations is equivalent.

Most stages have parameters that can be changed at runtime, and the available parameters are outlined in the documentation.

All the DSP stages can be imported into a Python file using:

```
from audio_dsp.stages import *
```

The following DSP stages are available for use in the Python DSP pipeline design.

Biquad Stages

Biquad Stages can be used for basic audio filters.

Biquad

class `audio_dsp.stages.Biquad`(***kwargs*)

A second order biquadratic filter, which can be used to make many common second order filters. The filter is initialised in a bypass state, and the `make_*` methods can be used to calculate the coefficients.

This Stage implements a direct form 1 biquad filter: `a0*y[n] = b0*x[n] + b1*x[n-1] + b2*x[n-2] - a1*y[n-1] - a2*y[n-2]`

For efficiency the biquad coefficients are normalised by `a0` and the output `a` coefficients multiplied by -1.

Attributes

dsp_block
[`audio_dsp.dsp.biquad.biquad`] The DSP block class; see *Single Biquad* for implementation details.

`make_allpass`(*f: float*, *q: float*) → Biquad

Make this biquad an all pass filter.

Parameters

f
[float] Center frequency of the filter in Hz.

q
[float] Q factor of the filter.

`make_bandpass`(*f: float*, *bw: float*) → Biquad

Make this biquad a second order bandpass filter.

Parameters

f
[float] Center frequency of the filter in Hz.

bw
[float] Bandwidth of the filter in octaves.

make_bandstop(*f: float*, *bw: float*) → Biquad

Make this biquad a second order bandstop filter.

Parameters

f
[float] Center frequency of the filter in Hz.

bw
[float] Bandwidth of the filter in octaves.

make_bypass() → Biquad

Make this biquad a bypass by setting the b0 coefficient to 1.

make_constant_q(*f: float*, *q: float*, *boost_db: float*) → Biquad

Make this biquad a peaking filter with constant Q.

Constant Q means that the bandwidth of the filter remains constant as the gain varies. It is commonly used for graphic equalisers.

Parameters

f
[float] Center frequency of the filter in Hz.

q
[float] Q factor of the filter.

boost_db
[float] Gain of the filter in decibels.

make_highpass(*f: float*, *q: float*) → Biquad

Make this biquad a second order high pass filter.

Parameters

f
[float] Cutoff frequency of the filter in Hz.

q
[float] Q factor of the filter roll-off. 0.707 is equivalent to a Butterworth response.

make_highshelf(*f: float*, *q: float*, *boost_db: float*) → Biquad

Make this biquad a second order high shelf filter.

The Q factor is defined in a similar way to standard high pass, i.e. > 0.707 will yield peakiness (where the shelf response does not monotonically change). The level change at f will be boost_db/2.

Parameters

f
[float] Cutoff frequency of the shelf in Hz, where the gain is boost_db/2

q
[float] Q factor of the filter.

boost_db
[float] Gain of the filter in decibels.

make_linkwitz(*f0: float*, *q0: float*, *fp: float*, *qp: float*) → Biquad

Make this biquad a Linkwitz Transform biquad filter.

The Linkwitz Transform changes the low frequency cutoff of a filter, and is commonly used to change the low frequency roll off slope of a loudspeaker. When applied to a loudspeaker, it will change the cutoff frequency from f0 to fp, and the Q factor from q0 to qp.

Parameters

f0
: [float] The original cutoff frequency of the filter in Hz.

q0
: [float] The original quality factor of the filter at f0.

fp
: [float] The target cutoff frequency for the filter in Hz.

qp
: [float] The target quality factor for the filter.

make_lowpass(*f: float*, *q: float*) → Biquad

Make this biquad a second order low pass filter.

Parameters

f
: [float] Cutoff frequency of the filter in Hz.

q
: [float] Q factor of the filter roll-off. 0.707 is equivalent to a Butterworth response.

make_lowshelf(*f: float*, *q: float*, *boost_db: float*) → Biquad

Make this biquad a second order low shelf filter.

The Q factor is defined in a similar way to standard low pass, i.e. > 0.707 will yield peakiness (where the shelf response does not monotonically change). The level change at f will be boost_db/2.

Parameters

f
: [float] Cutoff frequency of the shelf in Hz, where the gain is boost_db/2

q
: [float] Q factor of the filter.

boost_db
: [float] Gain of the filter in decibels.

make_notch(*f: float*, *q: float*) → Biquad

Make this biquad a notch filter.

Parameters

f
: [float] Center frequency of the filter in Hz.

q
: [float] Q factor of the filter.

make_peaking(*f: float*, *q: float*, *boost_db: float*) → Biquad

Make this biquad a peaking filter.

Parameters

f
[float] Center frequency of the filter in Hz.

q
[float] Q factor of the filter.

boost_db
[float] Gain of the filter in decibels.

Biquad Control The following runtime command ids are available for the Biquad Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_BIQUAD_LEFT_SHIFT The number of bits to shift the output left by, in order to compensate for any right shift applied to the biquad b coefficients.	`sizeof(int)`
CMD_BIQUAD_FILTER_COEFFS The normalised biquad filter coefficients, in the order *[b0, b1, b2, -a1, -a2]/a0*. The coefficients should be in Q1.30 format. If the maximum b coefficient magnitude is greater than 2.0, the b coefficients should be right shifted to fit in Q1.30 format, and the shift value passed as left_shift to correct the gain after filtering. Biquad coefficients can be generated using the helper functions in `control/biquad.h`. See *Biquad helpers*.	`sizeof(int32_t)*[5]`
CMD_BIQUAD_RESERVED Reserved memory to ensure the VPU receives 8 DWORD_ALIGNED coefficients. This command is read only. When sending a write control command, it will be ignored.	`sizeof(int32_t)*[3]`

BiquadSlew

class `audio_dsp.stages.BiquadSlew`(***kwargs*)

A second order biquadratic filter with slew, which can be used to make many common second order filters. The filter is initialised in a bypass state, and the `make_*` methods can be used to calculate the coefficients. This variant will slew between filter coefficients when they are changed.

This Stage implements a direct form 1 biquad filter: `a0*y[n] = b0*x[n] + b1*x[n-1] + b2*x[n-2] - a1*y[n-1] - a2*y[n-2]`

For efficiency the biquad coefficients are normalised by `a0` and the output `a` coefficients multiplied by -1.

Attributes

dsp_block
[`audio_dsp.dsp.biquad.biquad_slew`] The DSP block class; see *Single Slewing Biquad* for implementation details.

`make_allpass`(*f: float*, *q: float*) → Biquad

Make this biquad an all pass filter.

Parameters

f
[float] Center frequency of the filter in Hz.

q
[float] Q factor of the filter.

make_bandpass(*f: float*, *bw: float*) → Biquad

Make this biquad a second order bandpass filter.

Parameters

f
[float] Center frequency of the filter in Hz.

bw
[float] Bandwidth of the filter in octaves.

make_bandstop(*f: float*, *bw: float*) → Biquad

Make this biquad a second order bandstop filter.

Parameters

f
[float] Center frequency of the filter in Hz.

bw
[float] Bandwidth of the filter in octaves.

make_bypass() → Biquad

Make this biquad a bypass by setting the b0 coefficient to 1.

make_constant_q(*f: float*, *q: float*, *boost_db: float*) → Biquad

Make this biquad a peaking filter with constant Q.

Constant Q means that the bandwidth of the filter remains constant as the gain varies. It is commonly used for graphic equalisers.

Parameters

f
[float] Center frequency of the filter in Hz.

q
[float] Q factor of the filter.

boost_db
[float] Gain of the filter in decibels.

make_highpass(*f: float*, *q: float*) → Biquad

Make this biquad a second order high pass filter.

Parameters

f
[float] Cutoff frequency of the filter in Hz.

q
[float] Q factor of the filter roll-off. 0.707 is equivalent to a Butterworth response.

make_highshelf(*f: float*, *q: float*, *boost_db: float*) → Biquad

Make this biquad a second order high shelf filter.

The Q factor is defined in a similar way to standard high pass, i.e. > 0.707 will yield peakiness (where the shelf response does not monotonically change). The level change at f will be boost_db/2.

Parameters

f
[float] Cutoff frequency of the shelf in Hz, where the gain is boost_db/2

q
[float] Q factor of the filter.

boost_db
[float] Gain of the filter in decibels.

make_linkwitz(*f0: float*, *q0: float*, *fp: float*, *qp: float*) → Biquad

Make this biquad a Linkwitz Transform biquad filter.

The Linkwitz Transform changes the low frequency cutoff of a filter, and is commonly used to change the low frequency roll off slope of a loudspeaker. When applied to a loudspeaker, it will change the cutoff frequency from f0 to fp, and the Q factor from q0 to qp.

Parameters

f0
[float] The original cutoff frequency of the filter in Hz.

q0
[float] The original quality factor of the filter at f0.

fp
[float] The target cutoff frequency for the filter in Hz.

qp
[float] The target quality factor for the filter.

make_lowpass(*f: float*, *q: float*) → Biquad

Make this biquad a second order low pass filter.

Parameters

f
[float] Cutoff frequency of the filter in Hz.

q
[float] Q factor of the filter roll-off. 0.707 is equivalent to a Butterworth response.

make_lowshelf(*f: float*, *q: float*, *boost_db: float*) → Biquad

Make this biquad a second order low shelf filter.

The Q factor is defined in a similar way to standard low pass, i.e. > 0.707 will yield peakiness (where the shelf response does not monotonically change). The level change at f will be boost_db/2.

Parameters

f
[float] Cutoff frequency of the shelf in Hz, where the gain is boost_db/2

q
[float] Q factor of the filter.

boost_db
[float] Gain of the filter in decibels.

make_notch(*f: float*, *q: float*) → Biquad

Make this biquad a notch filter.

Parameters

f
[float] Center frequency of the filter in Hz.

q
[float] Q factor of the filter.

`make_peaking`(*f: float*, *q: float*, *boost_db: float*) → Biquad

Make this biquad a peaking filter.

Parameters

f
[float] Center frequency of the filter in Hz.

q
[float] Q factor of the filter.

boost_db
[float] Gain of the filter in decibels.

`set_slew_shift`(*slew_shift*)

Set the slew shift for a biquad object. This sets how fast the filter will slew between filter coefficients.

BiquadSlew Control The following runtime command ids are available for the BiquadSlew Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_BIQUAD_SLEW_LEFT_SHIFT The number of bits to shift the output left by, in order to compensate for any right shift applied to the biquad b coefficients.	`sizeof(int)`
CMD_BIQUAD_SLEW_FILTER_COEFFS The normalised biquad filter coefficients, in the order *[b0, b1, b2, -a1, -a2]/a0*. The coefficients should be in Q1.30 format. If the maximum b coefficient magnitude is greater than 2.0, the b coefficients should be right shifted to fit in Q1.30 format, and the shift value passed as left_shift to correct the gain after filtering. Biquad coefficients can be generated using the helper functions in `control/biquad.h`. See *Biquad helpers*.	`sizeof(int32_t)*[5]`
CMD_BIQUAD_SLEW_RESERVED Reserved memory to ensure the VPU receives 8 DWORD_ALIGNED coefficients. This command is read only. When sending a write control command, it will be ignored.	`sizeof(int32_t)*[3]`
CMD_BIQUAD_SLEW_SLEW_SHIFT The shift value used to set the slew rate. See the biquad slew control documentation for conversions between slew_shift and time constant.	`sizeof(int)`

Cascaded Biquads Stages

Cascaded biquads Stages consist of several biquad filters connected together in series.

CascadedBiquads

class `audio_dsp.stages.`**CascadedBiquads**(***kwargs*)

8 cascaded biquad filters. This allows up to 8 second order biquad filters to be run in series.

This can be used for either:

- an Nth order filter built out of cascaded second order sections
- a parametric EQ, where several biquad filters are used at once.

For documentation on the individual biquad filters, see `audio_dsp.stages.biquad.Biquad` and `audio_dsp.dsp.biquad.biquad`

Attributes

dsp_block
[`audio_dsp.dsp.cascaded_biquad.cascaded_biquad`] The DSP block class; see *Cascaded Biquads* for implementation details.

make_butterworth_highpass(*N: int*, *fc: float*) → CascadedBiquads

Configure this instance as an Nth order Butterworth highpass filter using N/2 cascaded biquads.

For details on the implementation, see `audio_dsp.dsp.cascaded_biquads.make_butterworth_highpass`

Parameters

N
[int] Filter order, must be even

fc
[float] -3 dB frequency in Hz.

make_butterworth_lowpass(*N: int*, *fc: float*) → CascadedBiquads

Configure this instance as an Nth order Butterworth lowpass filter using N/2 cascaded biquads.

For details on the implementation, see `audio_dsp.dsp.cascaded_biquads.make_butterworth_lowpass`

Parameters

N
[int] Filter order, must be even

fc
[float] -3 dB frequency in Hz.

make_parametric_eq(*filter_spec: list[list[Any]]*) → CascadedBiquads

Configure this instance as a Parametric Equaliser.

This allows each of the 8 biquads to be individually designed using the designer methods for the biquad. This expects to receive a list of up to 8 biquad design descriptions where a biquad design description is of the form:

```
["type", args...]
```

where "type" is a string defining how the biquad should be designed e.g. "lowpass", and args... is all the parameters to design that type of filter. All options and arguments are listed below:

```
["allpass", filter_freq, q_factor]
["bandpass", filter_freq, BW]
["bandstop", filter_freq, BW]
["bypass"]
["constant_q", filter_freq, q_factor, boost_db]
["gain", gain_db]
["highpass", filter_freq, q_factor]
["highshelf", filter_freq, q_factor, gain_db]
["linkwitz", f0, q0, fp, qp]
["lowpass", filter_freq, q_factor]
["lowshelf", filter_freq, q_factor, gain_db]
["mute"]
["notch", filter_freq, q_factor]
["peaking", filter_freq, q_factor, boost_db]
```

CascadedBiquads Control The following runtime command ids are available for the CascadedBiquads Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_CASCADED_BIQUADS_LEFT_SHIFT The coefficient shift applied to the output of each biquad in the cascade. The shifts should be in the same format as specified in the individual biquad.	`sizeof(int)*[8]`
CMD_CASCADED_BIQUADS_FILTER_COEFFS The normalised biquad filter coefficients for each biquad in the cascade as an array of [8][5], with 5 coefficients for up to 8 biquads. The coefficients should be in the same format as specified in the individual biquad. See *Biquad helpers*.	`sizeof(int32_t)*[40]`

CascadedBiquads16

class `audio_dsp.stages.`**CascadedBiquads16**(***kwargs*)

16 cascaded biquad filters. This allows up to 16 second order biquad filters to be run in series.

This can be used for either:

- an Nth order filter built out of cascaded second order sections
- a parametric EQ, where several biquad filters are used at once.

For documentation on the individual biquad filters, see `audio_dsp.stages.biquad.Biquad` and `audio_dsp.dsp.biquad.biquad`

Attributes

dsp_block
[`audio_dsp.dsp.cascaded_biquad.cascaded_biquad_16`] The DSP block class; see *Cascaded Biquads 16* for implementation details.

make_parametric_eq(*filter_spec: list[list[Any]]*) → CascadedBiquads16

Configure this instance as a Parametric Equaliser.

This allows each of the 16 biquads to be individually designed using the designer methods for the biquad. This expects to receive a list of up to 8 biquad design descriptions where a biquad design description is of the form:

```
["type", args...]
```

where “type” is a string defining how the biquad should be designed e.g. “low-pass”, and args… is all the parameters to design that type of filter. All options and arguments are listed below:

```
["allpass", filter_freq, q_factor]
["bandpass", filter_freq, BW]
["bandstop", filter_freq, BW]
["bypass"]
["constant_q", filter_freq, q_factor, boost_db]
["gain", gain_db]
["highpass", filter_freq, q_factor]
["highshelf", filter_freq, q_factor, gain_db]
["linkwitz", f0, q0, fp, qp]
["lowpass", filter_freq, q_factor]
["lowshelf", filter_freq, q_factor, gain_db]
["mute"]
["notch", filter_freq, q_factor]
["peaking", filter_freq, q_factor, boost_db]
```

CascadedBiquads16 Control The following runtime command ids are available for the CascadedBiquads16 Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_CASCADED_BIQUADS16_LEFT_SHIFT The coefficient shift applied to the output of each biquad in the cascade. The shifts should be in the same format as specified in the individual biquad.	`sizeof(int)*[16]`
CMD_CASCADED_BIQUADS16_FILTER_COEFFS_LOWER The normalised biquad filter coefficients for the first 8 biquads in the cascade as an array of [8][5], with 5 coefficients for 8 biquads. The coefficients should be in the same format as specified in the individual biquad. See *Biquad helpers*.	`sizeof(int32_t)*[40]`
CMD_CASCADED_BIQUADS16_FILTER_COEFFS_UPPER The normalised biquad filter coefficients for the last 8 biquads in the cascade as an array of [8][5], with 5 coefficients for up to 8 biquads. The coefficients should be in the same format as specified in the individual biquad. See *Biquad helpers*.	`sizeof(int32_t)*[40]`

Compressor Stages

Compressor stages allow for control of the dynamic range of the signal, such as reducing the level of loud sounds.

CompressorRMS

class `audio_dsp.stages.CompressorRMS`(***kwargs*)

A compressor based on the RMS envelope of the input signal.

When the RMS envelope of the signal exceeds the threshold, the signal amplitude is reduced by the compression ratio.

The threshold sets the value above which compression occurs. The ratio sets how much the signal is compressed. A ratio of 1 results in no compression, while a ratio of infinity results in the same behaviour as a limiter. The attack time sets how fast

the compressor starts compressing. The release time sets how long the signal takes to ramp up to its original level after the envelope is below the threshold.

Attributes

dsp_block
[`audio_dsp.dsp.drc.drc.compressor_rms`] The DSB block class; see *RMS Compressor* for implementation details.

`make_compressor_rms`(*ratio*, *threshold_db*, *attack_t*, *release_t*, *Q_sig=27*)

Update compressor configuration based on new parameters.

Parameters

ratio
[float] Compression gain ratio applied when the signal is above the threshold.

threshold_db
[float] Threshold in decibels above which compression occurs.

attack_t
[float] Attack time of the compressor in seconds.

release_t
[float] Release time of the compressor in seconds.

CompressorRMS Control The following runtime command ids are available for the CompressorRMS Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_COMPRESSOR_RMS_ATTACK_ALPHA The attack alpha in Q0.31 format. To convert an attack time in seconds to an `int32_t` control value, use the function *`calc_alpha()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_COMPRESSOR_RMS_RELEASE_ALPHA The release alpha in Q0.31 format. To convert a release time in seconds to an `int32_t` control value, use the function *`calc_alpha()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_COMPRESSOR_RMS_ENVELOPE The current RMS² envelope of the signal in Q_SIG format. To read the `int32_t` control value, use the function *`qxx_to_db_pow()`* in `control/helpers.h` with Q=Q_SIG. This command is read only. When sending a write control command, it will be ignored.	`sizeof(int32_t)`
CMD_COMPRESSOR_RMS_THRESHOLD The threshold in Q_SIG format above which compression will occur. To convert a threshold in dB to the `int32_t` control value, use the function *`calculate_rms_threshold()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_COMPRESSOR_RMS_GAIN The current gain applied by the compressor in Q0.31 format. To read the `int32_t` control value, use the function *`qxx_to_db()`* in `control/helpers.h` with Q=31. This command is read only. When sending a write control command, it will be ignored.	`sizeof(int32_t)`
CMD_COMPRESSOR_RMS_SLOPE The compression slope of the compressor. This is calculated as `(1 - 1 / ratio) / 2.0`. To convert a ratio to a slope, use the function *`rms_compressor_slope_from_ratio()`* in `control/helpers.h`.	`sizeof(float)`

Compressor Sidechain Stages

Sidechain compressor Stages use the envelope of one input to control the level of a different input.

CompressorSidechain

class audio_dsp.stages.**CompressorSidechain**(***kwargs*)

An sidechain compressor based on the RMS envelope of the detect signal.

This stage is limited to accepting 2 channels. The first is the channel that will be compressed. The second is the detect channel. The level of compression depends on the envelope of the second channel.

When the RMS envelope of the detect signal exceeds the threshold, the processed signal amplitude is reduced by the compression ratio.

The threshold sets the value above which compression occurs. The ratio sets how much the signal is compressed. A ratio of 1 results in no compression, while a ratio of infinity results in the same behaviour as a limiter. The attack time sets how fast the compressor starts compressing. The release time sets how long the signal takes to ramp up to its original level after the envelope is below the threshold.

Attributes

dsp_block
[`audio_dsp.dsp.drc.sidechain.compressor_rms_sidechain_mono`] The DSP block class; see *Sidechain RMS Compressor* for implementation details.

`make_compressor_sidechain`(*ratio*, *threshold_db*, *attack_t*, *release_t*, *Q_sig=27*)

Update compressor configuration based on new parameters.

Parameters

ratio
[float] Compression gain ratio applied when the signal is above the threshold.

threshold_db
[float] Threshold in decibels above which compression occurs.

attack_t
[float] Attack time of the compressor in seconds.

release_t
[float] Release time of the compressor in seconds.

CompressorSidechain Control The following runtime command ids are available for the CompressorSidechain Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_COMPRESSOR_SIDECHAIN_ATTACK_ALPHA The attack alpha in Q0.31 format. To convert an attack time in seconds to an `int32_t` control value, use the function *`calc_alpha()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_COMPRESSOR_SIDECHAIN_RELEASE_ALPHA The release alpha in Q0.31 format. To convert a release time in seconds to an `int32_t` control value, use the function *`calc_alpha()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_COMPRESSOR_SIDECHAIN_ENVELOPE The current RMS² envelope of the signal in Q_SIG format. To read the `int32_t` control value, use the function *`qxx_to_db_pow()`* in `control/helpers.h` with Q=Q_SIG. This command is read only. When sending a write control command, it will be ignored.	`sizeof(int32_t)`
CMD_COMPRESSOR_SIDECHAIN_THRESHOLD The threshold in Q_SIG format above which compression will occur. To convert a threshold in dB to the `int32_t` control value, use the function *`calculate_rms_threshold()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_COMPRESSOR_SIDECHAIN_GAIN The current gain applied by the compressor in Q0.31 format. To read the `int32_t` control value, use the function *`qxx_to_db()`* in `control/helpers.h` with Q=31. This command is read only. When sending a write control command, it will be ignored.	`sizeof(int32_t)`
CMD_COMPRESSOR_SIDECHAIN_SLOPE The compression slope of the compressor. This is calculated as *(1 - 1 / ratio) / 2.0*. To convert a ratio to a slope, use the function *`rms_compressor_slope_from_ratio()`* in `control/helpers.h`.	`sizeof(float)`

CompressorSidechainStereo

class `audio_dsp.stages.`**CompressorSidechainStereo**(***kwargs*)

An stereo sidechain compressor based on the RMS envelope of the detect signal.

This stage is limited to accepting 4 channels. The first pair are the channels that will be compressed. The second pair are the detect channels. The level of compression depends on the maximum envelope of the detect channels.

When the maximum RMS envelope of the detect signal exceeds the threshold, the processed signal amplitudes are reduced by the compression ratio.

The threshold sets the value above which compression occurs. The ratio sets how much the signals are compressed. A ratio of 1 results in no compression, while a ratio of infinity results in the same behaviour as a limiter. The attack time sets how fast the compressor starts compressing. The release time sets how long the signal takes to ramp up to its original level after the envelope is below the threshold.

Attributes

dsp_block
[`audio_dsp.dsp.drc.sidechain.compressor_rms_sidechain_stereo`] The DSP block class; see *Stereo Sidechain RMS Compressor* for implementation details.

`make_compressor_sidechain`(*ratio*, *threshold_db*, *attack_t*, *release_t*, *Q_sig=27*)

Update compressor configuration based on new parameters.

Parameters

ratio
[float] Compression gain ratio applied when the signal is above the threshold.

threshold_db
[float] Threshold in decibels above which compression occurs.

attack_t
[float] Attack time of the compressor in seconds.

release_t
[float] Release time of the compressor in seconds.

CompressorSidechainStereo Control The following runtime command ids are available for the CompressorSidechainStereo Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_COMPRESSOR_SIDECHAIN_STEREO_ATTACK_ALPHA The attack alpha in Q0.31 format. To convert an attack time in seconds to an `int32_t` control value, use the function *`calc_alpha()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_COMPRESSOR_SIDECHAIN_STEREO_RELEASE_ALPHA The release alpha in Q0.31 format. To convert a release time in seconds to an `int32_t` control value, use the function *`calc_alpha()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_COMPRESSOR_SIDECHAIN_STEREO_ENVELOPE The current RMS² envelope of the signal in Q_SIG format. To read the `int32_t` control value, use the function *`qxx_to_db_pow()`* in `control/helpers.h` with Q=Q_SIG. This command is read only. When sending a write control command, it will be ignored.	`sizeof(int32_t)`
CMD_COMPRESSOR_SIDECHAIN_STEREO_THRESHOLD The threshold in Q_SIG format above which compression will occur. To convert a threshold in dB to the `int32_t` control value, use the function *`calculate_rms_threshold()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_COMPRESSOR_SIDECHAIN_STEREO_GAIN The current gain applied by the compressor in Q0.31 format. To read the `int32_t` control value, use the function *`qxx_to_db()`* in `control/helpers.h` with Q=31. This command is read only. When sending a write control command, it will be ignored.	`sizeof(int32_t)`
CMD_COMPRESSOR_SIDECHAIN_STEREO_SLOPE The compression slope of the compressor. This is calculated as *(1 - 1 / ratio) / 2.0*. To convert a ratio to a slope, use the function *`rms_compressor_slope_from_ratio()`* in `control/helpers.h`.	`sizeof(float)`

Envelope Detector Stages

Envelope detector Stages measure how the average or peak amplitude of a signal varies over time.

EnvelopeDetectorPeak

class audio_dsp.stages.**EnvelopeDetectorPeak**(***kwargs*)

A stage with no outputs that measures the signal peak envelope.

The current envelope of the signal can be read out using this stage's `envelope` control.

Attributes

dsp_block
[`audio_dsp.dsp.drc.drc.envelope_detector_peak`]
The DSP block class; see *Peak Envelope Detector* for implementation details.

`make_env_det_peak`(*attack_t*, *release_t*, *Q_sig=27*)

Update envelope detector configuration based on new parameters.

Parameters

attack_t
[float] Attack time of the envelope detector in seconds.

release_t
[float] Release time of the envelope detector in seconds.

EnvelopeDetectorPeak Control The following runtime command ids are available for the EnvelopeDetectorPeak Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_ENVELOPE_DETECTOR_PEAK_ATTACK_ALPHA The attack alpha in Q0.31 format. To convert an attack time in seconds to an `int32_t` control value, use the function *`calc_alpha()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_ENVELOPE_DETECTOR_PEAK_RELEASE_ALPHA The release alpha in Q0.31 format. To convert a release time in seconds to an `int32_t` control value, use the function *`calc_alpha()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_ENVELOPE_DETECTOR_PEAK_ENVELOPE The current peak envelope of the signal in Q_SIG format. To read the `int32_t` control value, use the function *`qxx_to_db()`* in `control/helpers.h` with Q=Q_SIG. This command is read only. When sending a write control command, it will be ignored.	`sizeof(int32_t)`

EnvelopeDetectorRMS

`class` `audio_dsp.stages.`**`EnvelopeDetectorRMS`**(***kwargs*)

A stage with no outputs that measures the signal RMS envelope.

The current envelope of the signal can be read out using this stage's `envelope` control.

Attributes

dsp_block
[audio_dsp.dsp.drc.drc.envelope_detector_rms]
The DSP block class; see *RMS Envelope Detector* for implementation details.

make_env_det_rms(*attack_t*, *release_t*, *Q_sig=27*)

Update envelope detector configuration based on new parameters.

Parameters

attack_t
[float] Attack time of the envelope detector in seconds.

release_t
[float] Release time of the envelope detector in seconds.

EnvelopeDetectorRMS Control The following runtime command ids are available for the EnvelopeDetectorRMS Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_ENVELOPE_DETECTOR_RMS_ATTACK_ALPHA The attack alpha in Q0.31 format. To convert an attack time in seconds to an `int32_t` control value, use the function *`calc_alpha()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_ENVELOPE_DETECTOR_RMS_RELEASE_ALPHA The release alpha in Q0.31 format. To convert a release time in seconds to an `int32_t` control value, use the function *`calc_alpha()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_ENVELOPE_DETECTOR_RMS_ENVELOPE The current RMS² envelope of the signal in Q_SIG format. To read the `int32_t` control value, use the function *`qxx_to_db_pow()`* in `control/helpers.h` with Q=Q_SIG. This command is read only. When sending a write control command, it will be ignored.	`sizeof(int32_t)`

FIR Stages

Finite impulse response (FIR) filter Stages allow the use of arbitrary filters with a finite number of taps.

FirDirect

class audio_dsp.stages.**FirDirect**(*coeffs_path*, ***kwargs*)

A FIR filter implemented in the time domain. The input signal is convolved with the filter coefficients. The filter coefficients can only be set at compile time.

Parameters

coeffs_path
[Path] Path to a file containing the coefficients, in a format supported by np.loadtxt.

Attributes

dsp_block
[`audio_dsp.dsp.fir.fir_direct`] The DSP block class; see *FIR Direct* for implementation details.

`make_fir_direct`(*coeffs_path*, *Q_sig=27*)
Update FIR configuration based on new parameters.

Parameters

coeffs_path
[Path] Path to a file containing the coefficients, in a format supported by np.loadtxt.

FirDirect Control The FirDirect Stage has no runtime controllable parameters.

Graphic EQ Stages

Graphic EQs allow frequency response adjustments at fixed center frequencies.

GraphicEq10b
`class audio_dsp.stages.GraphicEq10b`(***kwargs*)
A 10 band graphic equaliser, with octave spaced center frequencies. The center frequencies are: [32, 64, 125, 250, 500, 1000, 2000, 4000, 8000, 16000]. The gain of each band can be adjusted between -12 and + 12 dB.

Note that for a 32 kHz sample rate, the 16 kHz band is not available, making a 9 band EQ. For a 16 kHz sample rate the 8k and 16 kHz bands are not available, making an 8 band EQ.

Attributes

dsp_block
[`audio_dsp.dsp.graphic_eq.graphic_eq_10_band`] The DSP block class; see *10 Band Graphic Equaliser* for implementation details

`set_gains`(*gains_db*)
Set the gains of the graphic eq in dB.

Parameters

gains_db
[list[float]] A list of the 10 gains of the graphic eq in dB.

GraphicEq10b Control The following runtime command ids are available for the GraphicEq10b Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_GRAPHIC_EQ10B_GAINS The graphic EQ gains in Q31 format. To convert a value in decibels to this format, the function *`geq_db_to_gain()`* in `control/helpers.h`.	`sizeof(int32_t)*[10]`

Limiter Stages

Limiter Stages allow the amplitude of the signal to be restricted based on its envelope.

LimiterRMS

class audio_dsp.stages.**LimiterRMS**(***kwargs*)

A limiter based on the RMS value of the signal. When the RMS envelope of the signal exceeds the threshold, the signal amplitude is reduced.

The threshold sets the value above which limiting occurs. The attack time sets how fast the limiter starts limiting. The release time sets how long the signal takes to ramp up to its original level after the envelope is below the threshold.

Attributes

dsp_block
[audio_dsp.dsp.drc.drc.limiter_rms] The DSP block class; see *RMS Limiter* for implementation details.

make_limiter_rms(*threshold_db*, *attack_t*, *release_t*, *Q_sig=27*)

Update limiter configuration based on new parameters.

Parameters

threshold_db
[float] Threshold in decibels above which limiting occurs.

attack_t
[float] Attack time of the limiter in seconds.

release_t
[float] Release time of the limiter in seconds.

LimiterRMS Control The following runtime command ids are available for the LimiterRMS Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_LIMITER_RMS_ATTACK_ALPHA The attack alpha in Q0.31 format. To convert an attack time in seconds to an `int32_t` control value, use the function *`calc_alpha()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_LIMITER_RMS_RELEASE_ALPHA The release alpha in Q0.31 format. To convert a release time in seconds to an `int32_t` control value, use the function *`calc_alpha()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_LIMITER_RMS_ENVELOPE The current RMS² envelope of the signal in Q_SIG format. To read the `int32_t` control value, use the function *`qxx_to_db_pow()`* in `control/helpers.h` with Q=Q_SIG. This command is read only. When sending a write control command, it will be ignored.	`sizeof(int32_t)`
CMD_LIMITER_RMS_THRESHOLD The threshold in Q_SIG format above which limiting will occur. To convert a threshold in dB to the `int32_t` control value, use the function *`calculate_rms_threshold()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_LIMITER_RMS_GAIN The current gain applied by the limiter in Q0.31 format. To read the `int32_t` control value, use the function *`qxx_to_db()`* in `control/helpers.h` with Q=31. This command is read only. When sending a write control command, it will be ignored.	`sizeof(int32_t)`

LimiterPeak

class `audio_dsp.stages.`**LimiterPeak**(***kwargs*)

A limiter based on the peak value of the signal. When the peak envelope of the signal exceeds the threshold, the signal amplitude is reduced.

The threshold sets the value above which limiting occurs. The attack time sets how fast the limiter starts limiting. The release time sets how long the signal takes to ramp up to its original level after the envelope is below the threshold.

Attributes

dsp_block
[`audio_dsp.dsp.drc.drc.limiter_peak`] The DSP block class; see *Peak Limiter* for implementation details.

make_limiter_peak(*threshold_db*, *attack_t*, *release_t*, *Q_sig=27*)

Update limiter configuration based on new parameters.

Parameters

threshold_db
[float] Threshold in decibels above which limiting occurs.

attack_t
[float] Attack time of the limiter in seconds.

release_t
[float] Release time of the limiter in seconds.

LimiterPeak Control The following runtime command ids are available for the LimiterPeak Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_LIMITER_PEAK_ATTACK_ALPHA The attack alpha in Q0.31 format. To convert an attack time in seconds to an `int32_t` control value, use the function *`calc_alpha()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_LIMITER_PEAK_RELEASE_ALPHA The release alpha in Q0.31 format. To convert a release time in seconds to an `int32_t` control value, use the function *`calc_alpha()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_LIMITER_PEAK_ENVELOPE The current peak envelope of the signal in Q_SIG format. To read the `int32_t` control value, use the function *`qxx_to_db()`* in `control/helpers.h` with Q=Q_SIG. This command is read only. When sending a write control command, it will be ignored.	`sizeof(int32_t)`
CMD_LIMITER_PEAK_THRESHOLD The threshold in Q_SIG format above which limiting will occur. To convert a threshold in dB to the `int32_t` control value, use the function *`calculate_peak_threshold()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_LIMITER_PEAK_GAIN The current gain applied by the limiter in Q0.31 format. To read the `int32_t` control value, use the function *`qxx_to_db()`* in `control/helpers.h` with Q=31. This command is read only. When sending a write control command, it will be ignored.	`sizeof(int32_t)`

HardLimiterPeak
`class audio_dsp.stages.HardLimiterPeak(`***kwargs*`)`

A limiter based on the peak value of the signal. The peak envelope of the signal may never exceed the threshold.

When the peak envelope of the signal exceeds the threshold, the signal amplitude is reduced. If the signal still exceeds the threshold, it is clipped.

The threshold sets the value above which limiting/clipping occurs. The attack time sets how fast the limiter starts limiting. The release time sets how long the signal takes to ramp up to its original level after the envelope is below the threshold.

Attributes

dsp_block
[`audio_dsp.dsp.drc.drc.hard_limiter_peak`] The DSP block class; see *Hard Peak Limiter* for implementation details.

`make_hard_limiter_peak`(*threshold_db*, *attack_t*, *release_t*, *Q_sig=27*)

Update limiter configuration based on new parameters.

Parameters

threshold_db
[float] Threshold in decibels above which limiting occurs.

attack_t
[float] Attack time of the limiter in seconds.

release_t
[float] Release time of the limiter in seconds.

HardLimiterPeak Control The following runtime command ids are available for the HardLimiterPeak Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_HARD_LIMITER_PEAK_ATTACK_ALPHA The attack alpha in Q0.31 format. To convert an attack time in seconds to an `int32_t` control value, use the function *`calc_alpha()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_HARD_LIMITER_PEAK_RELEASE_ALPHA The release alpha in Q0.31 format. To convert a release time in seconds to an `int32_t` control value, use the function *`calc_alpha()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_HARD_LIMITER_PEAK_ENVELOPE The current peak envelope of the signal in Q_SIG format. To read the `int32_t` control value, use the function *`qxx_to_db()`* in `control/helpers.h` with Q=Q_SIG. This command is read only. When sending a write control command, it will be ignored.	`sizeof(int32_t)`
CMD_HARD_LIMITER_PEAK_THRESHOLD The threshold in Q_SIG format above which limiting will occur. To convert a threshold in dB to the `int32_t` control value, use the function *`calculate_peak_threshold()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_HARD_LIMITER_PEAK_GAIN The current gain applied by the limiter in Q0.31 format. To read the `int32_t` control value, use the function *`qxx_to_db()`* in `control/helpers.h` with Q=31. This command is read only. When sending a write control command, it will be ignored.	`sizeof(int32_t)`

Clipper

class `audio_dsp.stages.Clipper`(***kwargs*)

A simple clipper that limits the signal to a specified threshold.

If the signal is greater than the threshold level, it is set to the threshold value.

Attributes

dsp_block

[`audio_dsp.dsp.drc.drc.clipper`] The DSP block class; see *Clipper* for implementation details.

`make_clipper`(*threshold_db*, *Q_sig=27*)

Update clipper configuration based on new parameters.

Parameters

threshold_db

[float] Threshold in decibels above which clipping occurs.

Clipper Control The following runtime command ids are available for the Clipper Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_CLIPPER_THRESHOLD The threshold in Q_SIG format above which clipping will occur. To convert a threshold in dB to the `int32_t` control value, use the function *`calculate_peak_threshold()`* in `control/helpers.h`.	`sizeof(int32_t)`

Noise Gate Stages

Noise gate Stages remove quiet signals from the audio output.

NoiseGate

class `audio_dsp.stages.`**NoiseGate**(***kwargs*)

A noise gate that reduces the level of an audio signal when it falls below a threshold.

When the signal envelope falls below the threshold, the gain applied to the signal is reduced to 0 over the release time. When the envelope returns above the threshold, the gain applied to the signal is increased to 1 over the attack time.

The initial state of the noise gate is with the gate open (no attenuation); this models a full scale signal having been present before t = 0.

Attributes

dsp_block
[`audio_dsp.dsp.drc.expander.noise_gate`] The DSP block class; see *Noise Gate* for implementation details.

make_noise_gate(*threshold_db*, *attack_t*, *release_t*, *Q_sig=27*)

Update noise gate configuration based on new parameters.

Parameters

threshold_db
[float] The threshold level in decibels below which the audio signal is attenuated.

attack_t
[float] Attack time of the noise gate in seconds.

release_t
[float] Release time of the noise gate in seconds.

NoiseGate Control The following runtime command ids are available for the NoiseGate Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_NOISE_GATE_ATTACK_ALPHA The attack alpha in Q0.31 format. To convert an attack time in seconds to an `int32_t` control value, use the function *calc_alpha()* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_NOISE_GATE_RELEASE_ALPHA The release alpha in Q0.31 format. To convert a release time in seconds to an `int32_t` control value, use the function *calc_alpha()* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_NOISE_GATE_ENVELOPE The current peak envelope of the signal in Q_SIG format. To read the `int32_t` control value, use the function *qxx_to_db()* with Q=Q_SIG in `control/helpers.h` This command is read only. When sending a write control command, it will be ignored.	`sizeof(int32_t)`
CMD_NOISE_GATE_THRESHOLD The threshold in Q_SIG format below which gating will occur. To convert a threshold in dB to the `int32_t` control value, use the function *calculate_peak_threshold()* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_NOISE_GATE_GAIN The current gain applied by the noise gate in Q0.31 format. To read the `int32_t` control value, use the function *qxx_to_db()* with Q=31 in `control/helpers.h` This command is read only. When sending a write control command, it will be ignored.	`sizeof(int32_t)`

Noise Suppressor Expander Stages

Noise suppressor and expander Stages control the behaviour of quiet signals, typically by tring to reduce the audibility of noise in the signal.

NoiseSuppressorExpander

class `audio_dsp.stages.`**NoiseSuppressorExpander**(***kwargs*)

The Noise Suppressor (Expander) stage. A noise suppressor that reduces the level of an audio signal when it falls below a threshold. This is also known as an expander.

When the signal envelope falls below the threshold, the gain applied to the signal is reduced relative to the expansion ratio over the release time. When the envelope returns above the threshold, the gain applied to the signal is increased to 1 over the attack time.

The initial state of the noise suppressor is with the suppression off; this models a full scale signal having been present before t = 0.

Attributes

dsp_block
[`audio_dsp.dsp.drc.expander.noise_suppressor_expander`] The DSP block class; see *Noise Suppressor/Expander* for implementation details.

`make_noise_suppressor_expander`(*ratio*, *threshold_db*, *attack_t*, *release_t*, *Q_sig=27*)

Update noise suppressor (expander) configuration based on new parameters. All parameters are passed to the constructor of `audio_dsp.dsp.drc.noise_suppressor_expander`.

Parameters

ratio
[float] The expansion ratio applied to the signal when the envelope falls below the threshold.

threshold_db
[float] The threshold level in decibels below which the audio signal is attenuated.

attack_t
[float] Attack time of the noise suppressor in seconds.

release_t
[float] Release time of the noise suppressor in seconds.

NoiseSuppressorExpander Control The following runtime command ids are available for the NoiseSuppressorExpander Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_NOISE_SUPPRESSOR_EXPANDER_ATTACK_ALPHA The attack alpha in Q0.31 format. To convert an attack time in seconds to an `int32_t` control value, use the function *`calc_alpha()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_NOISE_SUPPRESSOR_EXPANDER_RELEASE_ALPHA The release alpha in Q0.31 format. To convert a release time in seconds to an `int32_t` control value, use the function *`calc_alpha()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_NOISE_SUPPRESSOR_EXPANDER_ENVELOPE The current peak envelope of the signal in Q_SIG format. To read the `int32_t` control value, use the function *`qxx_to_db()`* in `control/helpers.h` with Q=Q_SIG. This command is read only. When sending a write control command, it will be ignored.	`sizeof(int32_t)`
CMD_NOISE_SUPPRESSOR_EXPANDER_THRESHOLD The threshold in Q_SIG format below which suppression will occur. To convert a threshold in dB to the `int32_t` control value, use the function *`calculate_peak_threshold()`* in `control/helpers.h`.	`sizeof(int32_t)`
CMD_NOISE_SUPPRESSOR_EXPANDER_GAIN The current gain applied by the noise suppressor in Q0.31 format. To read the `int32_t` control value, use the function *`qxx_to_db()`* in `control/helpers.h` with Q=31. This command is read only. When sending a write control command, it will be ignored.	`sizeof(int32_t)`
CMD_NOISE_SUPPRESSOR_EXPANDER_SLOPE The expansion slope of the noise suppressor. This is calculated as *(1 - ratio)*.To convert a ratio to a slope, use the function *`peak_expander_slope_from_ratio()`* in `control/helpers.h`.	`sizeof(float)`

Reverb Stages

Reverb Stages emulate the natural reverberance of rooms.

ReverbRoom

`class audio_dsp.stages.ReverbRoom(`*max_room_size=1, predelay=10, max_predelay=None, **kwargs*`)`

The room reverb stage. This is based on Freeverb by Jezar at Dreampoint, and consists of 8 parallel comb filters fed into 4 series all-pass filters.

Parameters

max_room_size
Sets the maximum room size for this reverb. The `room_size` parameter sets the fraction of this value actually used at any given time. For optimal memory usage, max_room_size should be set so that the longest reverb tail occurs when `room_size=1.0`.

predelay
[float, optional] The delay applied to the wet channel in ms.

max_predelay
[float, optional] The maximum predelay in ms.

Attributes

dsp_block
[`audio_dsp.dsp.reverb.reverb_room`] The DSP block class; see *Reverb Room* for implementation details.

`set_damping`(*damping*)

Set the damping of the reverb room stage. This controls how much high frequency attenuation is in the room. Higher values yield shorter reverberation times at high frequencies.

Parameters

damping
[float] How much high frequency attenuation in the room, between 0 and 1.

`set_decay`(*decay*)

Set the decay of the reverb room stage. This sets how reverberant the room is. Higher values will give a longer reverberation time for a given room size.

Parameters

decay
[float] How long the reverberation of the room is, between 0 and 1.

`set_dry_gain`(*gain_dB*)

Set the dry gain of the reverb room stage. This sets the level of the unprocessed signal.

Parameters

gain_db
[float] Dry gain in dB, less than 0 dB.

`set_pre_gain`(*pre_gain*)

Set the pre gain of the reverb room stage.

Parameters

pre_gain
[float] Pre gain value. Must be less than 1.

set_predelay(*predelay*)

Set the predelay of the wet channel.

Parameters

predelay
[float] Predelay in ms, less than max_predelay.

set_room_size(*new_room_size*)

Set the room size, will adjust the delay line lengths.
The room size is proportional to **max_room_size**, and must be between 0 and 1. To increase the room_size above 1.0, **max_room_size** must instead be increased. Optimal memory usage occurs when **room_size** is set to 1.0.

Parameters

new_room_size
[float] How big the room is as a proportion of max_room_size. This sets delay line lengths and must be between 0 and 1.

set_wet_dry_mix(*mix*)

Set the wet/dry gains so that the mix of 0 results in a fully dry output, the mix of 1 results in a fully wet output.

Parameters

mix
[float] The wet/dry mix, must be [0, 1].

set_wet_gain(*gain_dB*)

Set the wet gain of the reverb room stage. This sets the level of the reverberated signal.

Parameters

gain_db
[float] Wet gain in dB, less than 0 dB.

ReverbRoom Control The following runtime command ids are available for the ReverbRoom Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_REVERB_ROOM_ROOM_SIZE How big the room is as a proportion of max_room_size. This sets delay line lengths and must be between 0 and 1.	`sizeof(float)`
CMD_REVERB_ROOM_FEEDBACK feedback gain in Q0.31 format. Feedback can be calculated from decay as *(0.28 decay) + 0.7*. Use the function *`adsp_reverb_calculate_feedback()`* in `control/reverb.h`.	`sizeof(int32_t)`

continues on next page

Table 1 – continued from previous page

Control parameter	Payload length
CMD_REVERB_ROOM_DAMPING High frequency attenuation in Q0.31 format. Use the function *adsp_reverb_calculate_damping()* in `control/reverb.h`.	`sizeof(int32_t)`
CMD_REVERB_ROOM_WET_GAIN Gain applied to the wet signal in Q0.31 format. Use the function *adsp_reverb_db2int()* in `control/reverb.h`. Alternatively, both wet and dry gains can be obtained from *adsp_reverb_wet_dry_mix()*.	`sizeof(int32_t)`
CMD_REVERB_ROOM_DRY_GAIN Dry signal gain in Q0.31 format. Use the function *adsp_reverb_db2int()* in `control/reverb.h`. Alternatively, both wet and dry gains can be obtained from *adsp_reverb_wet_dry_mix()*.	`sizeof(int32_t)`
CMD_REVERB_ROOM_PREGAIN The pregain applied to the signal before the reverb. Changing this value is not recommended. Use the function *adsp_reverb_float2int()* in `control/reverb.h`.	`sizeof(int32_t)`
CMD_REVERB_ROOM_PREDELAY Predelay applied to the wet channel in samples. To convert a value in other units of time to samples, use *time_to_samples()* in `control/signal_chain.h`.	`sizeof(uint32_t)`

ReverbRoomStereo

class audio_dsp.stages.**ReverbRoomStereo**(*max_room_size=1*, *predelay=10*, *max_predelay=None*, ***kwargs*)

The stereo room reverb stage. This is based on Freeverb by Jezar at Dreampoint. Each channel consists of 8 parallel comb filters fed into 4 series all-pass filters, and the reverberator outputs are mixed according to the `width` parameter.

Parameters

max_room_size
Sets the maximum room size for this reverb. The `room_size` parameter sets the fraction of this value actually used at any given time. For optimal memory usage, max_room_size should be set so that the longest reverb tail occurs when `room_size=1.0`.

predelay
[float, optional] The delay applied to the wet channel in ms.

max_predelay
[float, optional] The maximum predelay in ms.

Attributes

dsp_block
[`audio_dsp.dsp.reverb_stereo.reverb_room_stereo`] The DSP block class; see *Reverb Room Stereo* for implementation details.

`set_damping`(*damping*)

Set the damping of the reverb room stage. This controls how much high frequency attenuation is in the room. Higher values yield shorter reverberation times at high frequencies.

Parameters

damping
[float] How much high frequency attenuation in the room, between 0 and 1.

`set_decay`(*decay*)

Set the decay of the reverb room stage. This sets how reverberant the room is. Higher values will give a longer reverberation time for a given room size.

Parameters

decay
[float] How long the reverberation of the room is, between 0 and 1.

`set_dry_gain`(*gain_dB*)

Set the dry gain of the reverb room stage. This sets the level of the unprocessed signal.

Parameters

gain_db
[float] Dry gain in dB, less than 0 dB.

`set_pre_gain`(*pre_gain*)

Set the pre gain of the reverb room stage.

Parameters

pre_gain
[float] Pre gain value. Must be less than 1.

`set_predelay`(*predelay*)

Set the predelay of the wet channel.

Parameters

predelay
[float] Predelay in ms, less than max_predelay.

`set_room_size`(*new_room_size*)

Set the room size, will adjust the delay line lengths.

The room size is proportional to `max_room_size`, and must be between 0 and 1. To increase the room_size above 1.0, `max_room_size` must instead be increased. Optimal memory usage occurs when `room_size` is set to 1.0.

Parameters

new_room_size
[float] How big the room is as a proportion of max_room_size. This sets delay line lengths and must be between 0 and 1.

`set_wet_dry_mix`(*mix*)

Set the wet/dry gains so that the mix of 0 results in a fully dry output, the mix of 1 results in a fully wet output.

Parameters

mix
[float] The wet/dry mix, must be [0, 1].

`set_wet_gain`(*gain_dB*)

Set the wet gain of the reverb room stage. This sets the level of the reverberated signal.

Parameters

gain_db
[float] Wet gain in dB, less than 0 dB.

`set_width`(*width*)

Set the decay of the reverb room stage. This sets how reverberant the room is. Higher values will give a longer reverberation time for a given room size.

Parameters

width
[float] How much stereo separation between the channels. A width of 0 indicates no stereo separation (i.e. mono). A width of 1 indicates maximum stereo separation.

ReverbRoomStereo Control The following runtime command ids are available for the ReverbRoomStereo Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_REVERB_ROOM_STEREO_ROOM_SIZE How big the room is as a proportion of max_room_size. This sets delay line lengths and must be between 0 and 1.	`sizeof(float)`
CMD_REVERB_ROOM_STEREO_FEEDBACK feedback gain in Q0.31 format. Feedback can be calculated from decay as *(0.28 decay) + 0.7*. Use the function *`adsp_reverb_calculate_feedback()`* in `control/reverb.h`.	`sizeof(int32_t)`
CMD_REVERB_ROOM_STEREO_DAMPING	`sizeof(int32_t)`

continues on next page

Table 2 – continued from previous page

Control parameter	Payload length
High frequency attenuation in Q0.31 format. Use the function *adsp_reverb_calculate_damping()* in `control/reverb.h`.	
CMD_REVERB_ROOM_STEREO_WET_GAIN1 Gain applied to obtain the wet signal in Q0.31 format. Use function *adsp_reverb_room_st_calc_wet_gains()* in `control/reverb.h`. Alternatively, all gains can be obtained from *adsp_reverb_st_wet_dry_mix()*.	`sizeof(int32_t)`
CMD_REVERB_ROOM_STEREO_WET_GAIN2 Gain applied to obtain the wet signal in Q0.31 format. Use function *adsp_reverb_room_st_calc_wet_gains()* in `control/reverb.h`. Alternatively, all gains can be obtained from *adsp_reverb_st_wet_dry_mix()*.	`sizeof(int32_t)`
CMD_REVERB_ROOM_STEREO_DRY_GAIN Dry signal gain in Q0.31 format. Use the function *adsp_reverb_db2int()* in `control/reverb.h`. Alternatively, all gains can be obtained from *adsp_reverb_st_wet_dry_mix()*.	`sizeof(int32_t)`
CMD_REVERB_ROOM_STEREO_PREGAIN The pregain applied to the signal before the reverb. Changing this value is not recommended. Use the function *adsp_reverb_float2int()* in `control/reverb.h`.	`sizeof(int32_t)`
CMD_REVERB_ROOM_STEREO_PREDELAY Predelay applied to the wet channel in samples. To convert a value in other units of time to samples, use *time_to_samples()* in `control/signal_chain.h`.	`sizeof(uint32_t)`

ReverbPlateStereo

class audio_dsp.stages.**ReverbPlateStereo**(*predelay=10*, *max_predelay=None*, ***kwargs*)

The stereo room plate stage. This is based on Dattorro's 1997 paper. This reverb consists of 4 allpass filters for input diffusion, followed by a figure of 8 reverb tank of allpasses, low-pass filters, and delays. The output is taken from multiple taps in the delay lines to get a desirable echo density.

Parameters

predelay
[float, optional] The delay applied to the wet channel in ms.

max_predelay
[float, optional] The maximum predelay in ms.

Attributes

dsp_block
[`audio_dsp.dsp.reverb.reverb_plate_stereo`] The DSP block class; see *Reverb Plate Stereo* for implementation details.

`set_bandwidth`(*bandwidth*)

Set the bandwidth of the plate reverb stage. This sets the low pass cutoff frequency of the reverb input. Higher values will give a higher cutoff frequency.

Parameters

bandwidth
[float] The bandwidth of the plate input signal, between 0 and 1.

`set_damping`(*damping*)

Set the damping of the plate reverb stage. This controls how much high frequency attenuation is in the plate. Higher values yield shorter reverberation times at high frequencies.

Parameters

damping
[float] How much high frequency attenuation in the plate, between 0 and 1.

`set_decay`(*decay*)

Set the decay of the plate reverb stage. This sets how reverberant the plate is. Higher values will give a longer reverberation time.

Parameters

decay
[float] How long the reverberation of the plate is, between 0 and 1.

`set_dry_gain`(*gain_dB*)

Set the dry gain of the reverb room stage. This sets the level of the unprocessed signal.

Parameters

gain_db
[float] Dry gain in dB, less than 0 dB.

`set_early_diffusion`(*diffusion*)

Set the early diffusion of the plate reverb stage. This sets how much diffusion is present in the first part of the reverberation. Higher values will give more diffusion.

Parameters

diffusion
[float] How diffuse the plate is, between 0 and 1.

set_late_diffusion(*diffusion*)

Set the late diffusion of the plate reverb stage. This sets how much diffusion is present in the latter part of the reverberation. Higher values will give more diffusion.

Parameters

diffusion
[float] How diffuse the plate is, between 0 and 1.

set_pre_gain(*pre_gain*)

Set the pre gain of the reverb room stage.

Parameters

pre_gain
[float] Pre gain value. Must be less than 1.

set_predelay(*predelay*)

Set the predelay of the wet channel.

Parameters

predelay
[float] Predelay in ms, less than max_predelay.

set_wet_dry_mix(*mix*)

Set the wet/dry gains so that the mix of 0 results in a fully dry output, the mix of 1 results in a fully wet output.

Parameters

mix
[float] The wet/dry mix, must be [0, 1].

set_wet_gain(*gain_dB*)

Set the wet gain of the reverb room stage. This sets the level of the reverberated signal.

Parameters

gain_db
[float] Wet gain in dB, less than 0 dB.

set_width(*width*)

Set the decay of the reverb room stage. This sets how reverberant the room is. Higher values will give a longer reverberation time for a given room size.

Parameters

width
[float] How much stereo separation between the channels. A width of 0 indicates no stereo separation (i.e. mono). A width of 1 indicates maximum stereo separation.

ReverbPlateStereo Control The following runtime command ids are available for the ReverbPlateStereo Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_REVERB_PLATE_STEREO_DECAY The amount of decay in the plate in Q0.31 format. To convert a decay value between 0 and 1 to an `int32_t` control value, use the function *adsp_reverb_float2int()* in `control/reverb.h`.	`sizeof(int32_t)`
CMD_REVERB_PLATE_STEREO_DAMPING The amount of high frequency attenuation in Q0.31 format. To convert a damping value between 0 and 1 to an `int32_t` control value, use the function *adsp_reverb_plate_calc_damping()* in `control/reverb.h`.	`sizeof(int32_t)`
CMD_REVERB_PLATE_STEREO_EARLY_DIFFUSION The amount of diffusion in the early part of the reverb. To convert a diffusion value between 0 and 1 to an `int32_t` control value, use the function *adsp_reverb_float2int()* in `control/reverb.h`.	`sizeof(int32_t)`
CMD_REVERB_PLATE_STEREO_LATE_DIFFUSION The amount of diffusion in the late part of the reverb. To convert a diffusion value between 0 and 1 to an `int32_t` control value, use the function *adsp_reverb_plate_calc_late_diffusion()* in `control/reverb_plate.h`.	`sizeof(int32_t)`
CMD_REVERB_PLATE_STEREO_BANDWIDTH The input low pass coefficient in Q0.31 format. A bandwidth in Hertz can be converted to an `int32_t` control value using the function *adsp_reverb_plate_calc_bandwidth()* in `control/reverb_plate.h`.	`sizeof(int32_t)`
CMD_REVERB_PLATE_STEREO_WET_GAIN1 Gain applied to the wet signal in Q0.31 format. To calculate the wet gains based on a level in dB and a stereo width, use the function *adsp_reverb_room_st_calc_wet_gains()* in `control/reverb.h`. Alternatively, wet and dry gains can be calculated using a wet/dry ratio with the function *adsp_reverb_st_wet_dry_mix()*.	`sizeof(int32_t)`
CMD_REVERB_PLATE_STEREO_WET_GAIN2	`sizeof(int32_t)`

continues on next page

Table 3 – continued from previous page

Control parameter	Payload length
Gain applied to the wet signal in Q0.31 format. To calculate the wet gains based on a level in dB and a stereo width, use the function *adsp_reverb_room_st_calc_wet_gains()* in control/reverb.h. Alternatively, wet and dry gains can be calculated using a wet/dry ratio with the function *adsp_reverb_st_wet_dry_mix()*.	
CMD_REVERB_PLATE_STEREO_DRY_GAIN Gain applied to the dry signal in Q0.31 format. To calculate the dry gain based on a level in dB, use the function *adsp_reverb_db2int()* in control/reverb.h. Alternatively, wet and dry gains can be calculated using a wet/dry ratio with the function *adsp_reverb_st_wet_dry_mix()*.	sizeof(int32_t)
CMD_REVERB_PLATE_STEREO_PREGAIN The pregain applied to the signal before the reverb. Changing this value is only required if saturation occurs in the reverb tank. To convert a linear gain value to an int32_t control value, use the function *adsp_reverb_float2int()* in control/reverb.h.	sizeof(int32_t)
CMD_REVERB_PLATE_STEREO_PREDELAY The wet channel predelay value in samples. To convert a value in other units of time to samples, use *time_to_samples()* in control/signal_chain.h. Note the minimum delay provided by this stage is 1 sample. Setting the delay to 0 will still yield a 1 sample delay.	sizeof(uint32_t)

Signal Chain Stages

Signal chain stages allow for the control of signal flow through the pipeline. This includes stages for combining and splitting signals, basic gain components, and delays.

Bypass

class audio_dsp.stages.**Bypass**(***kwargs*)

Stage which does not modify its inputs. Useful if data needs to flow through a thread which is not being processed on to keep pipeline lengths aligned.

process(*in_channels*)

Return a copy of the inputs.

Bypass Control The Bypass Stage has no runtime controllable parameters.

Fork

class audio_dsp.stages.**Fork**(*count=2*, ***kwargs*)

Fork the signal.

Use if the same data needs to be sent to multiple data paths:

```
a = t.stage(Example, ...)
f = t.stage(Fork, a, count=2)  # count optional, default is 2
b = t.stage(Example, f.forks[0])
c = t.stage(Example, f.forks[1])
```

Attributes

forks
[list[list[StageOutput]]] For convenience, each forked output will be available in this list each entry contains a set of outputs which will contain the same data as the input.

class ForkOutputList(*edges: list[audio_dsp.design.stage.StageOutput | None] | None = None*)

Custom StageOutputList that is created by Fork.
This allows convenient access to each fork output.

Attributes

forks: list[StageOutputList]
Fork duplicates its inputs, each entry in the forks list is a single copy of the input edges.

get_frequency_response(*nfft=32768*)

Fork has no sensible frequency response, not implemented.

process(*in_channels*)

Duplicate the inputs to the outputs based on this fork's configuration.

Fork Control The Fork Stage has no runtime controllable parameters.

Mixer

class audio_dsp.stages.**Mixer**(***kwargs*)

Mixes the input signals together. The mixer can be used to add signals together, or to attenuate the input signals.

Attributes

dsp_block
[audio_dsp.dsp.signal_chain.mixer] The DSP block class; see *Mixer* for implementation details

set_gain(*gain_db*)

Set the gain of the mixer in dB.

Parameters

gain_db
[float] The gain of the mixer in dB.

Mixer Control The following runtime command ids are available for the Mixer Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_MIXER_GAIN The current gain in Q_GAIN format. To convert a value in decibels to this format, the function *adsp_dB_to_gain()* in **control/signal_chain.h** can be used.	`sizeof(int32_t)`

Adder

class audio_dsp.stages.**Adder**(***kwargs*)

Add the input signals together. The adder can be used to add signals together.

Attributes

dsp_block

[**audio_dsp.dsp.signal_chain.adder**] The DSP block class; see *Adder* for implementation details.

Adder Control The Adder Stage has no runtime controllable parameters.

Subtractor

class audio_dsp.stages.**Subtractor**(***kwargs*)

Subtract the second input from the first. The subtractor can be used to subtract signals from each other. It must only have 2 inputs.

Attributes

dsp_block

[**audio_dsp.dsp.signal_chain.subtractor**] The DSP block class; see *Subtractor* for implementation details.

Subtractor Control The Subtractor Stage has no runtime controllable parameters.

FixedGain

class audio_dsp.stages.**FixedGain**(*gain_db=0*, ***kwargs*)

This stage implements a fixed gain. The input signal is multiplied by a gain. If the gain is changed at runtime, pops and clicks may occur.

If the gain needs to be changed at runtime, use a **VolumeControl** stage instead.

Parameters

gain_db

[float, optional] The gain of the mixer in dB.

Attributes

dsp_block

[**audio_dsp.dsp.signal_chain.fixed_gain**] The DSP block class; see *Fixed Gain* for implementation details.

set_gain(*gain_db*)

Set the gain of the fixed gain in dB.

Parameters

gain_db

[float] The gain of the fixed gain in dB.

FixedGain Control The following runtime command ids are available for the FixedGain Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_FIXED_GAIN_GAIN The gain value in Q_GAIN format. To convert a value in decibels to this format, the function *adsp_dB_to_gain()* in `control/signal_chain.h` can be used.	`sizeof(int32_t)`

VolumeControl

class `audio_dsp.stages.VolumeControl`(*gain_dB=0*, *mute_state=0*, ***kwargs*)

This stage implements a volume control. The input signal is multiplied by a gain. The gain can be changed at runtime. To avoid pops and clicks during gain changes, a slew is applied to the gain update. The stage can be muted and unmuted at runtime.

Parameters

gain_db
[float, optional] The gain of the mixer in dB.

mute_state
[int, optional] The mute state of the Volume Control: 0: unmuted, 1: muted.

Attributes

dsp_block
[`audio_dsp.dsp.signal_chain.volume_control`] The DSP block class; see *Volume Control* for implementation details.

`make_volume_control`(*gain_dB*, *slew_shift*, *mute_state*, *Q_sig=27*)

Update the settings of this volume control.

Parameters

gain_dB
Target gain of this volume control.

slew_shift
The shift value used in the exponential slew.

mute_state
The mute state of the Volume Control: 0: unmuted, 1: muted.

`set_gain`(*gain_dB*)

Set the gain of the volume control in dB.

Parameters

gain_db
[float] The gain of the volume control in dB.

`set_mute_state`(*mute_state*)

Set the mute state of the volume control.

Parameters

mute_state
[bool] The mute state of the volume control.

VolumeControl Control The following runtime command ids are available for the VolumeControl Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_VOLUME_CONTROL_TARGET_GAIN The target gain of the volume control in Q_GAIN format. To convert a value in decibels to this format, the function *adsp_dB_to_gain()* in **control/signal_chain.h** can be used.	`sizeof(int32_t)`
CMD_VOLUME_CONTROL_GAIN The current applied gain of the volume control in Q_GAIN format. The volume control will slew the applied gain towards the target gain. This command is read only. When sending a write control command, it will be ignored.	`sizeof(int32_t)`
CMD_VOLUME_CONTROL_SLEW_SHIFT The shift value used to set the slew rate. See the volume control documentation for conversions between slew_shift and time constant.	`sizeof(int32_t)`
CMD_VOLUME_CONTROL_MUTE_STATE Sets the mute state. 1 is muted and 0 is unmuted.	`sizeof(uint8_t)`

Switch

class audio_dsp.stages.**Switch**(*index=0*, ***kwargs*)

Switch the output to one of the inputs. The switch can be used to select between different signals.

Parameters

index
[int] The position to which to move the switch. This changes the output signal to the input[index]

move_switch(*position*)

Move the switch to the specified position.

Parameters

position
[int] The position to which to move the switch. This changes the output signal to the input[position]

Switch Control The following runtime command ids are available for the Switch Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_SWITCH_POSITION The current switch position.	`sizeof(int32_t)`

SwitchSlew

class `audio_dsp.stages.SwitchSlew`(*index=0*, ***kwargs*)

Switch the output to one of the inputs. The switch can be used to select between different signals. When the switch is move, a cosine slew is used to avoid clicks. This supports up to 16 inputs.

Parameters

index
[int] The position to which to move the switch. This changes the output signal to the input[index]

Attributes

dsp_block
[`audio_dsp.dsp.signal_chain.switch_slew`] The DSP block class; see *Switch with slew* for implementation details.

`move_switch`(*position*)

Move the switch to the specified position.

Parameters

position
[int] The position to which to move the switch. This changes the output signal to the input[position]

SwitchSlew Control The following runtime command ids are available for the SwitchSlew Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_SWITCH_SLEW_POSITION The current switch position.	`sizeof(int32_t)`

SwitchStereo

class `audio_dsp.stages.SwitchStereo`(*index=0*, ***kwargs*)

Switch the input to one of the stereo pairs of outputs. The switch can be used to select between different stereo signal pairs. The inputs should be passed in pairs, e.g. `[0_L, 0_R, 1_L, 1_R, ...]`. Setting the switch position will output the nth pair.

Parameters

index
[int] The position to which to move the switch. This changes the output signal to the [input[2*index], input[:2*index + 1]]

`move_switch`(*position*)
Move the switch to the specified position.

Parameters

position
[int] The position to which to move the switch. This changes the output signal to the [input[2*position], input[:2*position + 1]]

SwitchStereo Control The following runtime command ids are available for the SwitchStereo Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_SWITCH_STEREO_POSITION The current switch position.	`sizeof(int32_t)`

Delay

`class audio_dsp.stages.Delay`(*max_delay*, *starting_delay*, *units='samples'*, ***kwargs*)

Delay the input signal by a specified amount.

The maximum delay is set at compile time, and the runtime delay can be set between 0 and `max_delay`.

Parameters

max_delay
[float] The maximum delay in specified units. This can only be set at compile time.

starting_delay
[float] The starting delay in specified units.

units
[str, optional] The units of the delay, can be 'samples', 'ms' or 's'. Default is 'samples'.

Attributes

dsp_block
[`audio_dsp.dsp.signal_chain.delay`] The DSP block class; see *Delay* for implementation details.

`set_delay`(*delay*, *units='samples'*)
Set the length of the delay line, will saturate at max_delay.

Parameters

delay
[float] The delay in specified units.

units
[str] The units of the delay, can be 'samples', 'ms' or 's'. Default is 'samples'.

Delay Control The following runtime command ids are available for the Delay Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_DELAY_MAX_DELAY The maximum delay value in samples. This is only configurable at compile time. This command is read only. When sending a write control command, it will be ignored.	`sizeof(uint32_t)`
CMD_DELAY_DELAY The current delay value in samples. To convert a value in other units of time to samples, use *`time_to_samples()`* in `control/signal_chain.h`. Note the minimum delay provided by this stage is 1 sample. Setting the delay to 0 will still yield a 1 sample delay.	`sizeof(uint32_t)`

Crossfader

class `audio_dsp.stages.Crossfader`(*mix=0.5*, ***kwargs*)

The crossfader mixes between two inputs. The mix control sets the respective levels of each input. When the mix is changed, the gain is updated with a slew.

Parameters

mix
[float] The mix of the crossfader between 0 and 1.

Attributes

dsp_block
[`audio_dsp.dsp.signal_chain.crossfader`] The DSP block class; see *Crossfader* for implementation details.

`set_mix`(*mix*)

Set the mix of the crossfader.

When the mix is set to 0, only the first signal will be output. When the mix is set to 0.5, each channel has a gain of -4.5 dB. When the mix is set to 1, only they second signal will be output.

Parameters

mix
[float] The mix of the crossfader between 0 and 1.

Crossfader Control The following runtime command ids are available for the Crossfader Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_CROSSFADER_GAINS The gains of each input to the crossfader in Q0.31 format. A mix between 0 and 1 can be converted to gains by using the helper function :c:func‘adsp_crossfader_mix‘ in `control/signal_chain.h`.	`sizeof(int32_t)*[2]`

CrossfaderStereo

class audio_dsp.stages.**CrossfaderStereo**(*mix=0.5*, ***kwargs*)

The stereo crossfader mixes between two stereo inputs. The mix control sets the respective levels of each input pair. When the mix is changed, the gain is updated with a slew. The inputs should be passed in pairs, e.g. `[0_L, 0_R, 1_L, 1_R]`.

Attributes

dsp_block
[`audio_dsp.dsp.signal_chain.crossfader`] The DSP block class; see *Crossfader* for implementation details.

set_mix(*mix*)

Set the mix of the crossfader.

When the mix is set to 0, only the first signal will be output. When the mix is set to 0.5, each channel has a gain of -4.5 dB. When the mix is set to 1, only they second signal will be output.

Parameters

mix
[float] The mix of the crossfader between 0 and 1.

CrossfaderStereo Control The following runtime command ids are available for the CrossfaderStereo Stage. For details on reading and writing these commands, see the Run-Time Control User Guide.

Control parameter	Payload length
CMD_CROSSFADER_STEREO_GAINS The gains of each input to the crossfader in Q0.31 format. A mix between 0 and 1 can be converted to gains by using the helper function :c:func‘adsp_crossfader_mix‘ in `control/signal_chain.h`.	`sizeof(int32_t)*[2]`

5.2 DSP Modules

In lib_audio_dsp, DSP modules are the lower level functions and APIs. These can be used directly without the pipeline building tool. The documentation also includes more implementation details about the DSP algorithms. It includes topics such as C and Python APIs, providing more detailed view of the DSP modules.

Each DSP module has been implemented in floating point Python, fixed point int32 Python and fixed point int32 C, with optimisations for xcore. The Python and C fixed point implementations aim to be bit exact with each other, allowing for Python prototyping of DSP pipelines.

Biquad Filters

Single Biquad A second order biquadratic filter, which can be used to implement many common second order filters. The filter had been implemented in the direct form 1, and uses the xcore.ai vector unit to calculate the 5 filter taps in a single instruction.

Coefficients are stored in Q1.30 format to benefit from the vector unit, allowing for a filter coefficient range of `[-2, 1.999]`. For some high gain biquads (e.g. high shelf filters), the numerator coefficients may exceed this range. If this is the case, the numerator coefficients only should be right-shifted until they fit within the range (the denominator coefficients cannot become larger than 2.0 without the poles exceeding the unit circle). The shift should be passed into the API, and the output signal from the biquad will then have a left-shift applied. This is equivalent to reducing the overall signal level in the biquad, then returning to unity gain afterwards.

The `state` should be initialised to `0`. The `state` and `coeffs` must be word-aligned.

C API

int32_t **adsp_biquad**(int32_t new_sample, q2_30 coeffs[5], int32_t state[8], left_shift_t lsh)

Biquad filter. This function implements a biquad filter. The filter is implemented as a direct form 1. The `coeffs` parameter should contain b0/a0, b1/a0, b2/a0, -a1/a0, and -a2/a0 in that order, all represented by fixed-point values shifted left by `30-lsh` bits.

Note: No saturation applied. If output exceeds INT32_MAX, it will overflow.

Parameters

- **new_sample** – New sample to be filtered
- **coeffs** – Filter coefficients
- **state** – Filter state. Must be double-word aligned
- **lsh** – Left shift compensation value, must be positive

Returns

int32_t Filtered sample

Python API

class audio_dsp.dsp.biquad.**biquad**(*coeffs: list[float], fs: int, n_chans: int = 1, Q_sig: int = 27*)

A second order biquadratic filter instance.

This implements a direct form 1 biquad filter, using the coefficients provided at initialisation: *a0*y[n] = b0*x[n] + b1*x[n-1] + b2*x[n-2] - a1*y[n-1] - a2*y[n-2]*

For efficiency the biquad coefficients are normalised by a0 and the output *a* coefficients multiplied by -1.

When the coefficients are updated, the biquad states are reset. This helps avoid large errors, but can make this implementation unsuitable for real time control. For real time control, `biquad_slew` may be a better choice.

Parameters

coeffs
: [list[float]] List of normalised biquad coefficients in the form in the form *[b0, b1, b2, -a1, -a2]/a0*

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] Number of channels the block runs on.

Q_sig: int, optional
: Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] Number of channels the block runs on.

Q_sig: int
: Q format of the signal, number of bits after the decimal point.

coeffs
: [list[float]] List of normalised float biquad coefficients in the form in the form *[b0, b1, b2, -a1, -a2]/a0*, rounded to int32 precision.

int_coeffs
: [list[int]] List of normalised int biquad coefficients in the form in the form *[b0, b1, b2, -a1, -a2]/a0*, scaled and rounded to int32.

b_shift
: [int] The number of right shift bits applied to the b coefficients. The default coefficient scaling allows for a maximum coefficient value of 2, but high gain shelf and peaking filters can have coefficients above this value. Shifting the b coefficients down allows coefficients greater than 2, with the cost of b_shift bits of precision.

`process`(*sample: float*, *channel: int = 0*) → float

Filter a single sample using direct form 1 biquad using floating point maths.

Parameters

sample
: [float] The input sample to be processed.

channel
: [int, optional] The channel index to process the sample on. Default is 0.

Returns

float
: The processed sample.

update_coeffs(*new_coeffs: list[float]*)

Update the saved coefficients to the input values.

Parameters

new_coeffs

[list[float]] The new coefficients to be updated.

reset_state()

Reset the biquad saved states to zero.

Single Slewing Biquad This is similar to *Biquad*, but when the target coefficients are updated it slew the applied coefficients towards the new values. This can be used for real time adjustable filter control. If the left-shift of the coefficients changes, this is managed by the **biquad_slew_t** object:

struct **biquad_slew_t**

Slewing biquad state structure.

Public Members

q2_30 **target_coeffs**[8]

Target filter coefficients, the active coefficients are slewed towards these

q2_30 **active_coeffs**[8]

Active filter coefficients, used to filter the audio

left_shift_t **lsh**

Left shift compensation for if the filter coefficents are large and cannot be represented in Q1.30, must be positive

int32_t **slew_shift**

Shift value used by the exponential slew

left_shift_t **remaining_shifts**

Remaining shifts for cases when the left shift changes during a target_coeff update.

C API

Filtering the samples can be carried out using **adsp_biquad**:

```
for (int j=0; i < n_samples; i++){
    adsp_biquad_slew_coeffs(&slew_state, states, 1);
    for (int i=0; i < n_chans; i++){
      samp_out[i, j] = adsp_biquad(samp_in[i, j], slew_state.active_coeffs, states[i], slew_state.lsh);
    }
}
```

void **adsp_biquad_slew_coeffs**(*biquad_slew_t* *slew_state, int32_t **states, int32_t channels)

Slew the active filter coefficients towards the target filter coefficients. This function should be called either once per sample or per frame, and before calling **adsp_biquad** to do the filtering.

Parameters

- **slew_state** – Slewing biquad state object
- **states** – Filter state for each biquad channel
- **channels** – Number of channels in states

See also *adsp_biquad_slew_init()* and *adsp_biquad_slew_update_coeffs()*.

Python API

class audio_dsp.dsp.biquad.**biquad_slew**(*coeffs: list[float], fs: int, n_chans: int = 1, slew_shift: int = 6, Q_sig: int = 27*)

A second order biquadratic filter instance that slews between coefficient updates.

This implements a direct form 1 biquad filter, using the coefficients provided at initialisation: *a0*y[n] = b0*x[n] + b1*x[n-1] + b2*x[n-2] - a1*y[n-1] - a2*y[n-2]*

For efficiency the biquad coefficients are normalised by a0 and the output *a* coefficients multiplied by -1.

When the target coefficients are updated, the applied coefficients are slewed towards the new target values. This makes this implementation suitable for real time control. A table of the first 10 slew shifts is shown below:

slew_shift	Time constant (ms)
1	0.03
2	0.07
3	0.16
4	0.32
5	0.66
6	1.32
7	2.66
8	5.32
9	10.66
10	21.32

Parameters

coeffs
: [list[float]] List of normalised biquad coefficients in the form in the form *[b0, b1, b2, -a1, -a2]/a0*

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] Number of channels the block runs on.

slew_shift
: [int] The shift value used in the exponential slew.

Q_sig: int, optional
: Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
: [int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

Q_sig: int
Q format of the signal, number of bits after the decimal point.

coeffs
[list[float]] List of normalised float biquad coefficients in the form in the form *[b0, b1, b2, -a1, -a2]/a0*, rounded to int32 precision.

int_coeffs
[list[int]] List of normalised int biquad coefficients in the form in the form *[b0, b1, b2, -a1, -a2]/a0*, scaled and rounded to int32.

b_shift
[int] The number of right shift bits applied to the b coefficients. The default coefficient scaling allows for a maximum coefficient value of 2, but high gain shelf and peaking filters can have coefficients above this value. Shifting the b coefficients down allows coefficients greater than 2, with the cost of b_shift bits of precision.

target_coeffs
[list[float]] List of normalised float target biquad coefficients in the form in the form *[b0, b1, b2, -a1, -a2]/a0*, rounded to int32 precision. The coeffs are slewed towards these values.

target_coeffs_int
[list[int]] List of normalised int target biquad coefficients in the form in the form *[b0, b1, b2, -a1, -a2]/a0*, scaled and rounded to int32. The int_coeffs are slewed towards these values.

`process`(*sample: float*, *channel: int = 0*) → float

`process` is not implemented for the slewing biquad, as the coefficient slew is shared across the channels.

Parameters

sample
[float] The input sample to be processed.

channel
[int, optional] The channel index to process the sample on. Default is 0.

Returns

float
The processed sample.

`update_coeffs`(*new_coeffs: list[float]*)

Update the saved coefficients to the input values.

Parameters

new_coeffs
[list[float]] The new coefficients to be updated.

`reset_state`()

Reset the biquad saved states to zero.

Cascaded Biquads The cascaded biquad module is equivalent to 8 individual biquad filters connected in series. It can be used to implement a simple parametric equaliser or high-order Butterworth filters, implemented as cascaded second order sections.

C API

int32_t **adsp_cascaded_biquads_8b**(int32_t new_sample, q2_30 coeffs[40], int32_t state[64], left_shift_t lsh[8])

8-band cascaded biquad filter This function implements an 8-band cascaded biquad filter. The filter is implemented as a direct form 1 filter.

Note: The filter coefficients must be in [8][5]

Parameters

- **new_sample** – New sample to be filtered
- **coeffs** – Filter coefficients
- **state** – Filter state
- **lsh** – Left shift compensation value

Returns

int32_t Filtered sample

Python API

class audio_dsp.dsp.cascaded_biquads.**cascaded_biquads_8**(*coeffs_list*, *fs*, *n_chans*, *Q_sig=27*)

A class representing a cascaded biquad filter with up to 8 biquads.

This can be used to either implement a parametric equaliser or a higher order filter built out of second order sections.

8 biquad objects are always created, if there are less than 8 biquads in the cascade, the remaining biquads are set to bypass (b0 = 1).

For documentation on individual biquads, see **audio_dsp.dsp.biquad.biquad**.

Parameters

coeffs_list
[list] List of coefficients for each biquad in the cascade.

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

Q_sig: int, optional
Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

Q_sig: int
Q format of the signal, number of bits after the decimal point.

biquads
[list] List of biquad objects representing each biquad in the cascade.

`process`(*sample*, *channel=0*)
Process the input sample through the cascaded biquads using floating point maths.

Parameters

sample
[float] The input sample to be processed.

channel
[int] The channel index to process the sample on.

Returns

float
The processed output sample.

`reset_state`()
Reset the biquad saved states to zero.

Cascaded Biquads 16 This extends the CascadedBiquads class to have 16 cascaded filters. However, The 8 filter C implementation should still be used.

See *Cascaded Biquads*.

Python API

`class audio_dsp.dsp.cascaded_biquads.cascaded_biquads_16`(*coeffs_list*, *fs*, *n_chans*, *Q_sig=27*)

A class representing a cascaded biquad filter with up to 16 biquads.

This can be used to either implement a parametric equaliser or a higher order filter built out of second order sections.

16 biquad objects are always created, if there are less than 16 biquads in the cascade, the remaining biquads are set to bypass (b0 = 1).

For documentation on individual biquads, see `audio_dsp.dsp.biquad.biquad`.

Parameters

coeffs_list
[list] List of coefficients for each biquad in the cascade.

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

Q_sig: int, optional
Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

Q_sig: int
Q format of the signal, number of bits after the decimal point.

biquads
[list] List of biquad objects representing each biquad in the cascade.

`process`(*sample*, *channel=0*)

Process the input sample through the cascaded biquads using floating point maths.

Parameters

sample
[float] The input sample to be processed.

channel
[int] The channel index to process the sample on.

Returns

float
The processed output sample.

`reset_state`()

Reset the biquad saved states to zero.

Dynamic Range Control

Dynamic Range Control (DRC) in audio digital signal processing (DSP) refers to the automatic adjustment of an audio signal's amplitude to reduce its dynamic range - the difference between the loudest and quietest parts of the audio. They include compressors, limiters and clippers, as well as the envelope detectors used to detect the signal level.

Attack and Release Times Nearly all DRC modules feature an attack and release time to control the responsiveness of the module to changes in signal level. Attack and release times converted from seconds to alpha coefficients for use in the the exponential moving average calculation. The shorter the attack or release time, the bigger the alpha. Large alpha will result in the envelope becoming more reactive to the input samples. Small alpha values will give more smoothed behaviour. The difference between the input level and the current envelope or gain determines whether the attack or release alpha is used.

Envelope Detectors Envelope detectors run an exponential moving average (EMA) of the incoming signal. They are used as a part of the most DRC components. They can also be used to implement VU meters and level detectors.

They feature *attack and release times* to control the responsiveness of the envelope detector.

The C struct below is used for all the envelope detector implementations.

struct `env_detector_t`

Envelope detector state structure.

Public Members

q1_31 **attack_alpha**

Attack alpha

q1_31 **release_alpha**

Release alpha

int32_t **envelope**

Current envelope

Peak Envelope Detector A peak-based envelope detector will run its EMA using the absolute value of the input sample.

C API

void **adsp_env_detector_peak**(*env_detector_t* *env_det, int32_t new_sample)

Update the envelope detector peak with a new sample.

Parameters

- **env_det** – Envelope detector object
- **new_sample** – New sample

Python API

class audio_dsp.dsp.drc.**envelope_detector_peak**(*fs*, *n_chans*, *attack_t*, *release_t*, *Q_sig=27*)

Envelope detector that follows the absolute peak value of a signal.

The attack time sets how fast the envelope detector ramps up. The release time sets how fast the envelope detector ramps down.

Parameters

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

attack_t

[float] Attack time of the envelope detector in seconds. This cannot be faster than 2/fs seconds, and saturates to that value. Exceptionally large attack times may converge to zero.

release_t: float

Release time of the envelope detector in seconds. This cannot be faster than 2/fs seconds, and saturates to that value. Exceptionally large release times may converge to zero.

Q_sig: int, optional

Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs

[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

Q_sig: int
Q format of the signal, number of bits after the decimal point.

attack_t
[float] The attack time of the compressor/limiter in seconds; changing this property also sets the EWM alpha values for fixed and floating point implementations.

release_t
[float] The release time of the compressor/limiter in seconds; changing this property also sets the EWM alpha values for fixed and floating point implementations.

attack_alpha
[float] Attack time parameter used for exponential moving average in floating point processing.

release_alpha
[float] Release time parameter used for exponential moving average in floating point processing.

envelope
[list[float]] Current envelope value for each channel for floating point processing.

attack_alpha_int
[int] attack_alpha in 32-bit int format.

release_alpha_int
[int] release_alpha in 32-bit int format.

envelope_int
[list[int]] current envelope value for each channel in 32-bit int format.

`process`(*sample*, *channel=0*)

Update the peak envelope for a signal, using floating point maths.

Take one new sample and return the updated envelope. Input should be scaled with 0 dB = 1.0.

Parameters

sample
[float] The input sample to be processed.

channel
[int, optional] The channel index to process the sample on. Default is 0.

Returns

float
The processed sample.

`reset_state`()

Reset the envelope to zero.

RMS Envelope Detector An RMS-based envelope detector will run its EMA using the square of the input sample. It returns the mean² in order to avoid a square root.

C API

void **adsp_env_detector_rms**(*env_detector_t* *env_det, int32_t new_sample)

Update the envelope detector RMS with a new sample.

Parameters

- **env_det** – Envelope detector object
- **new_sample** – New sample

Python API

class audio_dsp.dsp.drc.**envelope_detector_rms**(*fs*, *n_chans*, *attack_t*, *release_t*, *Q_sig=27*)

Envelope detector that follows the RMS value of a signal.

Note this returns the mean² value, there is no need to do the sqrt() as if the output is converted to dB, 10log10() can be taken instead of 20log10().

The attack time sets how fast the envelope detector ramps up. The release time sets how fast the envelope detector ramps down.

Parameters

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] Number of channels the block runs on.

attack_t
: [float] Attack time of the envelope detector in seconds. This cannot be faster than 2/fs seconds, and saturates to that value. Exceptionally large attack times may converge to zero.

release_t: float
: Release time of the envelope detector in seconds. This cannot be faster than 2/fs seconds, and saturates to that value. Exceptionally large release times may converge to zero.

Q_sig: int, optional
: Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] Number of channels the block runs on.

Q_sig: int
: Q format of the signal, number of bits after the decimal point.

attack_t
: [float] The attack time of the compressor/limiter in seconds; changing this property also sets the EWM alpha values for fixed and floating point implementations.

release_t
: [float] The release time of the compressor/limiter in seconds; changing this property also sets the EWM alpha values for fixed and floating point implementations.

attack_alpha
: [float] Attack time parameter used for exponential moving average in floating point processing.

release_alpha
[float] Release time parameter used for exponential moving average in floating point processing.

envelope
[list[float]] Current envelope value for each channel for floating point processing.

attack_alpha_int
[int] attack_alpha in 32-bit int format.

release_alpha_int
[int] release_alpha in 32-bit int format.

envelope_int
[list[int]] current envelope value for each channel in 32-bit int format.

`process`(*sample*, *channel=0*)

Update the RMS envelope for a signal, using floating point maths.

Take one new sample and return the updated envelope. Input should be scaled with 0 dB = 1.0.

Note this returns the mean² value, there is no need to do the sqrt() as if the output is converted to dB, 10log10() can be taken instead of 20log10().

Parameters

sample
[float] The input sample to be processed.

channel
[int, optional] The channel index to process the sample on. Default is 0.

Returns

float
The processed sample.

`reset_state`()

Reset the envelope to zero.

Clipper A clipper limits the signal to a specified threshold. It is applied instantaneously, so has no attack or release times.

typedef int32_t `clipper_t`

Clipper state structure. Should be initilised with the linear threshold.

C API

int32_t `adsp_clipper`(*clipper_t* clip, int32_t new_samp)

Process a new sample with a clipper.

Parameters

- `clip` – Clipper object
- `new_samp` – New sample

Returns
int32_t Clipped sample

Python API

class audio_dsp.dsp.drc.**clipper**(*fs*, *n_chans*, *threshold_db*, *Q_sig=27*)

A simple clipper that limits the signal to a specified threshold.

Parameters

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

threshold_db
[float] Threshold above which clipping occurs in dB.

Q_sig: int, optional
Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

Q_sig: int
Q format of the signal, number of bits after the decimal point.

threshold_db
[float] The threshold in decibels; changing this property also updates the fixed and floating point thresholds in linear gain.

threshold
[float] Value above which clipping occurs for floating point processing.

threshold_int
[int] Value above which clipping occurs for int32 fixed point processing.

process(*sample*, *channel=0*)

Take one new sample and return the clipped sample, using floating point maths. Input should be scaled with 0 dB = 1.0.

Parameters

sample
[float] The input sample to be processed.

channel
[int, optional] The channel index to process the sample on. Default is 0.

Returns

float
The processed sample.

Limiters Limiters will reduce the amplitude of a signal when the signal envelope is greater than the desired threshold. This is similar behaviour to a compressor with an infinite ratio.

A limiter will run an internal envelope detector to get the signal envelope, then compare it to the threshold. If the envelope is greater than the threshold, the applied gain will be reduced. If the envelope is below the threshold, unity gain will be applied. The gain is run through an EMA to avoid abrupt changes. The same *attack and release times* are used for the envelope detector and the gain smoothing.

The C struct below is used for all the limiter implementations.

struct **limiter_t**

Limiter state structure.

Public Members

env_detector_t **env_det**

Envelope detector

int32_t **threshold**

Linear threshold

int32_t **gain**

Linear gain

Peak Limiter A peak limiter uses the *Peak Envelope Detector* to get an envelope. When envelope is above the threshold, the new gain is calculated as `threshold / envelope`.

C API

int32_t **adsp_limiter_peak**(*limiter_t* *lim, int32_t new_samp)

Process a new sample with a peak limiter.

Parameters

- **lim** – Limiter object
- **new_samp** – New sample

Returns

int32_t Limited sample

Python API

class audio_dsp.dsp.drc.**limiter_peak**(*fs*, *n_chans*, *threshold_db*, *attack_t*, *release_t*, *Q_sig=27*)

A limiter based on the peak value of the signal. When the peak envelope of the signal exceeds the threshold, the signal amplitude is reduced.

The threshold set the value above which limiting occurs. The attack time sets how fast the limiter starts limiting. The release time sets how long the signal takes to ramp up to its original level after the envelope is below the threshold.

Parameters

fs

[int] Sampling frequency in Hz.

n_chans
: [int] Number of parallel channels the compressor/limiter runs on. The channels are limited/compressed separately, only the constant parameters are shared.

threshold_db
: [float] Threshold in decibels above which limiting occurs.

attack_t
: [float] Attack time of the compressor/limiter in seconds. This cannot be faster than 2/fs seconds, and saturates to that value. Exceptionally large attack times may converge to zero.

release_t: float
: Release time of the compressor/limiter in seconds. This cannot be faster than 2/fs seconds, and saturates to that value. Exceptionally large release times may converge to zero.

Q_sig: int, optional
: Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] Number of channels the block runs on.

Q_sig: int
: Q format of the signal, number of bits after the decimal point.

`attack_t`
: [float] The attack time of the compressor/limiter in seconds; changing this property also sets the EWM alpha values for fixed and floating point implementations.

`release_t`
: [float] The release time of the compressor/limiter in seconds; changing this property also sets the EWM alpha values for fixed and floating point implementations.

threshold
: [float] Value above which compression/limiting occurs for floating point processing.

gain
: [list[float]] Current gain to be applied to the signal for each channel for floating point processing.

attack_alpha
: [float] Attack time parameter used for exponential moving average in floating point processing.

release_alpha
: [float] Release time parameter used for exponential moving average in floating point processing.

threshold_int
: [int] Value above which compression/limiting occurs for int32 fixed point processing.

gain_int
: [list[int]] Current gain to be applied to the signal for each channel for int32 fixed point processing.

attack_alpha_int
: [int] attack_alpha in 32-bit int format.

release_alpha_int
[int] release_alpha in 32-bit int format.

gain_calc
[function] function pointer to floating point gain calculation function.

gain_calc_int
[function] function pointer to fixed point gain calculation function.

threshold_db
[float] The threshold in decibels; changing this property also updates the fixed and floating point thresholds in linear gain.

env_detector
[envelope_detector_peak] Peak envelope detector object used to calculate the envelope of the signal.

process(*sample*, *channel=0*)

Update the envelope for a signal, then calculate and apply the required gain for compression/limiting, using floating point maths.

Take one new sample and return the compressed/limited sample. Input should be scaled with 0 dB = 1.0.

Parameters

sample
[float] The input sample to be processed.

channel
[int, optional] The channel index to process the sample on. Default is 0.

Returns

float
The processed sample.

reset_state()

Reset the envelope detector to 0 and the gain to 1.

Hard Peak Limiter A hard peak limiter is similar to a *Peak Limiter*, but will clip the output if it's still above the threshold after the peak limiter. This can be useful for a final output limiter before truncating any headroom bits.

C API

int32_t **adsp_hard_limiter_peak**(*limiter_t* *lim, int32_t new_samp)

Process a new sample with a hard limiter peak.

Parameters

- **lim** – Limiter object
- **new_samp** – New sample

Returns
int32_t Limited sample

Python API

class audio_dsp.dsp.drc.**hard_limiter_peak**(*fs*, *n_chans*, *threshold_db*, *attack_t*, *release_t*, *Q_sig=27*)

A limiter based on the peak value of the signal. When the peak envelope of the signal exceeds the threshold, the signal amplitude is reduced. If the signal still exceeds the threshold, it is clipped.

The threshold set the value above which limiting/clipping occurs. The attack time sets how fast the limiter starts limiting. The release time sets how long the signal takes to ramp up to it's original level after the envelope is below the threshold.

Parameters

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] Number of parallel channels the compressor/limiter runs on. The channels are limited/compressed separately, only the constant parameters are shared.

threshold_db
: [float] Threshold in decibels above which limiting occurs.

attack_t
: [float] Attack time of the compressor/limiter in seconds. This cannot be faster than 2/fs seconds, and saturates to that value. Exceptionally large attack times may converge to zero.

release_t: float
: Release time of the compressor/limiter in seconds. This cannot be faster than 2/fs seconds, and saturates to that value. Exceptionally large release times may converge to zero.

Q_sig: int, optional
: Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] Number of channels the block runs on.

Q_sig: int
: Q format of the signal, number of bits after the decimal point.

attack_t
: [float] The attack time of the compressor/limiter in seconds; changing this property also sets the EWM alpha values for fixed and floating point implementations.

release_t
: [float] The release time of the compressor/limiter in seconds; changing this property also sets the EWM alpha values for fixed and floating point implementations.

threshold
: [float] Value above which compression/limiting occurs for floating point processing.

gain
: [list[float]] Current gain to be applied to the signal for each channel for floating point processing.

attack_alpha
: [float] Attack time parameter used for exponential moving average in floating point processing.

release_alpha
: [float] Release time parameter used for exponential moving average in floating point processing.

threshold_int
: [int] Value above which compression/limiting occurs for int32 fixed point processing.

gain_int
: [list[int]] Current gain to be applied to the signal for each channel for int32 fixed point processing.

attack_alpha_int
: [int] attack_alpha in 32-bit int format.

release_alpha_int
: [int] release_alpha in 32-bit int format.

gain_calc
: [function] function pointer to floating point gain calculation function.

gain_calc_int
: [function] function pointer to fixed point gain calculation function.

threshold_db
: [float] The threshold in decibels; changing this property also updates the fixed and floating point thresholds in linear gain.

env_detector
: [envelope_detector_peak] Peak envelope detector object used to calculate the envelope of the signal.

`process`(*sample*, *channel=0*)

Update the envelope for a signal, then calculate and apply the required gain for limiting, using floating point maths. If the output signal exceeds the threshold, clip it to the threshold.

Take one new sample and return the limited sample. Input should be scaled with 0 dB = 1.0.

Parameters

sample
: [float] The input sample to be processed.

channel
: [int, optional] The channel index to process the sample on. Default is 0.

Returns

float
: The processed sample.

`reset_state`()

Reset the envelope detector to 0 and the gain to 1.

RMS Limiter A RMS limiter uses the *RMS Envelope Detector* to calculate an envelope. When envelope is above the threshold, the new gain is calculated as `sqrt(threshold / envelope)`.

C API

int32_t **adsp_limiter_rms**(*limiter_t* *lim, int32_t new_samp)

Process a new sample with an RMS limiter.

Parameters

- **lim** – Limiter object
- **new_samp** – New sample

Returns

int32_t Limited sample

Python API

class audio_dsp.dsp.drc.**limiter_rms**(*fs*, *n_chans*, *threshold_db*, *attack_t*, *release_t*, *Q_sig=27*)

A limiter based on the RMS value of the signal. When the RMS envelope of the signal exceeds the threshold, the signal amplitude is reduced.

The threshold set the value above which limiting occurs. The attack time sets how fast the limiter starts limiting. The release time sets how long the signal takes to ramp up to its original level after the envelope is below the threshold.

Parameters

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] Number of parallel channels the compressor/limiter runs on. The channels are limited/compressed separately, only the constant parameters are shared.

threshold_db
: [float] Threshold in decibels above which limiting occurs.

attack_t
: [float] Attack time of the compressor/limiter in seconds. This cannot be faster than 2/fs seconds, and saturates to that value. Exceptionally large attack times may converge to zero.

release_t: float
: Release time of the compressor/limiter in seconds. This cannot be faster than 2/fs seconds, and saturates to that value. Exceptionally large release times may converge to zero.

Q_sig: int, optional
: Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] Number of channels the block runs on.

Q_sig: int
: Q format of the signal, number of bits after the decimal point.

attack_t
: [float] The attack time of the compressor/limiter in seconds; changing this property also sets the EWM alpha values for fixed and floating point implementations.

release_t
: [float] The release time of the compressor/limiter in seconds; changing this property also sets the EWM alpha values for fixed and floating point implementations.

threshold
: [float] Value above which compression/limiting occurs for floating point processing. Note the threshold is saved in the power domain, as the RMS envelope detector returns x^2.

gain
: [list[float]] Current gain to be applied to the signal for each channel for floating point processing.

attack_alpha
: [float] Attack time parameter used for exponential moving average in floating point processing.

release_alpha
: [float] Release time parameter used for exponential moving average in floating point processing.

threshold_int
: [int] Value above which compression/limiting occurs for int32 fixed point processing.

gain_int
: [list[int]] Current gain to be applied to the signal for each channel for int32 fixed point processing.

attack_alpha_int
: [int] attack_alpha in 32-bit int format.

release_alpha_int
: [int] release_alpha in 32-bit int format.

gain_calc
: [function] function pointer to floating point gain calculation function.

gain_calc_int
: [function] function pointer to fixed point gain calculation function.

env_detector
: [envelope_detector_rms] RMS envelope detector object used to calculate the envelope of the signal.

process(*sample*, *channel=0*)

Update the envelope for a signal, then calculate and apply the required gain for compression/limiting, using floating point maths.

Take one new sample and return the compressed/limited sample. Input should be scaled with 0 dB = 1.0.

Parameters

sample
: [float] The input sample to be processed.

channel
: [int, optional] The channel index to process the sample on. Default is 0.

Returns

float
: The processed sample.

reset_state()

Reset the envelope detector to 0 and the gain to 1.

Compressors A compressor will attenuate the signal when the envelope is greater than the threshold. The input/output relationship above the threshold is defined by the compressor `ratio`.

As with a limiter, the compressor runs an internal envelope detector to get the signal envelope, then compares it to the threshold. If the envelope is greater than the threshold, the gain will be proportionally reduced by the `ratio`, such that it is greater than the threshold by a smaller amount. If the envelope is below the threshold, unity gain will be applied. The gain is then run through an EMA to avoid abrupt changes, before being applied.

The `ratio` defines the input/output gradient in the logarithmic domain. For example, a ratio of 2 will reduce the output gain by 0.5 dB for every 1 dB the envelope is over the threshold. A ratio of 1 will apply no compression. To avoid converting the envelope to the logarithmic domain for the gain calculation, the ratio is converted to the `slope` as `(1 - 1 / ratio) / 2`. The gain can then be calculated as an exponential in the linear domain.

The C structs below are used for all the compressors implementations.

struct **compressor_t**

Compressor state structure.

Public Members

env_detector_t **env_det**

Envelope detector

int32_t **threshold**

Linear threshold

int32_t **gain**

Linear gain

float **slope**

Slope of the compression curve

struct **compressor_stereo_t**

Stereo compressor state structure.

Public Members

env_detector_t **env_det_l**

Envelope detector for left channel

env_detector_t **env_det_r**

Envelope detector for right channel

int32_t **threshold**

Linear threshold

int32_t **gain**

Linear gain

float **slope**

Slope of the compression curve

RMS Compressor The RMS compressor uses the *RMS Envelope Detector* to calculate an envelope. When the envelope is above the threshold, the new gain is calculated as `(threshold / envelope) ^ slope`.

C API

int32_t **adsp_compressor_rms**(*compressor_t* *comp, int32_t new_samp)

Process a new sample with an RMS compressor.

Parameters

- **comp** – Compressor object
- **new_samp** – New sample

Returns

int32_t Compressed sample

Python API

class audio_dsp.dsp.drc.**compressor_rms**(*fs*, *n_chans*, *ratio*, *threshold_db*, *attack_t*, *release_t*, *Q_sig=27*)

A compressor based on the RMS value of the signal. When the RMS envelope of the signal exceeds the threshold, the signal amplitude is reduced by the compression ratio.

The threshold sets the value above which compression occurs. The ratio sets how much the signal is compressed. A ratio of 1 results in no compression, while a ratio of infinity results in the same behaviour as a limiter. The attack time sets how fast the compressor starts compressing. The release time sets how long the signal takes to ramp up to it's original level after the envelope is below the threshold.

Parameters

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of parallel channels the compressor/limiter runs on. The channels are limited/compressed separately, only the constant parameters are shared.

ratio

[float] Compression gain ratio applied when the signal is above the threshold

threshold_db

[float] Threshold in decibels above which limiting occurs.

attack_t
: [float] Attack time of the compressor/limiter in seconds. This cannot be faster than 2/fs seconds, and saturates to that value. Exceptionally large attack times may converge to zero.

release_t: float
: Release time of the compressor/limiter in seconds. This cannot be faster than 2/fs seconds, and saturates to that value. Exceptionally large release times may converge to zero.

Q_sig: int, optional
: Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] Number of channels the block runs on.

Q_sig: int
: Q format of the signal, number of bits after the decimal point.

`attack_t`
: [float] The attack time of the compressor/limiter in seconds; changing this property also sets the EWM alpha values for fixed and floating point implementations.

`release_t`
: [float] The release time of the compressor/limiter in seconds; changing this property also sets the EWM alpha values for fixed and floating point implementations.

threshold
: [float] Value above which compression/limiting occurs for floating point processing. Note the threshold is saved in the power domain, as the RMS envelope detector returns x^2.

gain
: [list[float]] Current gain to be applied to the signal for each channel for floating point processing.

attack_alpha
: [float] Attack time parameter used for exponential moving average in floating point processing.

release_alpha
: [float] Release time parameter used for exponential moving average in floating point processing.

threshold_int
: [int] Value above which compression/limiting occurs for int32 fixed point processing.

gain_int
: [list[int]] Current gain to be applied to the signal for each channel for int32 fixed point processing.

attack_alpha_int
: [int] attack_alpha in 32-bit int format.

release_alpha_int
: [int] release_alpha in 32-bit int format.

gain_calc
: [function] function pointer to floating point gain calculation function.

gain_calc_int
[function] function pointer to fixed point gain calculation function.

env_detector
[envelope_detector_rms] RMS envelope detector object used to calculate the envelope of the signal.

ratio
[float] Compression gain ratio applied when the signal is above the threshold; changing this property also updates the slope used in the fixed and floating point implementation.

slope
[float] The slope factor of the compressor, defined as *slope = (1 - 1/ratio) / 2*.

slope_f32
[float32] The slope factor of the compressor, used for int32 to float32 processing.

process(*sample*, *channel=0*)

Update the envelope for a signal, then calculate and apply the required gain for compression/limiting, using floating point maths.
Take one new sample and return the compressed/limited sample. Input should be scaled with 0 dB = 1.0.

Parameters

sample
[float] The input sample to be processed.

channel
[int, optional] The channel index to process the sample on. Default is 0.

Returns

float
The processed sample.

reset_state()

Reset the envelope detector to 0 and the gain to 1.

Sidechain RMS Compressor The sidechain RMS compressor calculates the envelope of one signal and uses it to compress another signal. It takes two signals: *detect* and *input*. The envelope of the *detect* signal is calculated using an internal *RMS Envelope Detector*. The gain is calculated in the same way as a *RMS Compressor*, but the gain is then applied to the *input* sample. This can be used to reduce the level of the *input* signal when the *detect* signal gets above the threshold.

C API

int32_t **adsp_compressor_rms_sidechain**(*compressor_t* *comp, int32_t input_samp, int32_t detect_samp)

Process a new sample with a sidechain RMS compressor.

Parameters

- **comp** – Compressor object
- **input_samp** – Input sample
- **detect_samp** – Sidechain sample

Returns
int32_t Compressed sample

Python API

class audio_dsp.dsp.drc.**compressor_rms_sidechain_mono**(*fs*, *ratio*, *threshold_db*, *attack_t*, *release_t*, *Q_sig=27*)

A mono sidechain compressor based on the RMS value of the signal. When the RMS envelope of the signal exceeds the threshold, the signal amplitude is reduced by the compression ratio.

The threshold sets the value above which compression occurs. The ratio sets how much the signal is compressed. A ratio of 1 results in no compression, while a ratio of infinity results in the same behaviour as a limiter. The attack time sets how fast the compressor starts compressing. The release time sets how long the signal takes to ramp up to it's original level after the envelope is below the threshold.

Parameters

fs
: [int] Sampling frequency in Hz.

ratio
: [float] Compression gain ratio applied when the signal is above the threshold

threshold_db
: [float] Threshold in decibels above which limiting occurs.

attack_t
: [float] Attack time of the compressor/limiter in seconds. This cannot be faster than 2/fs seconds, and saturates to that value. Exceptionally large attack times may converge to zero.

release_t: float
: Release time of the compressor/limiter in seconds. This cannot be faster than 2/fs seconds, and saturates to that value. Exceptionally large release times may converge to zero.

Q_sig: int, optional
: Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] Number of channels the block runs on.

Q_sig: int
: Q format of the signal, number of bits after the decimal point.

attack_t
: [float] The attack time of the compressor/limiter in seconds; changing this property also sets the EWM alpha values for fixed and floating point implementations.

release_t
: [float] The release time of the compressor/limiter in seconds; changing this property also sets the EWM alpha values for fixed and floating point implementations.

threshold
: [float] Value above which compression/limiting occurs for floating point processing. Note the threshold is saved in the power domain, as the RMS envelope detector returns x².

gain
: [list[float]] Current gain to be applied to the signal for each channel for floating point processing.

attack_alpha
: [float] Attack time parameter used for exponential moving average in floating point processing.

release_alpha
: [float] Release time parameter used for exponential moving average in floating point processing.

threshold_int
: [int] Value above which compression/limiting occurs for int32 fixed point processing.

gain_int
: [list[int]] Current gain to be applied to the signal for each channel for int32 fixed point processing.

attack_alpha_int
: [int] attack_alpha in 32-bit int format.

release_alpha_int
: [int] release_alpha in 32-bit int format.

gain_calc
: [function] function pointer to floating point gain calculation function.

gain_calc_int
: [function] function pointer to fixed point gain calculation function.

env_detector
: [envelope_detector_rms] RMS envelope detector object used to calculate the envelope of the signal.

`ratio`
: [float] Compression gain ratio applied when the signal is above the threshold; changing this property also updates the slope used in the fixed and floating point implementation.

slope
: [float] The slope factor of the compressor, defined as *slope = (1 - 1/ratio) / 2*.

slope_f32
: [float32] The slope factor of the compressor, used for int32 to float32 processing.

`process`(*input_sample: float*, *detect_sample: float*)

Update the envelope for the detection signal, then calculate and apply the required gain for compression/limiting, and apply to the input signal using floating point maths.

Take one new sample and return the compressed/limited sample. Input should be scaled with 0 dB = 1.0.

Parameters

input_sample
: [float] The input sample to be compressed.

detect_sample
[float] The sample used by the envelope detector to determine the amount of compression to apply to the input_sample.

Returns

float
The processed sample.

reset_state()
Reset the envelope detectors to 0 and the gain to 1.

Stereo Sidechain RMS Compressor The stereo sidechain RMS compressor expands the *Sidechain RMS Compressor* to take 2 input and 2 detection channels. The envelope of each detection channel is taken, and the maximum of the envelopes is used to calculate the compression to apply to the input channels.

C API

void **adsp_compressor_rms_sidechain_stereo**(*compressor_stereo_t* *comp, int32_t outputs_lr[2], int32_t input_samp_l, int32_t input_samp_r, int32_t detect_samp_l, int32_t detect_samp_r)

Process a pair of new samples with a stereo sidechain RMS compressor.

Parameters

- **comp** – Compressor object
- **outputs_lr** – Pointer to the outputs 0:left, 1:right
- **input_samp_l** – Left input sample
- **input_samp_r** – Right input sample
- **detect_samp_l** – Left sidechain sample
- **detect_samp_r** – Right sidechain sample

Python API

class audio_dsp.dsp.drc.**compressor_rms_sidechain_stereo**(*fs*, *ratio*, *thresh-old_db*, *attack_t*, *release_t*, *Q_sig=27*)

A stereo sidechain compressor based on the RMS value of the signal. When the RMS envelope of the signal exceeds the threshold, the signal amplitude is reduced by the compression ratio. The same compression is applied to both channels, using the highest individual channel envelope.

The threshold sets the value above which compression occurs. The ratio sets how much the signal is compressed. A ratio of 1 results in no compression, while a ratio of infinity results in the same behaviour as a limiter. The attack time sets how fast the compressor starts compressing. The release time sets how long the signal takes to ramp up to it's original level after the envelope is below the threshold.

Parameters

fs
[int] Sampling frequency in Hz.

ratio
: [float] Compression gain ratio applied when the signal is above the threshold

threshold_db
: [float] Threshold in decibels above which limiting occurs.

attack_t
: [float] Attack time of the compressor/limiter in seconds. This cannot be faster than 2/fs seconds, and saturates to that value. Exceptionally large attack times may converge to zero.

release_t: float
: Release time of the compressor/limiter in seconds. This cannot be faster than 2/fs seconds, and saturates to that value. Exceptionally large release times may converge to zero.

Q_sig: int, optional
: Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] Number of channels the block runs on.

Q_sig: int
: Q format of the signal, number of bits after the decimal point.

`threshold_db`
: [float] The threshold in decibels; changing this property also updates the fixed and floating point thresholds in linear gain.

`attack_t`
: [float] The attack time in seconds.

`release_t`
: [float] The release time in seconds.

threshold
: [float] Value above which compression/limiting occurs for floating point processing. Note the threshold is saved in the power domain, as the RMS envelope detector returns x^2.

gain
: [list[float]] Current gain to be applied to the signal for each channel for floating point processing.

attack_alpha
: [float] Attack time parameter used for exponential moving average in floating point processing.

release_alpha
: [float] Release time parameter used for exponential moving average in floating point processing.

threshold_int
: [int] Value above which compression/limiting occurs for int32 fixed point processing.

gain_int
: [list[int]] Current gain to be applied to the signal for each channel for int32 fixed point processing.

attack_alpha_int
: [int] attack_alpha in 32-bit int format.

release_alpha_int
[int] release_alpha in 32-bit int format.

gain_calc
[function] function pointer to floating point gain calculation function.

gain_calc_int
[function] function pointer to fixed point gain calculation function.

env_detector
[envelope_detector_rms] RMS envelope detector object used to calculate the envelope of the signal.

ratio
[float] Compression gain ratio applied when the signal is above the threshold; changing this property also updates the slope used in the fixed and floating point implementation.

slope
[float] The slope factor of the compressor, defined as *slope = (1 - 1/ratio)*.

slope_f32
[float32] The slope factor of the compressor, used for int32 to float32 processing.

`process`(*sample: float*, *channel=0*)

Take one new sample and give it back. Do no processing for the generic block.

Parameters

sample
[float] The input sample to be processed.

channel
[int, optional] The channel index to process the sample on. Default is 0.

Returns

float
The processed sample.

`reset_state`()

Reset the envelope detectors to 0 and the gain to 1.

Expanders An expander attenuates a signal when the envelope is below the threshold. This increases the dynamic range of the signal, and can be used to attenuate quiet signals, such as low level noise.

Like limiters and compressors, an expander will run an internal envelope detector to calculate the envelope and compare it to the threshold. If the envelope is below the threshold, the applied gain will be reduced. If the envelope is greater than the threshold, unity gain will be applied. The gain is run through an EMA to avoid abrupt changes. The same *attack and release times* are used for the envelope detector and the gain smoothing. In an expander, the attack time is defined as the speed at which the gain returns to unity after the signal has been below the threshold.

Noise Gate A noise gate uses the *Peak Envelope Detector* to calculate the envelope of the input signal. When the envelope is below the threshold, a gain of 0 is applied to the input signal. Otherwise, unity gain is applied.

typedef *limiter_t* **noise_gate_t**

Noise gate state structure.

C API

int32_t **adsp_noise_gate**(*noise_gate_t* *ng, int32_t new_samp)

Process a new sample with a noise gate.

Parameters

- **ng** – Noise gate object
- **new_samp** – New sample

Returns

int32_t Gated sample

Python API

class audio_dsp.dsp.drc.**noise_gate**(*fs*, *n_chans*, *threshold_db*, *attack_t*, *release_t*, *Q_sig=27*)

A noise gate that reduces the level of an audio signal when it falls below a threshold.

When the signal envelope falls below the threshold, the gain applied to the signal is reduced to 0 (based on the release time). When the envelope returns above the threshold, the gain applied to the signal is increased to 1 over the attack time.

The initial state of the noise gate is with the gate open (no attenuation), assuming a full scale signal has been present before t = 0.

Parameters

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] number of parallel channels the expander runs on. The channels are expanded separately, only the constant parameters are shared.

threshold_db
: [float] Threshold in decibels below which expansion occurs. This cannot be greater than the maximum value representable in Q_SIG format, and will saturate to that value.

attack_t
: [float] Attack time of the expander in seconds. This cannot be faster than 2/fs seconds, and saturates to that value. Exceptionally large attack times may converge to zero.

release_t: float
: Release time of the expander in seconds. This cannot be faster than 2/fs seconds, and saturates to that value. Exceptionally large release times may converge to zero.

Q_sig: int, optional
: Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
: [int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

Q_sig: int
Q format of the signal, number of bits after the decimal point.

attack_t
[float] The attack time of the compressor/limiter in seconds; changing this property also sets the EWM alpha values for fixed and floating point implementations.

release_t
[float] The release time of the compressor/limiter in seconds; changing this property also sets the EWM alpha values for fixed and floating point implementations.

threshold
[float] Value below which expanding occurs for floating point processing.

gain
[list[float]] Current gain to be applied to the signal for each channel for floating point processing.

attack_alpha
[float] Attack time parameter used for exponential moving average in floating point processing.

release_alpha
[float] Release time parameter used for exponential moving average in floating point processing.

threshold_int
[int] Value below which expanding occurs for int32 fixed point processing.

gain_int
[list[int]] Current gain to be applied to the signal for each channel for int32 fixed point processing.

attack_alpha_int
[int] attack_alpha in 32-bit int format.

release_alpha_int
[int] release_alpha in 32-bit int format.

gain_calc
[function] function pointer to floating point gain calculation function.

gain_calc_int
[function] function pointer to fixed point gain calculation function.

threshold_db
[float] The threshold in decibels; changing this property also updates the fixed and floating point thresholds in linear gain.

env_detector
[envelope_detector_peak] Peak envelope detector object used to calculate the envelope of the signal.

process(*sample*, *channel=0*)

Update the envelope for a signal, then calculate and apply the required gain for expanding, using floating point maths.

Take one new sample and return the expanded sample. Input should be scaled with 0 dB = 1.0.

Parameters

sample
[float] The input sample to be processed.

channel
[int, optional] The channel index to process the sample on. Default is 0.

Returns

float
The processed sample.

reset_state()
Reset the envelope detector to 1 and the gain to 1, so the gate starts off.

Noise Suppressor/Expander A basic expander can also be used as a noise suppressor. It uses the *Peak Envelope Detector* to calculate the envelope of the input signal. When the envelope is below the threshold, the gain of the signal is reduced according to the ratio. Otherwise, unity gain is applied.

Like a compressor, the `ratio` defines the input/output gradient in the logarithmic domain. For example, a ratio of 2 will reduce the output gain by 0.5 dB for every 1 dB the envelope is below the threshold. A ratio of 1 will apply no gain changes. To avoid converting the envelope to the logarithmic domain for the gain calculation, the ratio is converted to the `slope` as `(1 - ratio)`. The gain can then be calculated as an exponential in the linear domain.

For speed, some parameters such as `inv_threshold` are computed at initialisation to simplify run-time computation.

struct **noise_suppressor_expander_t**

Public Members

env_detector_t **env_det**
Envelope detector

int32_t **threshold**
Linear threshold

int64_t **inv_threshold**
Inverse threshold

int32_t **gain**
Linear gain

float **slope**
Slope of the noise suppression curve

C API

int32_t **adsp_noise_suppressor_expander**(*noise_suppressor_expander_t* *nse, int32_t new_samp)

Process a new sample with a noise suppressor (expander)

Parameters

- **nse** – Noise suppressor (Expander) object
- **new_samp** – New sample

Returns

int32_t Suppressed sample

Python API

class audio_dsp.dsp.drc.**noise_suppressor_expander**(*fs*, *n_chans*, *ratio*, *threshold_db*, *attack_t*, *release_t*, *Q_sig=27*)

A noise suppressor that reduces the level of an audio signal when it falls below a threshold. This is also known as an expander.

When the signal envelope falls below the threshold, the gain applied to the signal is reduced relative to the expansion ratio over the release time. When the envelope returns above the threshold, the gain applied to the signal is increased to 1 over the attack time.

The initial state of the noise suppressor is with the suppression off, assuming a full scale signal has been present before t = 0.

Parameters

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] number of parallel channels the expander runs on. The channels are expanded separately, only the constant parameters are shared.

ratio
: [float] The expansion ratio applied to the signal when the envelope falls below the threshold.

threshold_db
: [float] Threshold in decibels below which expansion occurs. This cannot be greater than the maximum value representable in Q_SIG format, and will saturate to that value.

attack_t
: [float] Attack time of the expander in seconds. This cannot be faster than 2/fs seconds, and saturates to that value. Exceptionally large attack times may converge to zero.

release_t: float
: Release time of the expander in seconds. This cannot be faster than 2/fs seconds, and saturates to that value. Exceptionally large release times may converge to zero.

Q_sig: int, optional
: Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] Number of channels the block runs on.

Q_sig: int
: Q format of the signal, number of bits after the decimal point.

attack_t
: [float] The attack time of the compressor/limiter in seconds; changing this property also sets the EWM alpha values for fixed and floating point implementations.

release_t
: [float] The release time of the compressor/limiter in seconds; changing this property also sets the EWM alpha values for fixed and floating point implementations.

threshold
: [float] Value below which expanding occurs for floating point processing.

gain
: [list[float]] Current gain to be applied to the signal for each channel for floating point processing.

attack_alpha
: [float] Attack time parameter used for exponential moving average in floating point processing.

release_alpha
: [float] Release time parameter used for exponential moving average in floating point processing.

threshold_int
: [int] Value below which expanding occurs for int32 fixed point processing.

gain_int
: [list[int]] Current gain to be applied to the signal for each channel for int32 fixed point processing.

attack_alpha_int
: [int] attack_alpha in 32-bit int format.

release_alpha_int
: [int] release_alpha in 32-bit int format.

gain_calc
: [function] function pointer to floating point gain calculation function.

gain_calc_int
: [function] function pointer to fixed point gain calculation function.

threshold_db
: [float] The threshold in decibels; changing this property also updates the fixed and floating point thresholds in linear gain.

env_detector
: [envelope_detector_peak] Peak envelope detector object used to calculate the envelope of the signal.

ratio
: [float] Expansion gain ratio applied when the signal is below the threshold; changing this property also updates the slope used in the fixed and floating point implementation.

slope
: [float] The slope factor of the expander, defined as *slope = 1 - ratio*.

slope_f32
: [float32] The slope factor of the expander, used for int32 to float32 processing.

process(*sample*, *channel=0*)

Update the envelope for a signal, then calculate and apply the required gain for expanding, using floating point maths.

Take one new sample and return the expanded sample. Input should be scaled with 0 dB = 1.0.

Parameters

sample
[float] The input sample to be processed.

channel
[int, optional] The channel index to process the sample on. Default is 0.

Returns

float
The processed sample.

reset_state()

Reset the envelope detector to 1 and the gain to 1, so the gate starts off.

Finite Impulse Response Filters

Finite impulse response (FIR) filters allow the use of arbitrary filters with a finite number of taps. This library does not provide FIR filter design tools, but allows for coefficients to be imported from other design tools, such as SciPy/filter_design.

FIR Direct The direct FIR implements the filter as a convolution in the time domain. This library uses FIR `filter_fir_s32` implementation from `lib_xcore_math` to run on xcore. More information on implementation can be found in XCORE Math Library documentation.

class audio_dsp.dsp.fir.**fir_direct**(*fs: float*, *n_chans: int*, *coeffs_path: Path*, *Q_sig: int = 27*)

An FIR filter, implemented in direct form in the time domain.

When the filter coefficients are converted to fixed point, if there will be leading zeros, a left shift is applied to the coefficients in order to use the full dynamic range of the VPU. A subsequent right shift is applied to the accumulator after the convolution to return to the same gain.

Parameters

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

coeffs_path
[Path] Path to a file containing the coefficients, in a format supported by np.loadtxt.

Q_sig: int, optional
Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.
Q_sig: int
Q format of the signal, number of bits after the decimal point.
coeffs
[np.ndarray] Array of the FIR coefficients in floating point format.
coeffs_int
[list] Array of the FIR coefficients in fixed point int32 format.
shift
[int] Right shift to be applied to the fixed point convolution result. This compensates for any left shift applied to the coefficients.
n_taps
[int] Number of taps in the filter.
buffer
[np.ndarray] Buffer of previous inputs for the convolution in floating point format.
buffer_int
[list] Buffer of previous inputs for the convolution in fixed point format.
buffer_idx
[list] List of the floating point buffer head for each channel.
buffer_idx_int
[list] List of the fixed point point buffer head for each channel.

process(*sample: float*, *channel: int = 0*) → float

Update the buffer with the current sample and convolve with the filter coefficients, using floating point math.

Parameters

sample
[float] The input sample to be processed.
channel
[int] The channel index to process the sample on.

Returns

float
The processed output sample.

reset_state() → None

Reset all the delay line values to zero.

check_coeff_scaling()

Check the coefficient scaling is optimal.
If there will be leading zeros, calculate a shift to use the full dynamic range of the VPU

Block Time Domain FIR The block time domain FIR implements the filter as a convolution in the time domain, but with a block size optimized for execution on the vector-unit of xcore.ai. The advantage with this one is it is over twice the efficiency of the lib_xcore_math implementation. This block will generate C code for the block time domain FIR filter. More information on implementation can be found in AN02027: Efficient computation of FIR filters on the XCORE.

Note: The block time domain FIR filter is not currently implemented as a DSP Stage, so cannot be used with the DSP pipeline tool yet.

Autogenerator

audio_dsp.dsp.td_block_fir.**generate_td_fir**(*td_coefs: ndarray*, *filter_name: str*, *output_path: Path*, *frame_advance=8*, *gain_db=0.0*, *verbose=False*)

Convert the input filter coefficients array into a header with block time domain structures to be included in a C project.

Parameters

td_coefs
: [np.ndarray] This is a 1D numpy float array of the coefficients of the filter.

filter_name
: [str] For use in identification of the filter from within the C code. All structs and defines that pertain to this filter will contain this identifier.

output_path
: [str] Where to output the resulting header file.

frame_advance
: [int, optional] The size in samples of a frame, measured in time domain samples, by default 8. Only multiples of 8 are supported.

gain_db
: [float, optional] A gain applied to the filter's output, by default 0.0

verbose
: [bool, optional] Enable verbose printing, by default False

Raises

ValueError: Bad config - Must be fixed

C API

void **td_block_fir_data_init**(td_block_fir_data_t *fir_data, int32_t *data, uint32_t data_buffer_elements)

Initialise a time domain block FIR data structure.

This manages the input data, rather than the coefficients, for a time domain block convolution. The python filter generator should be run first resulting in a header that defines the parameters for this function.

For example, running the generator with `--name={NAME}` would generate defines prepended with `{NAME}`, i.e. `{NAME}_DATA_BUFFER_ELEMENTS`, `{NAME}_TD_BLOCK_LENGTH`, etc. This function should then be called with:

```
td_block_fir_data_t {NAME}_fir_data;
int32_t {NAME}_data[{NAME}_DATA_BUFFER_ELEMENTS];
td_block_fir_data_init(&{NAME}_fir_data, {NAME}_data, {NAME}_DATA_BUFFER_ELEMENTS);
```

Parameters

- **fir_data** – Pointer to struct of type td_block_fir_data_t

- **data** – Pointer to an amount of memory to be used by the struct in order to hold a history of the samples. The define {NAME}_DATA_BUFFER_ELEMENTS specifies exactly the number of int32_t elements to allocate for the filter {NAME} to correctly function.
- **data_buffer_elements** – The number of words contained in the data array, this should be {NAME}_DATA_BUFFER_ELEMENTS.

void **td_block_fir_add_data**(int32_t samples_in[TD_BLOCK_FIR_LENGTH], td_block_fir_data_t *fir_data)

Function to add samples to the FIR data structure.

Parameters

- **samples_in** – Array of int32_t samples of length TD_BLOCK_FIR_LENGTH.
- **fir_data** – Pointer to struct of type td_block_fir_data_t to which the samples will be added.

void **td_block_fir_compute**(int32_t samples_out[TD_BLOCK_FIR_LENGTH], td_block_fir_data_t *fir_data, td_block_fir_filter_t *fir_filter)

Function to compute the convolution between fir_data and fir_filter.

Parameters

- **samples_out** – Array of length TD_BLOCK_FIR_LENGTH(8), which will be used to return the processed samples.
- **fir_data** – Pointer to struct of type td_block_fir_data_t from which the data samples will be obtained.
- **fir_filter** – Pointer to struct of type td_block_fir_filter_t from which the coefficients will be obtained.

Python API

class audio_dsp.dsp.td_block_fir.**fir_block_td**(*fs: float*, *n_chans: int*, *coeffs_path: Path*, *filter_name: str*, *output_path: Path*, *frame_advance=8*, *gain_db=0.0*, *Q_sig: int = 27*)

An FIR filter, implemented in block form in the time domain.

This will also autogenerate a .c and .h file containing the optimised block filter structures, designed for use in C.

Parameters

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

coeffs_path
[Path] Path to a file containing the coefficients, in a format supported by np.loadtxt.

filter_name
[str] Name of the filter, used for the autogen struct name

output_path
[Path] Output path for the autogenerated .c and .h files

frame_advance
[int, optional] Number of samples processed by the filter at once. This should be set to the same as the DSP pipeline frame size, and must be a multiple of 8.

gain_db
[float, optional] Additional gain applied by the filter

Q_sig: int, optional
Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

Q_sig: int
Q format of the signal, number of bits after the decimal point.

coeffs
[np.ndarray] Time domain coefficients

n_taps
[int] Length of time domain filter

frame_advance
[int] The number of new samples between subsequent frames.

buffer
[np.ndarray] Buffer of previous inputs for the convolution in floating point format.

buffer_int
[list] Buffer of previous inputs for the convolution in fixed point format.

`process_frame`(*frame: list*)

Update the buffer with the current samples and convolve with the filter coefficients, using floating point math.

Parameters

frame
[list[float]] The input samples to be processed.

Returns

float
The processed output sample.

`reset_state`() → None

Reset all the delay line values to zero.

Block Frequency Domain FIR This implementation is a frequency-domain implementation resulting in a lower algorithmic complexity than the time-domain versions. This will achieve the highest taps per second possible with the xcore. The main cost to using this implementation is the memory requirements double compared to the previous

two time-domain versions. This block will generate C code for the block frequency domain FIR filter. More information on implementation can be found in AN02027: Efficient computation of FIR filters on the XCORE.

Note: The block time domain FIR filter is not currently implemented as a DSP Stage, so cannot be used with the DSP pipeline tool yet.

Autogenerator

`audio_dsp.dsp.fd_block_fir.generate_fd_fir(`*td_coefs: ndarray*, *filter_name: str*, *output_path: Path*, *frame_advance: int*, *frame_overlap: int = 0*, *nfft: Optional[int] = None*, *gain_db: float = 0.0*, *verbose=False*`)`

Convert the input filter coefficients array into a header with block frequency domain structures to be included in a C project.

Parameters

td_coefs
: [np.ndarray] This is a 1D numpy float array of the coefficients of the filter.

filter_name
: [str] For use in identification of the filter from within the C code. All structs and defines that pertain to this filter will contain this identifier.

output_path
: [str] Where to output the resulting header file.

frame_advance
: [int] The number of new samples between subsequent frames.

frame_overlap
: [int, optional] The number of additional samples to output per frame. This allows windowing between frames to occur. By default no overlap occurs.

nfft
: [int, optional] The FFT size in samples of a frame, measured in time domain samples. If this is not set, the FFT size is set automatically. An initial attempt of `nfft = 2**(ceil(log2(frame_advance)) + 1)` is made, but may need to be increased for longer overlaps. If it is set, it must be a power of 2.

gain_db
: [float, optional] A gain applied to the filters output, by default 0.0

verbose
: [bool, optional] Enable verbose printing, by default False

Raises

ValueError: Bad config - Must be fixed

C API

void **fd_block_fir_data_init**(fd_fir_data_t *fir_data, int32_t *data, uint32_t frame_advance, uint32_t block_length, uint32_t block_count)

Initialise a frequency domain block FIR data structure.

This manages the input data, rather than the coefficients, for a frequency domain block convolution. The python filter generator should be run first resulting in a header that defines the parameters for this function.

For example, running the generator with `--name={NAME}` would generate defines prepended with `{NAME}`, i.e. `{NAME}_DATA_BUFFER_ELEMENTS`, `{NAME}_TD_BLOCK_LENGTH`, etc. This function should then be called with:

```
fd_fir_data_t {NAME}_fir_data;
int32_t {NAME}_data[{NAME}_DATA_BUFFER_ELEMENTS];
fd_block_fir_data_init(&{NAME}_fir_data, {NAME}_data,
    {NAME}_FRAME_ADVANCE,
    {NAME}_TD_BLOCK_LENGTH,
    {NAME}_BLOCK_COUNT);
```

Parameters

- **fir_data** – Pointer to struct of type fd_fir_data_t.
- **data** – An area of memory to be used by the struct in order to hold a history of the samples. The define {NAME}_DATA_BUFFER_ELEMENTS specifies exactly the number of int32_t elements to allocate for the filter {NAME} to correctly function.
- **frame_advance** – The number of samples contained in each frame, i.e. the samples count between updates. This should be initialised to {NAME}_FRAME_ADVANCE.
- **block_length** – The length of the processing block, independent to the frame_advance. Must be a power of two. This should be initialised to {NAME}_TD_BLOCK_LENGTH.
- **block_count** – The count of blocks required to implement the filter. This should be initialised to {NAME}_BLOCK_COUNT.

void **fd_block_fir_add_data**(int32_t *samples_in, fd_fir_data_t *fir_data)

Function to add samples to the FIR data structure.

Parameters

- **samples_in** – Array of int32_t samples of length expected to be fir_data->frame_advance.
- **fir_data** – Pointer to struct of type fd_fir_data_t to which the samples will be added.

void **fd_block_fir_compute**(int32_t *samples_out, fd_fir_data_t *fir_data, fd_fir_filter_t *fir_filter)

Function to compute the convolution between fir_data and fir_filter.

Parameters

- **samples_out** – Array of length fir_data->td_block_length, which will be used to return the processed samples. The samples will be returned from element 0 for `(fir_data-td_block_length + 1 - fir_filter->taps_per_block)` elements. The remaining samples of the array are used as scratch for the processing to be in-place.

- **fir_data** – Pointer to struct of type fd_fir_data_t from which the data samples will be obtained.
- **fir_filter** – Pointer to struct of type fd_fir_filter_t from which the coefficients will be obtained.

Python API

class audio_dsp.dsp.fd_block_fir.**fir_block_fd**(*fs: float*, *n_chans: int*, *coeffs_path: Path*, *filter_name: str*, *output_path: Path*, *frame_advance: int*, *frame_overlap: int = 0*, *nfft: Optional[int] = None*, *gain_db: float = 0.0*, *Q_sig: int = 27*)

An FIR filter, implemented in block form in the frequency domain.

This will also autogenerate a .c and .h file containing the optimised block filter structures, designed for use in C.

Parameters

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] Number of channels the block runs on.

coeffs_path
: [Path] Path to a file containing the coefficients, in a format supported by np.loadtxt.

filter_name
: [str] For use in identification of the filter from within the C code. All structs and defines that pertain to this filter will contain this identifier.

output_path
: [str] Where to output the resulting header file.

frame_advance
: [int] The number of new samples between subsequent frames.

frame_overlap
: [int, optional] The number of additional samples to output per frame. This allows windowing between frames to occur. By default no overlap occurs.

nfft
: [int, optional] The FFT size in samples of a frame, measured in time domain samples. If this is not set, the FFT size is set automatically. An initial attempt of `nfft = 2**(ceil(log2(frame_advance)) + 1)` is made, but may need to be increased for longer overlaps. If it is set, it must be a power of 2.

gain_db
: [float, optional] A gain applied to the filters output, by default 0.0

Q_sig: int, optional
: Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

Q_sig: int
Q format of the signal, number of bits after the decimal point.

coeffs
[np.ndarray] Time domain coefficients

n_taps
[int] Length of time domain filter

frame_advance
[int] The number of new samples between subsequent frames.

frame_overlap
[int] The number of additional samples to output per frame.

nfft
[int, optional] The FFT size in samples of a frame.

coeffs_fs
[np.ndarray] The frequency domain coefficients.

n_fd_buffers
[int] The number of frames of frequency domain coefficients, set the number of buffers that need to be saved.

td_buffer
[np.ndarray] Buffer of last nfft time domain inputs in floating point format

td_buffer_int
[list] Buffer of last nfft time domain inputs in fixed point format

fd_buffer
[np.ndarray] Buffer of last n_fd_buffers of the spectrums of previous td_buffers.

`process_frame`(*frame: list*)

Update the buffer with the current samples and convolve with the filter coefficients, using floating point math.

Parameters

frame
[list[float]] The input samples to be processed.

Returns

float
The processed output sample.

`reset_state`() → None

Reset all the delay line values to zero.

Graphic Equalisers

10 Band Graphic Equaliser

The graphic EQ module creates a 10 band equaliser, with octave spaced center frequencies. This can be used to The equaliser is implemented as a set of parallel 4th order bandpass filters, with a gain controlling the level of each parallel branch. The center frequencies are: [32, 64, 125, 250, 500, 1000, 2000, 4000, 8000, 16000].

C API

int32_t **adsp_graphic_eq_10b**(int32_t new_sample, int32_t gains[10], q2_30 coeffs[50], int32_t state[160])

10-band graphic equaliser

This function implements an 10-band graphic equalizer filter. The equaliser is implemented as a set of parallel 4th order bandpass filters, with a gain controlling the level of each parallel branch.

Note: The filter coefficients can be generated using `adsp_graphic_eq_10b_init`.

Parameters

- **new_sample** – New sample to be filtered
- **gains** – The gains of each band in Q_GEQ format
- **coeffs** – Filter coefficients
- **state** – Filter state, must be DWORD_ALIGNED

Returns

int32_t Filtered sample

See also *adsp_graphic_eq_10b_init()*.

Python API

class audio_dsp.dsp.graphic_eq.**graphic_eq_10_band**(*fs*, *n_chans*, *gains_db*, *gain_offset=-12*, *Q_sig=27*)

A 10 band graphic equaliser, with octave spaced center frequencies.

The equaliser is implemented as a set of parallel 4th order bandpass filters, with a gain controlling the level of each parallel branch. The center frequencies are: [32, 64, 125, 250, 500, 1000, 2000, 4000, 8000, 16000]. Due to the nature of the bandpass filters, frequencies below 25 Hz and above 19 kHz are filtered out. The filter coefficients have been hand tuned for common sample rates to minimise ripple in the combined output. As with analog graphic equalisers, interactions between neighbouring bands means the frequency response is not guaranteed to be equal to the slider positions.

Note that for a 32 kHz sample rate, the 16 kHz band is not available, making a 9 band EQ. For a 16 kHz sample rate the 8k and 16 kHz bands are not available, making an 8 band EQ.

The frequency response ripple with all the gains set to the same level is +/- 0.2 dB

Parameters

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

gains_db
[list[float]] A list of the 10 gains of the graphic eq in dB.

gain_offset
[float] Shifts the gains_db values by a number of decibels, by default -12dB to allow for an expected gains_db range of -12 to +12 dB. This means that setting gains_db to -gains_offset (+12dB) will not result in clipping, with the compromise that a gains_db value of 0dB will actually reduce the signal level by gain_offset (-12 dB).

Q_sig: int, optional
Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

Q_sig: int
Q format of the signal, number of bits after the decimal point.

`process`(*sample*, *channel=0*)

Process the input sample through the 10 band graphic equaliser using floating point maths.

Parameters

sample
[float] The input sample to be processed.

channel
[int] The channel index to process the sample on.

Returns

float
The processed output sample.

`property gains_db: list[float]`

A list of the gains in decibels for each frequency band. This must be a list with 10 values.

Reverb

Reverb Room The room reverb module imitates the reflections of a room. The algorithm is a Schroeder style reverberation, based on Freeverb by Jezar at Dreampoint. It consists of the wet predelay, 8 parallel comb filters fed into 4 series all-pass filters, with a wet and dry microphone control to set the effect level.

For more details on the algorithm, see Physical Audio Signal Processing by Julius Smith.

struct `reverb_room_t`

A room reverb filter structure.

Public Members

uint32_t `total_buffer_length`

Total buffer length

float **room_size**
Room size

int32_t **wet_gain**
Wet linear gain

int32_t **dry_gain**
Dry linear gain

int32_t **pre_gain**
Linear pre-gain

comb_fv_t **combs**[ADSP_RVR_N_COMBS]
Comb filters

allpass_fv_t **allpasses**[ADSP_RVR_N_APS]
Allpass filters

delay_t **predelay**
Predelay applied to the wet channel

C API

int32_t **adsp_reverb_room**(*reverb_room_t* *rv, int32_t new_samp)

Process a sample through a reverb room object.

Parameters

- **rv** – Reverb room object
- **new_samp** – New sample to process

Returns
int32_t Processed sample

Python API

class audio_dsp.dsp.reverb.**reverb_room**(*fs*, *n_chans*, *max_room_size=1*, *room_size=1*, *decay=0.5*, *damping=0.4*, *wet_gain_db=-1*, *dry_gain_db=-1*, *pregain=0.015*, *predelay=10*, *max_predelay=None*, *Q_sig=27*)

Generate a room reverb effect. This is based on Freeverb by Jezar at Dreampoint, and consists of 8 parallel comb filters fed into 4 series all-pass filters.

Parameters

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

max_room_size
[float, optional] sets the maximum size of the delay buffers, can only be set at initialisation.

room_size
: [float, optional] how big the room is as a proportion of max_room_size. This sets delay line lengths and must be between 0 and 1.

decay
: [int, optional] The length of the reverberation of the room, between 0 and 1.

damping
: [float, optional] how much high frequency attenuation in the room, between 0 and 1

wet_gain_db
: [int, optional] wet signal gain, less than 0 dB.

dry_gain_db
: [int, optional] dry signal gain, less than 0 dB.

pregain
: [float, optional] the amount of gain applied to the signal before being passed into the reverb, less than 1. If the reverb raises an OverflowWarning, this value should be reduced until it does not. The default value of 0.015 should be sufficient for most Q27 signals.

predelay
: [float, optional] the delay applied to the wet channel in ms.

max_predelay
: [float, optional] the maximum predelay in ms.

Q_sig: int, optional
: Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] Number of channels the block runs on.

Q_sig: int
: Q format of the signal, number of bits after the decimal point.

`pregain`
: [float] The pregain applied before the reverb as a floating point number.

pregain_int
: [int] The pregain applied before the reverb as a fixed point number.

`wet_db`
: [float] The gain applied to the wet signal in dB.

`wet`
: [float] The linear gain applied to the wet signal.

wet_int
: [int] The linear gain applied to the wet signal as a fixed point number.

`dry`
: [float] The linear gain applied to the dry signal.

`dry_db`
: [float] The gain applied to the dry signal in dB.

dry_int
[int] The linear gain applied to the dry signal as a fixed point number.

predelay
[float]

comb_lengths
[np.ndarray] An array of the comb filter delay line lengths, scaled by max_room_size.

ap_length
[np.ndarray] An array of the all pass filter delay line lengths, scaled by max_room_size.

combs
[list] A list of comb_fv objects containing the comb filters for the reverb.

allpasses
[list] A list of allpass_fv objects containing the all pass filters for the reverb.

`room_size`
[float] The room size as a proportion of the max_room_size.

`decay`
[float] The length of the reverberation of the room, between 0 and 1.

`feedback`
[float] Gain of the feedback line in the reverb filters.

feedback_int
[int] feedback as a fixed point integer.

`damping`
[float] How much high frequency attenuation in the room, between 0 and 1.

damping_int
[int] damping as a fixed point integer.

`process`(*sample*, *channel=0*)

Add reverberation to a signal, using floating point maths.

Take one new sample and return the sample with reverb. Input should be scaled with 0 dB = 1.0.

Parameters

sample
[float] The input sample to be processed.

channel
[int, optional] The channel index to process the sample on. Default is 0.

Returns

float
The processed sample.

`reset_state`()

Reset all the delay line values to zero.

`set_wet_dry_mix`(*mix*)

Will mix wet and dry signal by adjusting wet and dry gains. So that when the mix is 0, the output signal is fully dry, when 1, the output signal is fully wet. Tries to maintain a stable signal level using -4.5 dB Pan Law.

Parameters

mix
[float] The wet/dry mix, must be [0, 1].

set_pre_gain(*pre_gain*)

Deprecated since version 1.0.0: set_pre_gain will be removed in 2.0.0. Replace *reverb_room.set_pre_gain(x)* with *reverb_room.pregain = x*

Set the pre gain.

Parameters

pre_gain
[float] pre gain value, less than 1.

set_wet_gain(*wet_gain_db*)

Deprecated since version 1.0.0: set_wet_gain will be removed in 2.0.0. Replace *reverb_room.set_wet_gain(x)* with *reverb_room.wet_db = x*

Set the wet gain.

Parameters

wet_gain_db
[float] Wet gain in dB, less than 0 dB.

set_dry_gain(*dry_gain_db*)

Deprecated since version 1.0.0: set_dry_gain will be removed in 2.0.0. Replace *reverb_room.set_dry_gain(x)* with *reverb_room.dry_db = x*

Set the dry gain.

Parameters

dry_gain_db
[float] Dry gain in dB, lees than 0 dB.

set_decay(*decay*)

Deprecated since version 1.0.0: set_decay will be removed in 2.0.0. Replace *reverb_room.set_decay(x)* with *reverb_room.decay = x*

Set the decay of the reverb.

Parameters

decay
[float] How long the reverberation of the room is, between 0 and 1.

set_damping(*damping*)

Deprecated since version 1.0.0: set_damping will be removed in 2.0.0. Replace *reverb_room.set_damping(x)* with *reverb_room.damping = x*

Set the damping of the reverb.

Parameters

damping
[float] How much high frequency attenuation in the room, between 0 and 1.

set_room_size(*room_size*)

Deprecated since version 1.0.0: set_room_size will be removed in 2.0.0. Replace *reverb_room.set_room_size(x)* with *reverb_room.room_size = x*

Set the current room size; will adjust the delay line lengths accordingly.

Parameters

room_size
[float] How big the room is as a proportion of max_room_size. This sets delay line lengths and must be between 0 and 1.

Reverb Room Stereo The stereo room reverb module extends the mono *Reverb Room* by adding a second set of comb and all-pass filters in parallel, and mixing the output of the two networks. Varying the mix of the networks changes the stereo width of the effect.

For more details on the algorithm, see Physical Audio Signal Processing by Julius Smith.

struct **reverb_room_st_t**

A stereo room reverb filter structure.

Public Members

uint32_t **total_buffer_length**

Total buffer length

uint32_t **spread_length**

Spread length

float **room_size**

Room size

int32_t **wet_gain1**

Wet 1 linear gain

int32_t **wet_gain2**

Wet 2 linear gain

int32_t **dry_gain**

Dry linear gain

int32_t **pre_gain**

Linear pre-gain

comb_fv_t **combs**[2][ADSP_RVR_N_COMBS]

Comb filters, 0:left, 1:right

allpass_fv_t **allpasses**[2][ADSP_RVR_N_APS]

Allpass filters, 0:left, 1:right

delay_t **predelay**

Predelay applied to the wet channel

C API

void **adsp_reverb_room_st**(*reverb_room_st_t* *rv, int32_t outputs_lr[2], int32_t in_left, int32_t in_right)

Process samples through a stereo reverb room object.

Parameters

- **rv** – Stereo reverb room object
- **outputs_lr** – Pointer to the outputs 0:left, 1:right
- **in_left** – New left sample to process
- **in_right** – New right sample to process

Python API

class audio_dsp.dsp.reverb_stereo.**reverb_room_stereo**(*fs*, *n_chans*, *max_room_size=1*, *room_size=1*, *decay=0.5*, *damping=0.4*, *width=1.0*, *wet_gain_db=-1*, *dry_gain_db=-1*, *pre-gain=0.0075*, *predelay=10*, *max_predelay=None*, *Q_sig=27*)

Generate a stereo room reverb effect. This is based on Freeverb by Jezar at Dreampoint. Each channel consists of 8 parallel comb filters fed into 4 series all-pass filters, and the reverberator outputs are mixed according to the **width** parameter.

Parameters

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] Number of channels the block runs on.

max_room_size
: [float, optional] sets the maximum size of the delay buffers, can only be set at initialisation.

room_size
: [float, optional] how big the room is as a proportion of max_room_size. This sets delay line lengths and must be between 0 and 1.

decay
: [int, optional] The length of the reverberation of the room, between 0 and 1.

damping
: [float, optional] how much high frequency attenuation in the room, between 0 and 1

width
: [float, optional] how much stereo separation there is between

the left and right channels. Setting width to 0 will yield a mono signal, whilst setting width to 1 will yield the most stereo separation.

wet_gain_db
[int, optional] wet signal gain, less than 0 dB.

dry_gain_db
[int, optional] dry signal gain, less than 0 dB.

pregain
[float, optional] the amount of gain applied to the signal before being passed into the reverb, less than 1. If the reverb raises an OverflowWarning, this value should be reduced until it does not. The default value of 0.015 should be sufficient for most Q27 signals.

predelay
[float, optional] the delay applied to the wet channel in ms.

max_predelay
[float, optional] the maximum predelay in ms.

Q_sig: int, optional
Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

Q_sig: int
Q format of the signal, number of bits after the decimal point.

pregain
[float] The pregain applied before the reverb as a floating point number.

pregain_int
[int] The pregain applied before the reverb as a fixed point number.

wet_db
[float] The gain applied to the wet signal in dB.

wet
[float] The linear gain applied to the wet signal.

wet_int
[int] The linear gain applied to the wet signal as a fixed point number.

dry
[float] The linear gain applied to the dry signal.

dry_db
[float] The gain applied to the dry signal in dB.

dry_int
[int] The linear gain applied to the dry signal as a fixed point number.

predelay
[float]

width
[float] Stereo separation of the reverberated signal.

comb_lengths
[np.ndarray] An array of the comb filter delay line lengths, scaled by max_room_size.

ap_length
[np.ndarray] An array of the all pass filter delay line lengths, scaled by max_room_size.

combs
[list] A list of comb_fv objects containing the comb filters for the reverb.

allpasses
[list] A list of allpass_fv objects containing the all pass filters for the reverb.

room_size
[float] The room size as a proportion of the max_room_size.

decay
[float] The length of the reverberation of the room, between 0 and 1.

feedback
[float] Gain of the feedback line in the reverb filters.

feedback_int
[int] feedback as a fixed point integer.

damping
[float] How much high frequency attenuation in the room, between 0 and 1.

damping_int
[int] damping as a fixed point integer.

process(*sample*, *channel=0*)

Process is not implemented for the stereo reverb, as it needs 2 channels at once.

Parameters

sample
[float] The input sample to be processed.

channel
[int, optional] The channel index to process the sample on. Default is 0.

Returns

float
The processed sample.

reset_state()

Reset all the delay line values to zero.

set_wet_dry_mix(*mix*)

Will mix wet and dry signal by adjusting wet and dry gains. So that when the mix is 0, the output signal is fully dry, when 1, the output signal is fully wet. Tries to maintain a stable signal level using -4.5 dB Pan Law.

Parameters

mix
[float] The wet/dry mix, must be [0, 1].

property wet_db

The gain applied to the wet signal in dB.

property dry_db

The gain applied to the dry signal in dB.

property decay

The length of the reverberation of the room, between 0 and 1.

property damping

How much high frequency attenuation in the room, between 0 and 1.

Reverb Plate Stereo The plate reverb module imitates the reflections of a plate reverb, which has more early reflections than the room reverb. The algorithm is based on Dattorro's 1997 paper. This reverb consists of 4 allpass filters for input diffusion, followed by a figure of 8 reverb tank of allpasses, low-pass filters, and delays. The output is taken from multiple taps in the delay lines to get a desirable echo density. The left and right output can be mixed with various widths.

For more details on the algorithm, see Effect Design, Part 1: Reverberator and Other Filters by Jon Dattorro.

struct **reverb_plate_t**

A plate reverb structure.

Public Members

int32_t **decay**

Reverb decay

int32_t **wet_gain1**

Wet 1 linear gain

int32_t **wet_gain2**

Wet 2 linear gain

int32_t **dry_gain**

Dry linear gain

int32_t **pre_gain**

Linear pre-gain

int32_t **paths**[ADSP_RVP_N_PATHS]

Saved output paths

int32_t **taps_l**[ADSP_RVP_N_OUT_TAPS]

Indexes for the left channel calculation

int32_t **taps_len_l**[ADSP_RVP_N_OUT_TAPS]

Max lenghts of buffers use for the left channel calculation

int32_t **taps_r**[ADSP_RVP_N_OUT_TAPS]

Indexes for the right channel calculation

int32_t **taps_len_r**[ADSP_RVP_N_OUT_TAPS]

Max lenghts of buffers use for the right channel calculation

lowpass_1ord_t **lowpasses**[ADSP_RVP_N_LPS]

FIrst order lowpass filters

allpass_fv_t **mod_allpasses**[ADSP_RVP_N_PATHS]

Modulated allpass filters

allpass_fv_t **allpasses**[ADSP_RVP_N_APS]

Allpass filters

delay_t **delays**[ADSP_RVP_N_DELAYS]

Delay lines

delay_t **predelay**

Predelay applied to the wet channel

C API

void **adsp_reverb_plate**(*reverb_plate_t* *rv, int32_t outputs_lr[2], int32_t in_left, int32_t in_right)

Process samples through a reverb plate object.

Parameters

- **rv** – Reverb plate object
- **outputs_lr** – Pointer to the outputs 0:left, 1:right
- **in_left** – New left sample to process
- **in_right** – New right sample to process

Python API

class audio_dsp.dsp.reverb_plate.**reverb_plate_stereo**(*fs*, *n_chans*, *decay=0.4*, *damping=0.75*, *bandwidth=8000*, *early_diffusion=0.75*, *late_diffusion=0.7*, *width=1.0*, *wet_gain_db=-3*, *dry_gain_db=-3*, *pregain=0.5*, *predelay=10*, *max_predelay=None*, *Q_sig=27*)

Generate a stereo plate reverb effect, based on Dattorro's 1997 paper. This reverb consists of 4 allpass filters for input diffusion, followed by a figure of 8 reverb tank of allpasses, low-pass filters, and delays. The output is taken from multiple taps in the delay lines to get a desirable echo density.

Parameters

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

decay
[int, optional] The length of the reverberation of the room, between 0 and 1.

damping
[float, optional] How much high frequency attenuation in the room, between 0 and 1

bandwidth
[float, optional] Controls the low pass filter cutoff frequency at the start of the reverb, in Hz.

early_diffusion
[float, optional] Controls how much diffusion the early echoes have.

late_diffusion
[float, optional] Controls how much diffusion the late echoes have.

width
[float, optional] how much stereo separation there is between the left and right channels. Setting width to 0 will yield a mono signal, whilst setting width to 1 will yield the most stereo separation.

wet_gain_db
[int, optional] wet signal gain, less than 0 dB.

dry_gain_db
[int, optional] dry signal gain, less than 0 dB.

pregain
[float, optional] the amount of gain applied to the signal before being passed into the reverb, less than 1. If the reverb raises an OverflowWarning, this value should be reduced until it does not. The default value of 0.5 should be sufficient for most Q27 signals, and should be reduced by 1 bit per increase in Q format, e.g. 0.25 for Q28, 0.125 for Q29 etc.

predelay
[float, optional] the delay applied to the wet channel in ms.

max_predelay
[float, optional] the maximum predelay in ms.

Q_sig: int, optional
Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

Q_sig: int
Q format of the signal, number of bits after the decimal point.

pregain
[float] The pregain applied before the reverb as a floating point number.

pregain_int
[int] The pregain applied before the reverb as a fixed point number.

wet_db
[float] The gain applied to the wet signal in dB.

wet
[float] The linear gain applied to the wet signal.

wet_int
[int] The linear gain applied to the wet signal as a fixed point number.

dry
[float] The linear gain applied to the dry signal.

dry_db
[float] The gain applied to the dry signal in dB.

dry_int
[int] The linear gain applied to the dry signal as a fixed point number.

predelay
[float]

width
[float] Stereo separation of the reverberated signal.

allpasses
[list] A list of allpass objects containing the all pass filters for the reverb.

lowpasses
[list] A list of lowpass objects containing the low pass filters for the reverb.

delays
[list] A list of delay objects containing the delay lines for the reverb.

mod_allpasses
[list] A list of allpass objects containing the modulated all pass objects for the reverb.

taps_l
[list] A list of the current output tap locations for the left output.

taps_r
[list] A list of the current output tap locations for the right output.

tap_lens_l
[list] A list of the buffer lengths used by taps_l, to aid wrapping the read head at the end of the buffer

tap_lens_r
[list] As tap lens_l, but for the right output channel.

decay
[float] The length of the reverberation of the room, between 0 and 1.

decay_int
[int] decay as a fixed point integer.

damping
[float] How much high frequency attenuation in the room, between 0 and 1.

damping_int
[int] damping as a fixed point integer.
bandwidth
[float] The bandwidth of the reverb input signal, in Hertz.
early_diffusion
[float] How much early diffusion in the reverb, between 0 and 1.
late_diffusion
[float] How much late diffusion in the reverb, between 0 and 1.

process(*sample*, *channel=0*)

Process is not implemented for the stereo reverb, as it needs 2 channels at once.

Parameters

sample
[float] The input sample to be processed.
channel
[int, optional] The channel index to process the sample on. Default is 0.

Returns

float
The processed sample.

reset_state()

Reset all the delay line values to zero.

set_wet_dry_mix(*mix*)

Will mix wet and dry signal by adjusting wet and dry gains. So that when the mix is 0, the output signal is fully dry, when 1, the output signal is fully wet. Tries to maintain a stable signal level using -4.5 dB Pan Law.

Parameters

mix
[float] The wet/dry mix, must be [0, 1].

Signal Chain Components

Signal chain components includes DSP modules for: * combining signals, such as subtracting, adding, and mixing * forks for splitting signals * basic gain components, such as fixed gain, volume control, and mute * basic delay buffers.

Adder The adder will add samples from N inputs together. It will round and saturate the result to the Q0.31 range.

C API

int32_t **adsp_adder**(int32_t *input, unsigned n_ch)

Saturating addition of an array of samples.

Note: Will work for any q format

Parameters

- **input** – Array of samples
- **n_ch** – Number of channels

Returns

int32_t Sum of samples

Python API

class audio_dsp.dsp.signal_chain.**adder**(*fs: float*, *n_chans: int*, *Q_sig: int = 27*)

A class representing an adder in a signal chain.

This class inherits from the *mixer* class and provides an adder with no attenuation.

Parameters

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

Q_sig: int, optional
Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

Q_sig: int
Q format of the signal, number of bits after the decimal point.

gain_db
[float] The mixer gain in decibels.

gain
[float] Gain as a linear value.

gain_int
[int] Gain as an integer value.

process_channels(*sample_list: list[float]*) → list[float]

Process a single sample. Apply the gain to all the input samples then sum them using floating point maths.

Parameters

sample_list
[list] List of input samples

Returns

list[float]
Output sample.

Subtractor The subtractor will subtract one sample from another, then round and saturate the difference to Q0.31 range.

C API

int32_t **adsp_subtractor**(int32_t x, int32_t y)

Saturating subtraction of two samples, this returns `x - y`.

Note: Will work for any q format

Parameters

- **x** – Minuend
- **y** – Subtrahend

Returns

int32_t Difference

Python API

class audio_dsp.dsp.signal_chain.**subtractor**(*fs: float*, *Q_sig: int = 27*)

Subtractor class for subtracting two signals.

Parameters

fs
[int] Sampling frequency in Hz.

Q_sig: int, optional
Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

Q_sig: int
Q format of the signal, number of bits after the decimal point.

process_channels(*sample_list: list[float]*) → list[float]

Subtract the second input sample from the first using floating point maths.

Parameters

sample_list
[list[float]] List of input samples.

Returns

float
Result of the subtraction.

Fixed Gain This module applies a fixed gain to a sample, with rounding and saturation to Q0.31 range. The gain must be in **Q_GAIN** format.

Q_GAIN

Gain format to be used in the gain APIs

C API

int32_t **adsp_fixed_gain**(int32_t input, int32_t gain)

Fixed-point gain.

Note: One of the inputs has to be in Q_GAIN format

Parameters

- **input** – Input sample
- **gain** – Gain

Returns

int32_t Output sample

Python API

class audio_dsp.dsp.signal_chain.**fixed_gain**(*fs: float*, *n_chans: int*, *gain_db: float*, *Q_sig: int = 27*)

Multiply every sample by a fixed gain value.

In the current implementation, the maximum boost is +24 dB.

Parameters

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] Number of channels the block runs on.

gain_db
: [float] The gain in decibels. Maximum fixed gain is +24 dB.

Q_sig: int, optional
: Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] Number of channels the block runs on.

Q_sig: int
: Q format of the signal, number of bits after the decimal point.

gain_db
: [float] The mixer gain in decibels.

gain
: [float] Gain as a linear value.

gain_int
: [int] Gain as an integer value.

process(*sample: float*, *channel: int = 0*) → float

Multiply the input sample by the gain, using floating point maths.

Parameters

sample
[float] The input sample to be processed.

channel
[int] The channel index to process the sample on, not used by this module.

Returns

float
The processed output sample.

Mixer The mixer applies a gain to all N channels of input samples and adds them together. The sum is rounded and saturated to Q0.31 range. The gain must be in `Q_GAIN` format.

C API

int32_t **adsp_mixer**(int32_t *input, unsigned n_ch, int32_t gain)

Mixer. Will add signals with gain applied to each signal before mixing.

Note: Inputs or gain have to be in Q_GAIN format

Parameters

- **input** – Array of samples
- **n_ch** – Number of channels
- **gain** – Gain

Returns
int32_t Mixed sample

An alternative way to implement a mixer is to multiply-accumulate the input samples into a 64-bit word, then saturate it to a 32-bit word using:

int32_t **adsp_saturate_32b**(int64_t acc)

Saturating 64-bit accumulator. Will saturate to 32-bit, so that the output value is in the range of int32_t.

Parameters

- **acc** – Accumulator

Returns
int32_t Saturated value

Python API

class audio_dsp.dsp.signal_chain.**mixer**(*fs: float*, *n_chans: int*, *gain_db: float = -6*, *Q_sig: int = 27*)

Mixer class for adding signals with attenuation to maintain headroom.

Parameters

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.
gain_db
[float] Gain in decibels (default is -6 dB).
Q_sig: int, optional
Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
[int] Sampling frequency in Hz.
n_chans
[int] Number of channels the block runs on.
Q_sig: int
Q format of the signal, number of bits after the decimal point.
gain_db
[float] The mixer gain in decibels.
gain
[float] Gain as a linear value.
gain_int
[int] Gain as an integer value.

process_channels(*sample_list: list[float]*) → list[float]

Process a single sample. Apply the gain to all the input samples then sum them using floating point maths.

Parameters

sample_list
[list] List of input samples

Returns

list[float]
Output sample.

Volume Control The volume control allows safe real-time gain adjustments with minimal artifacts. When the target gain is changed, a slew is used to move from the current gain to the target gain. This allows smooth gain change and no clicks in the output signal.

The mute API allows the user to safely mute the signal by setting the target gain to **0**, with the slew ensuring no pops or clicks. Unmuting will restore the pre-mute target gain. The new gain can be set while muted, but will not take effect until unmute is called. There are separate APIs for process, setting the gain, muting and unmuting so that volume control can easily be implemented into the control system.

The slew is applied as an exponential of the difference between the target and current gain. For run-time efficiency, instead of an EMA-style alpha, the difference is right shifted by the **slew_shift** parameter. The relation between **slew_shift** and time is further discussed in the Python class documentation.

struct **volume_control_t**

Volume control state structure.

Public Members

int32_t **target_gain**
: Target linear gain

int32_t **gain**
: Current linear gain

int32_t **slew_shift**
: Slew shift

int32_t **saved_gain**
: Saved linear gain

uint8_t **mute_state**
: Mute state: 0: unmuted, 1 muted

C API

int32_t **adsp_volume_control**(*volume_control_t* *vol_ctl, int32_t samp)

Process a new sample with a volume control.

Parameters

- **vol_ctl** – Volume control object
- **samp** – New sample

Returns

int32_t Processed sample

void **adsp_volume_control_set_gain**(*volume_control_t* *vol_ctl, int32_t new_gain)

Set the target gain of a volume control.

Parameters

- **vol_ctl** – Volume control object
- **new_gain** – New target linear gain

void **adsp_volume_control_mute**(*volume_control_t* *vol_ctl)

Mute a volume control. Will save the current target gain and set the target gain to 0.

Parameters

- **vol_ctl** – Volume control object

void **adsp_volume_control_unmute**(*volume_control_t* *vol_ctl)

Unmute a volume control. Will restore the saved target gain.

Parameters

- **vol_ctl** – Volume control object

Python API

class audio_dsp.dsp.signal_chain.**volume_control**(*fs: float*, *n_chans: int*, *gain_db: float = -6*, *slew_shift: int = 7*, *mute_state: int = 0*, *Q_sig: int = 27*)

A volume control class that allows setting the gain in decibels. When the gain is updated, an exponential slew is applied to reduce artifacts.

The slew is implemented as a shift operation. The slew rate can be converted to a time constant using the formula: *time_constant = -1/ln(1 - 2^-slew_shift) * (1/fs)*

A table of the first 10 slew shifts is shown below:

slew_shift	Time constant (ms)
1	0.03
2	0.07
3	0.16
4	0.32
5	0.66
6	1.32
7	2.66
8	5.32
9	10.66
10	21.32

Parameters

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

gain_db
[float, optional] The initial gain in decibels

slew_shift
[int, optional] The shift value used in the exponential slew.

mute_state
[int, optional] The mute state of the Volume Control: 0: unmuted, 1: muted.

Q_sig: int, optional
Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Raises

ValueError
If the gain_db parameter is greater than 24 dB.

Attributes

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

Q_sig: int
Q format of the signal, number of bits after the decimal point.

target_gain_db
[float] The target gain in decibels.

target_gain
[float] The target gain as a linear value.

target_gain_int
[int] The target gain as a fixed-point integer value.

gain_db
[float] The current gain in decibels.

gain
[float] The current gain as a linear value.

gain_int
[int] The current gain as a fixed-point integer value.

slew_shift
[int] The shift value used in the exponential slew.

mute_state
[int] The mute state of the Volume Control: 0: unmuted, 1: muted

process(*sample: float*, *channel: int = 0*) → float

Update the current gain, then multiply the input sample by it, using floating point maths.

Parameters

sample
[float] The input sample to be processed.

channel
[int, optional] The channel index to process the sample on. Not used by this module.

Returns

float
The processed output sample.

set_gain(*gain_db: float*) → None

Deprecated since version 1.0.0: set_gain will be removed in 2.0.0. Replace *volume_control.set_gain(x)* with *volume_control.target_gain_db = x*

Set the gain of the volume control.

Parameters

gain_db
[float] The gain in decibels. Must be less than or equal to 24 dB.

Raises

ValueError
If the gain_db parameter is greater than 24 dB.

mute() → None

Mute the volume control.

unmute() → None

Unmute the volume control.

Delay The delay module uses a memory buffer to return a sample after a specified time period. The returned samples will be delayed by a specified value. The `max_delay` is set at initialisation, and sets the amount of memory used by the buffers. It cannot be changed at runtime. The current `delay` value can be changed at runtime within the range `[0, max_delay]`

struct **delay_t**

Delay state structure.

Public Members

float **fs**

Sampling frequency

uint32_t **delay**

Current delay in samples

uint32_t **max_delay**

Maximum delay in samples

uint32_t **buffer_idx**

Current buffer index

int32_t ***buffer**

Buffer

C API

int32_t **adsp_delay**(*delay_t* *delay, int32_t samp)

Process a new sample through a delay object.

Note: The minimum delay provided by this block is 1 sample. Setting the delay to 0 will still yield a 1 sample delay.

Parameters

- **delay** – Delay object
- **samp** – New sample

Returns

int32_t Oldest sample

Python API

class audio_dsp.dsp.signal_chain.**delay**(*fs*, *n_chans*, *max_delay: float*, *starting_delay: float*, *units: str = 'samples'*)

A simple delay line class.

Note the minimum delay provided by this block is 1 sample. Setting the delay to 0 will still yield a 1 sample delay.

Parameters

fs
[int] Sampling frequency in Hz.
n_chans
[int] Number of channels the block runs on.
max_delay
[float] The maximum delay in specified units.
starting_delay
[float] The starting delay in specified units.
units
[str, optional] The units of the delay, can be 'samples', 'ms' or 's'. Default is 'samples'.

Attributes

fs
[int] Sampling frequency in Hz.
n_chans
[int] Number of channels the block runs on.
Q_sig: int
Q format of the signal, number of bits after the decimal point.
max_delay
[int] The maximum delay in samples.
delay
[int] The delay in samples.
buffer
[np.ndarray] The delay line buffer.
buffer_idx
[int] The current index of the buffer.

process_channels(*sample_list: list[float]*) → list[float]

Put the new sample in the buffer and return the oldest sample.

Parameters

sample_list
[list[float]] The input samples to be processed. Each sample represents a different channel

Returns

float
List of delayed samples.

reset_state() → None

Reset all the delay line values to zero.

set_delay(*delay: float*, *units: str = 'samples'*) → None

Set the length of the delay line, will saturate at max_delay.

Parameters

delay
[float] The delay length in specified units.
units
[str, optional] The units of the delay, can be 'samples', 'ms' or 's'. Default is 'samples'.

Switch with slew The slewing switch module uses a cosine crossfade when moving switch position in order to avoid clicks.

struct **switch_slew_t**

Slewing switch state structure.

Public Members

bool **switching**

If slewing, switching is True until slewing is over.

int32_t **position**

Current switch pole position.

int32_t **last_position**

Last switch pole position.

int32_t **counter**

Counter for timing slew length.

int32_t **step**

Step increment of counter.

C API

int32_t **adsp_switch_slew**(*switch_slew_t* *switch_slew, int32_t *samples)

Process a sample through a slewing switch. If the switch position has recently changed, this will slew between the desired input channel and previous channel.

Parameters

- **switch_slew** – Slewing switch state object.
- **samples** – An array of input samples for each input channel.

Returns

int32_t The output of the switch.

Python API

class audio_dsp.dsp.signal_chain.**switch_slew**(*fs*, *n_chans*, *Q_sig: int = 27*)

A class representing a switch in a signal chain. When the switch is moved, a cosine crossfade is used to slew between the positions.

The cosine crossfade is implemented as a polynomial, with coefficients derived from a Chebyshev polynomial fit.

Parameters

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

Q_sig: int, optional
Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

Q_sig: int
Q format of the signal, number of bits after the decimal point.

switch_position
[int] The current position of the switch.

switching
[bool] True if the switch is in the process of moving.

step
[int] Step size used for cosine calculation.

counter
[int] Counter used dor cosine calculation.

p_coef
[list[float]] Polynomial cosine approximation coefficients as floats.

p_coef_int
[list[int]] Polynomial cosine approximation coefficients as ints.

process_channels(*sample_list: list[float]*) → list[float]

Return the sample at the current switch position.

This method takes a list of samples and returns the sample at the current switch position. If the switch position has recently changed, it will slew between the inputs.

Parameters

sample_list
[list] A list of samples for each of the switch inputs.

Returns

y
[float] The sample at the current switch position.

move_switch(*position: int*) → None

Move the switch to the specified position. This will cause the channel in sample_list[position] to be output.

Parameters

position
[int] The position to move the switch to.

Crossfader The crossfader mixes between two sets of inputs.

C API

static inline int32_t **adsp_crossfader**(int32_t in1, int32_t in2, int32_t gain1, int32_t gain2, int32_t q_gain)

Crossfade between two channels using their gains. Will do: (in1 * gain1) + (in2 * gain2).

Parameters

- **in1** – First signal
- **in2** – Second signal
- **gain1** – First gain
- **gain2** – Second gain
- **q_gain** – Q factor of the gain

Returns

int32_t Mixed signal

Python API

Only a slewing crossfader Python API is provided.

Crossfader with slew The crossfader mixes between two sets of inputs, with slew applied to the gains when they are changed.

struct **crossfader_slew_t**

Slewing crossfader state structure.

Public Members

gain_slew_t **gain_1**

Slewing gain struct for first crossfader input.

gain_slew_t **gain_2**

Slewing gain struct for second crossfader input.

float **mix**

Mix of the inputs.

C API

int32_t **adsp_crossfader_slew**(*crossfader_slew_t* *crossfader, int32_t in1, int32_t in2)

Crossfade between two channels with slew applied to the gains. Will do: (in1 * crossfader->gain1.gain) + (in2 * crossfader->gain2.gain).

Parameters

- **crossfader** – Slewing crossfader state object.
- **in1** – First signal
- **in2** – Second signal

Returns

int32_t Mixed signal

Python API

class audio_dsp.dsp.signal_chain.**crossfader**(*fs: float*, *n_chans: int*, *mix: float = 0.5*, *slew_shift: int = 7*, *Q_sig: int = 27*)

The crossfader mixes between two sets of inputs. The mix control sets the respective levels of each input. When the mix is updated, an exponential slew is applied to reduce artifacts.

The slew is implemented as a shift operation. The slew rate can be converted to a time constant using the formula: *time_constant = -1/ln(1 - 2^-slew_shift) * (1/fs)*

Parameters

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

mix
[float] The channel mix, must be set between [0, 1]

slew_shift
[int, optional] The shift value used in the exponential slew.

Q_sig: int, optional
Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

Q_sig: int
Q format of the signal, number of bits after the decimal point.

n_outs
[int] Number of outputs, half the number of inputs.

gains
[list[float]] Floating point gains for each input for a given mix value.

gains_int
[list[int]] Fixed point gains for each input for a given mix value.

slew_shift
[int] The shift value used in the exponential slew.

process_channels(*sample_list: list[float]*) → list[float]

Process a single sample. Apply the crossfader gain to all the input samples using floating point maths.

Parameters

sample_list
[list] List of input samples

Returns

list[float]
Output sample.

Python module base class

All the Python DSP modules are based on a common base class. In order to keep the documentation short, all Python classes in the previous sections only had the `process` method described, and control methods where necessary. This section provides the user with a more in-depth information of the Python API, which may be useful when adding custom DSP modules.

Some classes overload the base class APIs where they require different input data types or dimensions. However, they will all have the attributes and methods described below.

The process methods can be split into 2 groups:

1) `process` is a 64b floating point implementation

2) `process_xcore` is a 32b fixed-point implementation, with the aim of being bit exact with the C/assembly implementation.

The `process_xcore` methods can be used to simulate the xcore implementation precision and the noise floor. The Python `process_xcore` implementations have very similar accuracy to the xcore C `adsp_*` implementations (subject to the module and implementation). Python simulation methods tend to be slower as Python has a limited support for the fixed point processing. Bit exactness is not always possible for modules that use 32b float operations, as the rounding of these can differ between C libraries.

There are 3 layers of `process` functions:

1) `process` operates for a single sample on a single channel

2) `process_channels` operates on all channels for a single sample

3) `process_frame` operates on all samples and channels for a single frame.

A DSP module may overload one or all of these, depending if it operates sample-wise, channel-wise or frame-wise. It is expected that `process_frame` will be called by higher level DSP Stages.

class audio_dsp.dsp.generic.**dsp_block**(*fs*, *n_chans*, *Q_sig=27*)

Generic DSP block, all blocks should inherit from this class and implement it's methods.

By using the metaclass NumpyDocstringInheritanceInitMeta, parameter and attribute documentation can be inherited by the child classes.

Parameters

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] Number of channels the block runs on.

Q_sig: int, optional
: Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs
: [int] Sampling frequency in Hz.

n_chans
: [int] Number of channels the block runs on.

Q_sig: int
Q format of the signal, number of bits after the decimal point.

freq_response(*nfft=32768*)
Calculate the frequency response of the module for a nominal input.
The generic module has a flat frequency response.

Parameters

nfft
[int, optional] The number of points to use for the FFT, by default 512

Returns

tuple
A tuple containing the frequency values and the corresponding complex response.

process(*sample: float*, *channel=0*)
Take one new sample and give it back. Do no processing for the generic block.

Parameters

sample
[float] The input sample to be processed.

channel
[int, optional] The channel index to process the sample on. Default is 0.

Returns

float
The processed sample.

process_channels(*sample_list: list[float]*) → list[float]
Process the sample in each audio channel using floating point maths.
The generic implementation calls self.process for each channel.

Parameters

sample_list
[list[float]] The input samples to be processed. Each sample represents a different channel

Returns

list[float]
The processed samples for each channel.

process_channels_xcore(*sample_list: list[float]*) → list[float]
Process the sample in each audio channel using fixed point maths.
The generic implementation calls self.process_xcore for each channel.

Parameters

sample_list
[list[float]] The input samples to be processed. Each sample represents a different channel

Returns

list[float]
The processed samples for each channel.

process_frame(*frame: list*)

Take a list frames of samples and return the processed frames.
A frame is defined as a list of 1-D numpy arrays, where the number of arrays is equal to the number of channels, and the length of the arrays is equal to the frame size.
For the generic implementation, just call process for each sample for each channel.

Parameters

frame
[list] List of frames, where each frame is a 1-D numpy array.

Returns

list
List of processed frames, with the same structure as the input frame.

process_frame_xcore(*frame: list*)

Take a list frames of samples and return the processed frames, using an xcore-like implementation.
A frame is defined as a list of 1-D numpy arrays, where the number of arrays is equal to the number of channels, and the length of the arrays is equal to the frame size.
For the generic implementation, just call process for each sample for each channel.

Parameters

frame
[list] List of frames, where each frame is a 1-D numpy array.

Returns

list
List of processed frames, with the same structure as the input frame.

process_xcore(*sample: float*, *channel=0*)

Take one new sample and return 1 processed sample.
For the generic implementation, scale and quantize the input, call the xcore-like implementation, then scale back to 1.0 = 0 dB.

Parameters

sample
[float] The input sample to be processed.

channel
[int, optional] The channel index to process the sample on. Default is 0.

Returns

float
The processed output sample.

5.3 Integration and Control

This section covers the API necessary to integrate the generated DSP pipeline into your application, both data and control-wise. It will introduce the API necessary for converting from high-level DSP parameters, to the ones that can be sent to the DSP pipeline.

Pipeline

Generated pipeline interface. Use the source and sink functions defined here to send samples to the generated DSP and receive processed samples back.

Functions

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

Pass samples into the DSP pipeline.

These samples are sent by value to the other thread, therefore the data buffer can be reused immediately after this function returns.

Parameters

- **adsp** – The initialised pipeline.
- **data** – An array of arrays of samples. The length of the array shall be the number of pipeline input channels. Each array contained within shall be contain a frame of samples large enough to pass to the stage that it is connected to.

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

Receive samples from the DSP pipeline.

Parameters

- **adsp** – The initialised pipeline.
- **data** – An array of arrays that will be filled with processed samples from the pipeline. The length of the array shall be the number of pipeline input channels. Each array contained within shall be contain a frame of samples large enough to pass to the stage that it is connected to.

static inline bool **adsp_pipeline_sink_nowait**(*adsp_pipeline_t* *adsp, int32_t **data)

Non-blocking receive from the pipeline. It is risky to use this API in an isochronous application as the sink thread can lose synchronisation with the source thread which can cause the source thread to block.

Parameters

- **adsp** – The initialised pipeline.
- **data** – See adsp_pipeline_sink for details of same named param.

Return values

- **true** – The data buffer has been filled with new values from the pipeline.
- **false** – The pipeline has not produced any more data. The data buffer was untouched.

struct **adsp_pipeline_t**

#include <adsp_pipeline.h> The DSP pipeline.

The generated pipeline will contain an init function that returns a pointer to one of these. It can be used to send data in and out of the pipeline, and also execute control commands.

Public Members

module_instance_t ***modules**

Array of DSP stage states, must be used when calling one of the control functions.

size_t **n_modules**

Number of modules in the *adsp_pipeline_t::modules* array.

Private Members

channel_t ***p_in**

size_t **n_in**

channel_t ***p_out**

size_t **n_out**

channel_t ***p_link**

size_t **n_link**

adsp_mux_t **input_mux**

adsp_mux_t **output_mux**

Module

Defines the generic structs that will hold the state and control configuration for each stage.

Enums

enum **config_rw_state_t**

Control states, used to communicate between DSP and control threads to notify when control needs processing.

Values:

enumerator **config_read_pending**

Control waiting to read the updated config from DSP.

enumerator **config_write_pending**

Config written by control and waiting for DSP to update.

enumerator **config_read_updated**

Stage has succesfully consumed a read command.

enumerator **config_none_pending**

All done. Control and DSP not waiting on anything.

struct **module_control_t**

#include <adsp_module.h> Control related information shared between control thread and DSP.

Public Members

void ***config**

Pointer to a stage-specific config struct which is used by the control thread.

uint32_t **id**

Unique module identifier assigned by the host.

uint32_t **num_control_commands**

The number of control commands for this stage.

uint8_t **module_type**

Identifies the stage type. Each type of stage has a unique identifier.

uint8_t **cmd_id**

Is set to the current command being processed.

config_rw_state_t **config_rw_state**

intptr_t **current_controller**

id of the current control object that requested a read, do not modify.

swlock_t **lock**

lock used by controlling threads to manage access

struct **module_instance_t**

#include <adsp_module.h> The entire state of a stage in the pipeline.

Public Members

void ***state**

Pointer to the module's state memory.

module_control_t **control**

Module's control state.

void ***constants**

Control

The control API for the generated DSP.

These functions can be executed on any thread which is on the same tile as the generated DSP threads.

Enums

enum **adsp_control_status_t**

Control status.

Values:

enumerator **ADSP_CONTROL_SUCCESS**

Command succesfully executed.

enumerator **ADSP_CONTROL_BUSY**

Stage has not yet processed the command, call again.

Functions

void **adsp_controller_init**(*adsp_controller_t* *ctrl, *adsp_pipeline_t* *pipeline)

Create a DSP controller instance for a particular pipeline.

Parameters

- **ctrl** – The controller instance to initialise.
- **pipeline** – The DSP pipeline that will be controlled with this controller.

adsp_control_status_t **adsp_read_module_config**(*adsp_controller_t* *ctrl, *adsp_stage_control_cmd_t* *cmd)

Initiate a read command by passing in an intialised *adsp_stage_control_cmd_t*.

Must be called repeatedly with the same cmd until ADSP_CONTROL_SUCCESS is returned. If the caller abandons the attempt to read before SUCCESS is returned then this will leave the stage in a state where it can never be read from again.

Parameters

- **ctrl** – An instance of *adsp_controller_t* which has been initialised to control the DSP pipeline.
- **cmd** – An initialised *adsp_stage_control_cmd_t*.

Returns

adsp_control_status_t

adsp_control_status_t **adsp_write_module_config**(*adsp_controller_t* *ctrl, *adsp_stage_control_cmd_t* *cmd)

Initiate a write command by passing in an initialised *adsp_stage_control_cmd_t*.

Must be called repeatedly with the same cmd until ADSP_CONTROL_SUCCESS is returned.

Parameters

- **ctrl** – An instance of *adsp_controller_t* which has been initialised to control the DSP pipeline.
- **cmd** – An initialised *adsp_stage_control_cmd_t*.

Returns

adsp_control_status_t

void **adsp_control_xscope_register_probe**()

Default xscope setup function.

Sets up a single xscope probe with name ADSP, type XSCOPE_CONTINUOUS, and datatype XSCOPE_UINT. Should be called within xscope_user_init().

chanend_t **adsp_control_xscope_init**()

Creates an xscope chanend and connects it to the host. Must be called on the same tile as the DSP pipeline.

Returns

chanend_t

adsp_control_status_t **adsp_control_xscope_process**(chanend_t c_xscope, *adsp_controller_t* *ctrl)

Process an xscope chanend containing a control command from the host.

Parameters

- **c_xscope** – A chanend which has been connected to the host.
- **ctrl** – An instance of *adsp_controller_t* which has been initialised to control the DSP pipeline.

Returns

adsp_control_status_t

void **adsp_control_xscope**(*adsp_pipeline_t* *adsp)

Creates an xscope handler thread for ADSP control.

Handles all xscope traffic and calls to *adsp_read_module_config* and *adsp_write_module_config*. If the application already uses xscope, do not call this function; instead, identify host-to-device packets by the ADSP header and pass them to *adsp_control_xscope_process* manually.

Parameters

- **adsp** – The DSP pipeline that will be controlled with this xscope thread.

struct **adsp_stage_control_cmd_t**

#include <adsp_control.h> The command to execute. Specifies which stage, what command and contains the buffer to read from or write to.

Public Members

uint8_t **instance_id**

The ID of the stage to target. Consider setting the label parameter in the pipeline definition to ensure that a usable identifier gets generated for using with control.

uint8_t **cmd_id**

"See the generated cmds.h for the available commands. Make sure to use a command which is supported for the target stage.

uint16_t **payload_len**

Length of the command in bytes.

void *payload

The buffer. Must be set to a valid array of size payload_len before calling the read or write functions.

struct adsp_controller_t

#include <adsp_control.h> Object used to control a DSP pipeline.

As there may be multiple threads attempting to interact with the DSP pipeline at the same time, a separate instance of *adsp_controller_t* must be used by each to ensure that control can proceed safely.

Initialise each instance of *adsp_controller_t* with *adsp_controller_init*.

Private Members

module_instance_t *modules

size_t num_modules

Control Helper Functions

Most DSP Stages have fixed point control parameters. To aid conversion from typical tuning units (e.g. decibels) to the correct fixed point format, the helper functions below have been provided.

Biquad helpers

Functions

biquad_slew_t adsp_biquad_slew_init(q2_30 target_coeffs[8], left_shift_t lsh, left_shift_t slew_shift)

Initialise a slewing biquad filter object. This sets the active filter coefficients to the target value.

Parameters

- target_coeffs – Filter coefficients
- lsh – Filter left shift compensation value
- slew_shift – Shift value used in the exponential slew

Returns

biquad_slew_t Slewing biquad object

void adsp_biquad_slew_update_coeffs(*biquad_slew_t* *slew_state, int32_t **states, int32_t channels, q2_30 target_coeffs[8], left_shift_t lsh)

Update the target coefficients in a slewing biquad filter object. This updates the target coefficients, and manages any change in filter coefficient left shift. This may require shifting the active filter coefficients and states.

Parameters

- slew_state – Slewing biquad state object
- states – Filter state for each biquad channel
- channels – Number of channels in states
- target_coeffs – New filter coefficients
- lsh – New filter left shift compensation value

left_shift_t **adsp_design_biquad_bypass**(q2_30 coeffs[5])

Design biquad filter bypass This function creeates a bypass biquad filter. Only the b0 coefficient is set.

Parameters

- **coeffs** – Bypass filter coefficients

Returns

left_shift_t Left shift compensation value

left_shift_t **adsp_design_biquad_mute**(q2_30 coeffs[5])

Design mute biquad filter This function creates a mute biquad filter. All the coefficients are 0.

Parameters

- **coeffs** – Mute filter coefficients

Returns

left_shift_t Left shift compensation value

left_shift_t **adsp_design_biquad_gain**(q2_30 coeffs[5], const float gain_db)

Design gain biquad filter This function creates a biquad filter with a specified gain.

Parameters

- **coeffs** – Gain filter coefficients
- **gain_db** – Gain in dB

Returns

left_shift_t Left shift compensation value

left_shift_t **adsp_design_biquad_lowpass**(q2_30 coeffs[5], const float fc, const float fs, const float filter_Q)

Design lowpass biquad filter This function creates a biquad filter with a lowpass response `fc` must be less than `fs/2`, otherwise it will be saturated to `fs/2`.

Parameters

- **coeffs** – Lowpass filter coefficients
- **fc** – Cutoff frequency
- **fs** – Sampling frequency
- **filter_Q** – Filter Q

Returns

left_shift_t Left shift compensation value

left_shift_t **adsp_design_biquad_highpass**(q2_30 coeffs[5], const float fc, const float fs, const float filter_Q)

Design highpass biquad filter This function creates a biquad filter with a highpass response `fc` must be less than `fs/2`, otherwise it will be saturated to `fs/2`.

Parameters

- **coeffs** – Highpass filter coefficients
- **fc** – Cutoff frequency
- **fs** – Sampling frequency
- **filter_Q** – Filter Q

Returns

left_shift_t Left shift compensation value

left_shift_t **adsp_design_biquad_bandpass**(q2_30 coeffs[5], const float fc, const float fs, const float bandwidth)

Design bandpass biquad filter This function creates a biquad filter with a bandpass response `fc` must be less than `fs/2`, otherwise it will be saturated to `fs/2`.

Parameters

- **coeffs** – Bandpass filter coefficients
- **fc** – Central frequency
- **fs** – Sampling frequency
- **bandwidth** – Bandwidth

Returns

left_shift_t Left shift compensation value

left_shift_t **adsp_design_biquad_bandstop**(q2_30 coeffs[5], const float fc, const float fs, const float bandwidth)

Design bandstop biquad filter This function creates a biquad filter with a bandstop response `fc` must be less than `fs/2`, otherwise it will be saturated to `fs/2`.

Parameters

- **coeffs** – Bandstop filter coefficients
- **fc** – Central frequency
- **fs** – Sampling frequency
- **bandwidth** – Bandwidth

Returns

left_shift_t Left shift compensation value

left_shift_t **adsp_design_biquad_notch**(q2_30 coeffs[5], const float fc, const float fs, const float filter_Q)

Design notch biquad filter This function creates a biquad filter with an notch response `fc` must be less than `fs/2`, otherwise it will be saturated to `fs/2`.

Parameters

- **coeffs** – Notch filter coefficients
- **fc** – Central frequency
- **fs** – Sampling frequency
- **filter_Q** – Filter Q

Returns

left_shift_t Left shift compensation value

left_shift_t **adsp_design_biquad_allpass**(q2_30 coeffs[5], const float fc, const float fs, const float filter_Q)

Design allpass biquad filter This function creates a biquad filter with an allpass response `fc` must be less than `fs/2`, otherwise it will be saturated to `fs/2`.

Parameters

- **coeffs** – Allpass filter coefficients
- **fc** – Central frequency
- **fs** – Sampling frequency
- **filter_Q** – Filter Q

Returns

left_shift_t Left shift compensation value

left_shift_t **adsp_design_biquad_peaking**(q2_30 coeffs[5], const float fc, const float fs, const float filter_Q, const float gain_db)

Design peaking biquad filter This function creates a biquad filter with a peaking response `fc` must be less than `fs/2`, otherwise it will be saturated to `fs/2`.

The gain must be less than 18 dB, otherwise the coefficients may overflow. If the gain is greater than 18 dB, it is saturated to that value.

Parameters

- **coeffs** – Peaking filter coefficients
- **fc** – Central frequency
- **fs** – Sampling frequency
- **filter_Q** – Filter Q
- **gain_db** – Gain in dB

Returns

left_shift_t Left shift compensation value

left_shift_t **adsp_design_biquad_const_q**(q2_30 coeffs[5], const float fc, const float fs, const float filter_Q, const float gain_db)

Design constant Q peaking biquad filter This function creates a biquad filter with a constant Q peaking response.

Constant Q means that the bandwidth of the filter remains constant as the gain varies. It is commonly used for graphic equalisers. `fc` must be less than `fs/2`, otherwise it will be saturated to `fs/2`.

The gain must be less than 18 dB, otherwise the coefficients may overflow. If the gain is greater than 18 dB, it is saturated to that value.

Parameters

- **coeffs** – Constant Q filter coefficients
- **fc** – Central frequency
- **fs** – Sampling frequency
- **filter_Q** – Filter Q
- **gain_db** – Gain in dB

Returns

left_shift_t Left shift compensation value

left_shift_t **adsp_design_biquad_lowshelf**(q2_30 coeffs[5], const float fc, const float fs, const float filter_Q, const float gain_db)

Design lowshelf biquad filter This function creates a biquad filter with a lowshelf response.

The Q factor is defined in a similar way to standard low pass, i.e. Q > 0.707 will yield peakiness (where the shelf response does not monotonically change). The level change at f will be boost_db/2. `fc` must be less than `fs/2`, otherwise it will be saturated to `fs/2`.

The gain must be less than 12 dB, otherwise the coefficients may overflow. If the gain is greater than 12 dB, it is saturated to that value.

Parameters

- **coeffs** – Lowshelf filter coefficients
- **fc** – Cutoff frequency
- **fs** – Sampling frequency

- **filter_Q** – Filter Q
- **gain_db** – Gain in dB

Returns

left_shift_t Left shift compensation value

left_shift_t **adsp_design_biquad_highshelf**(q2_30 coeffs[5], const float fc, const float fs, const float filter_Q, const float gain_db)

Design highshelf biquad filter This function creates a biquad filter with a highshelf response.

The Q factor is defined in a similar way to standard high pass, i.e. Q > 0.707 will yield peakiness. The level change at f will be boost_db/2. `fc` must be less than `fs/2`, otherwise it will be saturated to `fs/2`.

The gain must be less than 12 dB, otherwise the coefficients may overflow. If the gain is greater than 12 dB, it is saturated to that value.

Parameters

- **coeffs** – Highshelf filter coefficients
- **fc** – Cutoff frequency
- **fs** – Sampling frequency
- **filter_Q** – Filter Q
- **gain_db** – Gain in dB

Returns

left_shift_t Left shift compensation value

left_shift_t **adsp_design_biquad_linkwitz**(q2_30 coeffs[5], const float f0, const float fs, const float q0, const float fp, const float qp)

Design Linkwitz transform biquad filter This function creates a biquad filter with a Linkwitz transform response.

The Linkwitz Transform is commonly used to change the low frequency roll off slope of a loudspeaker. When applied to a loudspeaker, it will change the cutoff frequency from f0 to fp, and the quality factor from q0 to qp. `f0` and `fp` must be less than `fs/2`, otherwise they will be saturated to `fs/2`.

Parameters

- **coeffs** – Linkwitz filter coefficients
- **f0** – Original cutoff frequency
- **fs** – Sampling frequency
- **q0** – Original quality factor at f0
- **fp** – Target cutoff frequency
- **qp** – Target quality factor of the filter

Returns

left_shift_t Left shift compensation value

left_shift_t **adsp_apply_biquad_gain**(q2_30 coeffs[5], left_shift_t b_sh, float gain_db)

Modify the gain of a set of biquad filter coefficients.

Parameters

- **coeffs** – Existing filter coefficients
- **b_sh** – Existing left shift compensation value
- **gain_db** – Gain in dB

Returns
left_shift_t Left shift compensation value

DRC helpers

static inline int32_t **calc_alpha**(float fs, float time)

Convert an attack or release time in seconds to an EWM alpha value as a fixed point int32 number in Q_alpha format. If the desired time is too large or small to be represented in the fixed point format, it is saturated.

Parameters

- **fs** – sampling frequency in Hz
- **time** – attack/release time in seconds

Returns
int32_t attack/release alpha as an int32_t

static inline int32_t **calculate_peak_threshold**(float level_db)

Convert a peak compressor/limiter/expander threshold in decibels to an int32 fixed point gain in Q_SIG Q format. If the threshold is higher than representable in the fixed point format, it is saturated. The minimum threshold returned by this function is 1.

Parameters

- **level_db** – the desired threshold in decibels

Returns
int32_t the threshold as a fixed point integer.

static inline int32_t **calculate_rms_threshold**(float level_db)

Convert an RMS² compressor/limiter/expander threshold in decibels to an int32 fixed point gain in Q_SIG Q format. If the threshold is higher than representable in the fixed point format, it is saturated. The minimum threshold returned by this function is 1.

Parameters

- **level_db** – the desired threshold in decibels

Returns
int32_t the threshold as a fixed point integer.

static inline float **rms_compressor_slope_from_ratio**(float ratio)

Convert a compressor ratio to the slope, where the slope is defined as (1 - 1 / ratio) / 2.0. The division by 2 compensates for the RMS envelope detector returning the RMS². The ratio must be greater than 1, if it is not the ratio is set to 1.

Parameters

- **ratio** – the desired compressor ratio

Returns
float slope of the compressor

static inline float **peak_expander_slope_from_ratio**(float ratio)

Convert an expander ratio to the slope, where the slope is defined as (1 - ratio). The ratio must be greater than 1, if it is not the ratio is set to 1.

Parameters

- **ratio** – the desired expander ratio

Returns
float slope of the expander

static inline float **qxx_to_db**(int32_t level, int q_format)

Convert a fixed point int32 number in the given Q format to a value in decibels.

Parameters

- **level** – Level in the fixed point format specified by q_format
- **q_format** – Q format of the input

Returns
float level in dB for the signal

static inline float **qxx_to_db_pow**(int32_t level, int q_format)

Convert a fixed point int32 number in the given Q format to a value in decibels, when the input level is power.

Parameters

- **level** – Power level in the fixed point format specified by q_format
- **q_format** – Q format of the input

Returns
float level in dB for the signal

Functions

env_detector_t **adsp_env_detector_init**(float fs, float attack_t, float release_t)

Initialise an envelope detector object.

Note: Detect time is optional. If specified, attack and release times will be equal.

Parameters

- **fs** – Sampling frequency
- **attack_t** – Attack time in seconds
- **release_t** – Release time in seconds

Returns
env_detector_t Initialised envelope detector object

limiter_t **adsp_limiter_peak_init**(float fs, float threshold_db, float attack_t, float release_t)

Initialise a (hard) limiter peak object.

Parameters

- **fs** – Sampling frequency
- **threshold_db** – Threshold in dB
- **attack_t** – Attack time in seconds
- **release_t** – Release time in seconds

Returns
limiter_t Initialised limiter object

limiter_t **adsp_limiter_rms_init**(float fs, float threshold_db, float attack_t, float release_t)

Initialise an RMS limiter object.

Parameters

- **fs** – Sampling frequency
- **threshold_db** – Threshold in dB
- **attack_t** – Attack time in seconds
- **release_t** – Release time in seconds

Returns

limiter_t Initialised limiter object

noise_gate_t **adsp_noise_gate_init**(float fs, float threshold_db, float attack_t, float release_t)

Initialise a noise gate object.

Parameters

- **fs** – Sampling frequency
- **threshold_db** – Threshold in dB
- **attack_t** – Attack time in seconds
- **release_t** – Release time in seconds

Returns

noise_gate_t Initialised noise gate object

noise_suppressor_expander_t **adsp_noise_suppressor_expander_init**(float fs, float threshold_db, float attack_t, float release_t, float ratio)

Initialise a noise suppressor (expander) object.

Parameters

- **fs** – Sampling frequency
- **threshold_db** – Threshold in dB
- **attack_t** – Attack time in seconds
- **release_t** – Release time in seconds
- **ratio** – Noise suppression ratio

Returns

noise_suppressor_expander_t Initialised noise suppressor (expander) object

void **adsp_noise_suppressor_expander_set_th**(*noise_suppressor_expander_t* *nse, int32_t new_th)

Set the threshold of a noise suppressor (expander)

Parameters

- **nse** – Noise suppressor (Expander) object
- **new_th** – New threshold in Q_SIG

compressor_t **adsp_compressor_rms_init**(float fs, float threshold_db, float attack_t, float release_t, float ratio)

Initialise a compressor object.

Parameters

- **fs** – Sampling frequency
- **threshold_db** – Threshold in dB
- **attack_t** – Attack time in seconds
- **release_t** – Release time in seconds
- **ratio** – Compression ratio

Returns

compressor_t Initialised compressor object

compressor_stereo_t **adsp_compressor_rms_stereo_init**(float fs, float threshold_db, float attack_t, float release_t, float ratio)

Initialise a stereo compressor object.

Parameters

- **fs** – Sampling frequency
- **threshold_db** – Threshold in dB
- **attack_t** – Attack time in seconds
- **release_t** – Release time in seconds
- **ratio** – Compression ratio

Returns

compressor_stereo_t Initialised stereo compressor object

Graphic EQ helpers

q2_30 ***adsp_graphic_eq_10b_init**(float fs)

Generate the filter coefficients for a 10-band graphic equaliser.

Returns a pointer to a set of bandpass filters that can use used by `adsp_graphic_eq_10b`. Sample rates between 16kHz and 192 kHz are supported.

Parameters

- **fs** – Sample rate of the graphic eq

Returns

int32_t* Pointer to the filter coefficients

static inline int32_t **geq_db_to_gain**(float level_db)

Convert a graphic equaliser gain in decibels to a fixed point int32 number in Q31 format. The input level is shifted by -12 dB. This means that all the graphic EQ sliders can be set to +12 without clipping, at the cost of -12dB level when the slider gains are set to 0dB.

Parameters

- **level_db** – Level in db

Returns

int32_t level_db as an int32_t

Reverb helpers

Functions

static inline int32_t **adsp_reverb_float2int**(float x)

Convert a floating point value to the Q_RVR format, saturate out of range values. Accepted range is 0 to 1.

Parameters

- **x** – A floating point number, will be capped to [0, 1]

Returns

Q_RVR int32_t value

static inline int32_t **adsp_reverb_db2int**(float db)

Convert a floating point gain in decibels into a linear Q_RVR value for use in controlling the reverb gains.

Parameters

- **db** – Floating point value in dB, values above 0 will be clipped.

Returns

Q_RVR fixed point linear gain.

static inline int32_t **adsp_reverb_calculate_damping**(float damping)

Convert a user damping value into a Q_RVR fixed point value suitable for passing to a reverb.

Parameters

- **damping** – The chose value of damping.

Returns

Damping as a Q_RVR fixed point integer, clipped to the accepted range.

static inline int32_t **adsp_reverb_calculate_feedback**(float decay)

Calculate a Q_RVR feedback value for a given decay. Use to calculate the feedback parameter in reverb_room.

Parameters

- **decay** – The desired decay value.

Returns

Calculated feedback as a Q_RVR fixed point integer.

static inline int32_t **adsp_reverb_room_calc_gain**(float gain_db)

Calculate the reverb gain in linear scale.

Will convert a gain in dB to a linear scale in Q_RVR format. To be used for converting wet and dry gains for the room_reverb.

Parameters

- **gain_db** – Gain in dB

Returns

int32_t Linear gain in a Q_RVR format

static inline void **adsp_reverb_wet_dry_mix**(int32_t gains[2], float mix)

Calculate the wet and dry gains according to the mix amount.

When the mix is set to 0, only the dry signal will be output. The wet gain will be 0 and the dry gain will be max. When the mix is set to 1, only they wet signal will be

output. The wet gain is max, the dry gain will be 0. In order to maintain a consistent signal level across all mix values, the signals are panned with a -4.5 dB panning law.

Parameters

- **gains** – Output gains: [0] - Dry; [1] - Wet
- **mix** – Mix applied from 0 to 1

reverb_room_t **adsp_reverb_room_init**(float fs, float max_room_size, float room_size, float decay, float damping, float wet_gain, float dry_gain, float pregain, float max_predelay, float predelay, void *reverb_heap)

Initialise a reverb room object A room reverb effect based on Freeverb by Jezar at Dreampoint.

Parameters

- **fs** – Sampling frequency
- **max_room_size** – Maximum room size of delay filters
- **room_size** – Room size compared to the maximum room size [0, 1]
- **decay** – Length of the reverb tail [0, 1]
- **damping** – High frequency attenuation
- **wet_gain** – Wet gain in dB
- **dry_gain** – Dry gain in dB
- **pregain** – Linear pre-gain
- **max_predelay** – Maximum size of the predelay buffer in ms
- **predelay** – Initial predelay in ms
- **reverb_heap** – Pointer to heap to allocate reverb memory

Returns

reverb_room_t Initialised reverb room object

void **adsp_reverb_room_st_calc_wet_gains**(int32_t wet_gains[2], float wet_gain, float width)

Calculate the stereo wet gains of the stereo reverb room.

Parameters

- **wet_gains** – Output linear wet_1 and wet_2 gains in Q_RVR
- **wet_gain** – Input wet gain in dB
- **width** – Stereo separation of the room [0, 1]

void **adsp_reverb_st_wet_dry_mix**(int32_t gains[3], float mix, float width)

Calculate the stereo wet and dry gains according to the mix amount.

When the mix is set to 0, only the dry signal will be output. The wet gain will be 0 and the dry gain will be max. When the mix is set to 1, only they wet signal will be output. The wet gain is max, the dry gain will be 0. In order to maintain a consistent signal level across all mix values, the signals are panned with a -4.5 dB panning law. The width controls the mixing between the left and right wet channels

Parameters

- **gains** – Output gains: [0] - Dry; [1] - Wet_1; [2] - Wet_2
- **mix** – Mix applied from 0 to 1
- **width** – Stereo separation of the room [0, 1]

reverb_room_st_t **adsp_reverb_room_st_init**(float fs, float max_room_size, float room_size, float decay, float damping, float width, float wet_gain, float dry_gain, float pregain, float max_predelay, float predelay, void *reverb_heap)

Initialise a stereo reverb room object A room reverb effect based on Freeverb by Jezar at Dreampoint.

Parameters

- **fs** – Sampling frequency
- **max_room_size** – Maximum room size of delay filters
- **room_size** – Room size compared to the maximum room size [0, 1]
- **decay** – Length of the reverb tail [0, 1]
- **damping** – High frequency attenuation
- **width** – Stereo separation of the room [0, 1]
- **wet_gain** – Wet gain in dB
- **dry_gain** – Dry gain in dB
- **pregain** – Linear pre-gain
- **max_predelay** – Maximum size of the predelay buffer in ms
- **predelay** – Initial predelay in ms
- **reverb_heap** – Pointer to heap to allocate reverb memory

Returns

reverb_room_st_t Initialised stereo reverb room object

Functions

static inline int32_t **adsp_reverb_plate_calc_late_diffusion**(float late_diffusion)

Convert a user late diffusion value into a Q_RVP fixed point value suitable for passing to a reverb.

Parameters

- **late_diffusion** – The chose value of late diffusion.

Returns

Late diffusion as a Q_RVP fixed point integer, clipped to the accepted range.

static inline int32_t **adsp_reverb_plate_calc_damping**(float damping)

Convert a user damping value into a Q_RVP fixed point value suitable for passing to a reverb.

Parameters

- **damping** – The chose value of damping.

Returns

Damping as a Q_RVP fixed point integer, clipped to the accepted range.

static inline int32_t **adsp_reverb_plate_calc_bandwidth**(float bandwidth, float fs)

Convert a user bandwidth value in Hz into a Q_RVP fixed point value suitable for passing to a reverb.

Parameters

- **bandwidth** – The chose value of bandwidth.
- **fs** – The sampling frequency in Hz

Returns

Bandwidth as a Q_RVP fixed point integer, clipped to the accepted range.

reverb_plate_t **adsp_reverb_plate_init**(float fs, float decay, float damping, float bandwidth, float early_diffusion, float late_diffusion, float width, float wet_gain, float dry_gain, float pregain, float max_predelay, float predelay, void *reverb_heap)

Initialise a reverb plate object.

Parameters

- **fs** – Sampling frequency
- **decay** – Length of the reverb tail [0, 1]
- **damping** – High frequency attenuation
- **bandwidth** – Pre lowpass
- **early_diffusion** – Early diffusion
- **late_diffusion** – Late diffusion
- **width** – Stereo separation of the room [0, 1]
- **wet_gain** – Wet gain in dB
- **dry_gain** – Dry gain in dB
- **pregain** – Linear pre-gain
- **max_predelay** – Maximum size of the predelay buffer in ms
- **predelay** – Initial predelay in ms
- **reverb_heap** – Pointer to heap to allocate reverb memory

Returns

reverb_plate_t Initialised reverb plate object

Signal chain helpers

Enums

enum **time_units_t**

Enum for different time units.

Values:

enumerator **SAMPLES**

Time in samples

enumerator **MILLISECONDS**

Time in milliseconds

enumerator **SECONDS**

Time in seconds

Functions

int32_t **adsp_dB_to_gain**(float dB_gain)

Convert dB gain to linear gain.

Note: With the current Q_GAIN format, the maximum gain is +24 dB, dB_gain will be saturated to this value

Note: Passing -INFINITY to this function will give a linear gain of 0.

Parameters

- **dB_gain** – Gain in dB

Returns

int32_t Linear gain in Q_GAIN format

gain_slew_t **adsp_slew_gain_init**(int32_t init_gain, int32_t slew_shift)

Initialise a slewing gain object.

The slew shift will determine the speed of the volume change. A list of the first 10 slew shifts is shown below:

1 -> 0.03 ms, 2 -> 0.07 ms, 3 -> 0.16 ms, 4 -> 0.32 ms, 5 -> 0.66 ms, 6 -> 1.32 ms, 7 -> 2.66 ms, 8 -> 5.32 ms, 9 -> 10.66 ms, 10 -> 21.32 ms.

Parameters

- **init_gain** – Initial gain
- **slew_shift** – Shift value used in the exponential slew

Returns

gain_slew_t The slewing gain object.

volume_control_t **adsp_volume_control_init**(float gain_dB, int32_t slew_shift, uint8_t mute_state)

Initialise volume control object. The slew shift will determine the speed of the volume change. A list of the first 10 slew shifts is shown below:

1 -> 0.03 ms, 2 -> 0.07 ms, 3 -> 0.16 ms, 4 -> 0.32 ms, 5 -> 0.66 ms, 6 -> 1.32 ms, 7 -> 2.66 ms, 8 -> 5.32 ms, 9 -> 10.66 ms, 10 -> 21.32 ms.

Parameters

- **gain_dB** – Target gain in dB
- **slew_shift** – Shift value used in the exponential slew
- **mute_state** – Initial mute state

Returns

volume_control_t Volume control state object

delay_t **adsp_delay_init**(float fs, float max_delay, float starting_delay, *time_units_t* units, void *delay_heap)

Initialise a delay object.

Parameters

- **fs** – Sampling frequency
- **max_delay** – Maximum delay in specified units
- **starting_delay** – Initial delay in specified units

- **units** – Time units (SAMPLES, MILLISECONDS, SECONDS). If an invalid unit is passed, SAMPLES is used.
- **delay_heap** – Pointer to the allocated delay memory

Returns

delay_t Delay state object

void **adsp_set_delay**(*delay_t* *delay, float delay_time, *time_units_t* units)

Set the delay of a delay object. Will set the delay to the new value, saturating to the maximum delay.

Parameters

- **delay** – Delay object
- **delay_time** – New delay time in specified units
- **units** – Time units (SAMPLES, MILLISECONDS, SECONDS). If an invalid unit is passed, SAMPLES is used.

uint32_t **time_to_samples**(float fs, float time, *time_units_t* units)

Convert a time in seconds/milliseconds/samples to samples for a given sampling frequency.

Parameters

- **fs** – Sampling frequency
- **time** – New delay time in specified units
- **units** – Time units (SAMPLES, MILLISECONDS, SECONDS) . If an invalid unit is passed, SAMPLES is used.

Returns

uint32_t Time in samples

switch_slew_t **adsp_switch_slew_init**(float fs, int32_t init_position)

Initialise a slewing switch object.

Parameters

- **fs** – Sampling frequency, used to calculate the step size.
- **init_position** – Starting position of the switch.

Returns

switch_slew_t The slewing switch object.

void **adsp_switch_slew_move**(*switch_slew_t* *switch_slew, int32_t new_position)

Move the position of the switch. This sets the state of the switch for slewing on subsequent samples.

Parameters

- **switch_slew** – Slewing switch state object.
- **new_position** – The desired input channel to switch to.

void **adsp_crossfader_mix**(int32_t gains[2], float mix)

Calculate the gains for a crossfader according to the mix amount.

When the mix is set to 0, only the first signal will be output. gains[0] will be max and gains[1] will be 0. When the mix is set to 1, only they second signal will be output. gains[0] will be 0 and gains[1] will be max. In order to maintain a consistent signal level across all mix values, when the mix is set to 0.5, each channel has a gain of -4.5 dB.

Parameters

- **gains** – Output gains
- **mix** – Mix applied from 0 to 1

5.4 Pipeline Design API

This page describes the C and Python APIs that will be needed when using the pipeline design utility.

When designing a pipeline first create an instance of **Pipeline**, add threads to it with **Pipeline.add_thread()**. Then add DSP stages such as **Biquad** using **CompositeStage.stage()**. The pipeline can be visualised in a Jupyter Notebook using **Pipeline.draw()** and the xcore source code for the pipeline can be generated using **generate_dsp_main()**.

audio_dsp.design.build_utils

Utility functions for building and running the application within the Jupyter notebook.

class audio_dsp.design.build_utils.**XCommonCMakeHelper**(*source_dir: str | pathlib.Path | None = None, build_dir: str | pathlib.Path | None = None, bin_dir: str | pathlib.Path | None = None, project_name: str | None = None, config_name: str | None = None*)

This class packages a set of helper utilities for configuring, building, and running xcore applications using xcommon-cmake within Python.

Parameters

source_dir
[str | pathlib.Path | None] Specify a source directory for this build, passed as the -S parameter to CMake. If None passed or unspecified, defaults to the current working directory.

build_dir
[str | pathlib.Path | None] Specify a build directory for this build, passed as the -B parameter to CMake. If None passed or unspecified, defaults to "build" within the current working directory.

bin_dir
[str | pathlib.Path | None] Specify a binary output directory for this build. This should match what is configured to be the output directory from "cmake –build" within the application. If None passed or unspecified, defaults to "bin" within the current working directory.

project_name
[str | None] The name of the project() specified in the project's CMakeLists.txt. If None or unspecified, defaults to the name of the current working directory (so if in /app_example_name/, the project name is assumed to be app_example_name).

config_name
[str | None] The name of the configuration to use from the project's CMakeLists.txt. If None or unspecified, defaults to nothing - therefore the –target option to CMake will be just the project name, and the output binary will be assumed to be "<current working directory>/<bin_dir>/<project_name>.xe". If specified, the –target option to CMake will be "<project name>_<config name>", and the output binary will be assumed to be "<current working directory>/<bin_dir>/<config_name>/<project name>_<config name>.xe".

`build()` → int

Invoke CMake's build with the options specified in this class instance. Invokation will be of the form `cmake --build <build_dir> --target <target_name>`, where the target name is constructed as per this class' docstring.

Returns

returncode
Return code from the invokation of CMake. 0 if success.

`configure()` → int | None

Invoke CMake with the options specified in this class instance. Invokation will be of the form `cmake -S <source_dir> -B <build_dir>`. On first run, the invokation will also contain `-G <generator>`, where "generator" will be either "Ninja" if Ninja is present on the current system or "Unix Makefiles" if it is not.

Returns

returncode
Return code from the invokation of CMake. 0 if success.

`configure_build_run(`*xscope: bool = True*`)` → None

Run, in order, this class' .configure(), .build(), and .run() methods. If any return

code from any of the three is nonzero, returns early. Otherwise, sleeps for 5 seconds after the .run() stage and prints "Done!".

Parameters

xscope
[bool] Passed directly to the call to .run(); determines whether to start an xscope server or not.

`run`(*xscope: bool = True*, *hostname: str = 'localhost'*, *port: str = '12345'*) → int

Invoke xrun or xgdb with the options specified in this class instance.

If xscope is True the invocation will be of the form:

```
xgdb -q --return-child-result --batch
-ex "connect --xscope-port <hostname>:<port> --xscope"
-ex "load"
-ex "continue"
<binary>
```

whereas if xscope if False the invocation will be of the form:

```
xrun <binary>
```

where the path to the binary is constructed as per this class' docstring.

Parameters

xscope
[bool] Specify whether to set up an xscope server or not.

hostname
[str] Hostname to use for the xscope server if xscope is True

port
[str] Port to use for the xscope server if xscope is True

Returns

returncode
Return code from the invokation of xrun or xgdb. 0 if success.

audio_dsp.design.composite_stage

Contains the higher order stage class CompositeStage.

`class audio_dsp.design.composite_stage.CompositeStage`(*graph: Graph*, *name: str = ''*)

This is a higher order stage.

Contains stages as well as other composite stages. A thread will be a composite stage. Composite stages allow:

- drawing the detail with graphviz
- process
- frequency response

TODO: - Process method on the composite stage will need to know its inputs and the order of the inputs (which input index corresponds to each input edge). However a CompositeStage doesn't know all of its inputs when it is created.

Parameters

graph
[audio_dsp.graph.Graph] instance of graph that all stages in this composite will be added to.

name
[str] Name of this instance to use when drawing the pipeline, defaults to class name.

add_to_dot(*dot*)

Recursively adds composite stages to a dot diagram which is being constructed. Does not add the edges.

Parameters

dot
[graphviz.Diagraph] dot instance to add edges to.

composite_stage(*name: str = ''*) → CompositeStage

Create a new composite stage that will be a included in the current composite. The new stage can have stages added to it dynamically.

contains_stage(*stage: Stage*) → bool

Recursively search self for the stage.

Returns

bool
True if this composite contains the stage else False

draw()

Draws the stages and edges present in this instance of a composite stage.

get_all_stages() → list[audio_dsp.design.stage.Stage]

Get a flat list of all stages contained within this composite stage and the composite stages within.

Returns

list of stages.

property o: StageOutputList

Outputs of this composite.

Dynamically computed by searching the graph for edges which originate in this composite and whose destination is outside this composite. Order not currently specified.

process(*data*)

Execute the stages in this composite on the host.

> **Warning:** Not implemented.

stage(*stage_type: Type[_StageOrComposite], inputs: StageOutputList, label: str | None = None, **kwargs*) → _StageOrComposite

Create a new stage or composite stage and register it with this composite stage.

Parameters

stage_type
Must be a subclass of Stage or CompositeStage

inputs
Edges of the pipeline that will be connected to the newly created stage.

kwargs
[dict] Additional args are forwarded to the stages constructors (__init__)

Returns

stage_type
Newly created stage or composite stage.

`stages`(*stage_types: list[Type[_StageOrComposite]]*, *inputs: StageOutputList*) → list[_StageOrComposite]

Iterate through the provided stages and connect them linearly.
Returns a list of the created instances.

audio_dsp.design.graph

Basic data structures for managing the pipeline graph.

`class audio_dsp.design.graph.Edge`

Graph node.

Attributes

id
[uuid.UUID4] A unique identifier for this node.

source
[Node | None]

dest
[Node | None] source and dest are the graph nodes that this edge connects between.

`set_dest`(*node: Node*)

Set the dest node of this edge.

Parameters

node
The instance to set as the dest.

`set_source`(*node: Node*)

Set the source node of this edge.

Parameters

node
The instance to set as the source.

`class audio_dsp.design.graph.Graph`

A container of nodes and edges.

Attributes

nodes
A list of the nodes in this graph.

edges
A list of the edges in this graph.

add_edge(*edge*) → None

Append an edge to this graph.

add_node(*node: NodeSubClass*) → None

Append a node to this graph.

The node's index attribute is set here and therefore the node may not coexist in multiple graphs.

get_dependency_dict() → dict[NodeSubClass, set[NodeSubClass]]

Return a mapping of nodes to their dependencies ready for use with the graphlib utilities.

get_view(*nodes: list[NodeSubClass]*) → Graph[NodeSubClass]

Get a filtered view of the graph, including only the provided nodes and the edges which connect to them.

lock()

Lock the graph. Adding nodes or edges to a locked graph will cause a runtime exception. The graph is locked once the pipeline checksum is computed.

sort() → tuple[NodeSubClass, ...]

Sort the nodes in the graph based on the order they should be executed. This is determined by looking at the edges in the graph and resolving the order.

Returns

tuple[Node]
Ordered list of nodes

class audio_dsp.design.graph.**Node**

Graph node.

Attributes

id
[uuid.UUID4] A unique identifier for this node.

index
[None | int] node index in the graph. This is set by Graph when it is added to the graph.

audio_dsp.design.parse_config

Script for use at build time to generate header files.

Use as:

```
python -m audio_dsp.design.parse_config -c CONFIG_DIR -o OUTPUT_DIR
```

audio_dsp.design.parse_config.**main**(*args*)

Use the mako templates to build the autogenerated files.

audio_dsp.design.pipeline

Top level pipeline design class and code generation functions.

class audio_dsp.design.pipeline.**Pipeline**(*n_in*, *identifier='auto'*, *frame_size=1*, *fs=48000*, *generate_xscope_task=False*)

Top level class which is a container for a list of threads that are connected in series.

Parameters

n_in
[int] Number of input channels into the pipeline

identifier: string
Unique identifier for this pipeline. This identifier will be included in the generated header file name (as "adsp_generated_<identifier>.h"), the generated source file name (as "adsp_generated_<identifier>.c"), and the pipeline's generated initialisation and main functions (as "adsp_<identifier>_pipeline_init" and "adsp_<identifier>_pipeline_main")

frame_size
[int] Size of the input frame of all input channels

fs
[int] Sample rate of the input channels

generate_xscope_task
[bool] Determines whether the generated pipeline automatically instantiates a task to handle tuning over xscope. False by default. If False, the application code will need to explicitly call the "adsp_control_xscope_*" functions defined in adsp_control.h in order to handle tuning over xscope, such as that undertaken by the XScopeTransport() class.

Attributes

i
[list(StageOutput)] The inputs to the pipeline should be passed as the inputs to the first stages in the pipeline

threads
[list(Thread)] List of all the threads in the pipeline

pipeline_stage
[PipelineStage | None] Stage corresponding to the pipeline. Needed for handling pipeline level control commands

add_pipeline_stage(*thread*)

Add a PipelineStage stage for the pipeline.

static begin(*n_in*, *identifier='auto'*, *frame_size=1*, *fs=48000*, *generate_xscope_task=False*)

Create a new Pipeline and get the attributes required for design.

Returns

Pipeline, Thread, StageOutputList
The pipeline instance, the initial thread and the pipeline input edges.

draw(*path: pathlib.Path | None = None*)

Render a dot diagram of this pipeline.
If *path* is not none then the image will be saved to the named file instead of drawing to the jupyter notebook.

executor() → PipelineExecutor

Create an executor instance which can be used to simulate the pipeline.

generate_pipeline_hash(*threads: list*, *edges: list*)

Generate a hash unique to the pipeline and save it in the 'checksum' control field of the pipeline stage.

Parameters

"threads": list of [[(stage index, stage type name), ...], ...] for all threads in the pipeline
"edges": list of [[[source stage, source index], [dest stage, dest index]], ...] for all edges in the pipeline

next_thread() → None

Update the thread which stages will be added to.

This will always create a new thread.

resolve_pipeline()

Generate a dictionary with all of the information about the thread. Actual stage instances not included.

Returns

dict
'identifier': string identifier for the pipeline "threads": list of [[(stage index, stage type name, stage memory use), ...], ...] for all threads "edges": list of [[[source stage, source index], [dest stage, dest index]], ...] for all edges "configs": list of dicts containing stage config for each stage. "modules": list of stage yaml configs for all types of stage that are present "labels": dictionary {label: instance_id} defining mapping between the user defined stage labels and the index of the stage "xscope": bool indicating whether or not to create an xscope task for control

set_outputs(*output_edges: StageOutputList*)

Set the pipeline outputs, configures the output channel index.

Parameters

output_edges
[Iterable(None | StageOutput)] configure the output channels and their indices. Outputs of the pipeline will be in the same indices as the input to this function. To have an empty output index, pass in None.

stage(*stage_type: Type[audio_dsp.design.stage.Stage | audio_dsp.design.composite_stage.CompositeStage]*, *inputs: StageOutputList*, *label: str | None = None*, ***kwargs*) → StageOutputList

Add a new stage to the pipeline.

Parameters

stage_type
The type of stage to add.
inputs
A StageOutputList containing edges in this pipeline.
label
An optional label that can be used for tuning and will also be converted into a macro in the generated pipeline. Label must be set if tuning or run time control is required for this stage.

property stages

Flattened list of all the stages in the pipeline.

validate()

TODO validate pipeline assumptions.

- Thread connections must not lead to a scenario where the pipeline hangs
- Stages must fit on thread
- feedback must be within a thread (future)
- All edges have the same fs and frame_size (until future enhancements)

class audio_dsp.design.pipeline.**PipelineStage**(***kwargs*)

Stage for the pipelne. Does not support processing of data through it. Only used for pipeline level control commands, for example, querying the pipeline checksum.

add_to_dot(*dot*)

Override the CompositeStage.add_to_dot() function to ensure PipelineStage type stages are not added to the dot diagram.

Parameters

dot
[graphviz.Diagraph]
dot instance to add edges to.

audio_dsp.design.pipeline.**callonce**(*f*)

Decorate functions to ensure they only execute once despite being called multiple times.

audio_dsp.design.pipeline.**generate_dsp_main**(*pipeline: Pipeline*, *out_dir='build/dsp_pipeline'*)

Generate the source code for adsp_generated_<x>.c.

Parameters

pipeline
[Pipeline] The pipeline to generate code for.
out_dir
[str] Directory to store generated code in.

audio_dsp.design.pipeline_executor

Utilities for processing the pipeline on the host machine.

class audio_dsp.design.pipeline_executor.**ExecutionResult**(*result: ndarray*, *fs: float*)

The result of processing samples through the pipeline.

Parameters

result
The data produced by the pipeline.
fs
sample rate

Attributes

data
ndarray containing the results of the pipeline.
fs
Sample rate.

play(*channel: int*)

Create a widget in the jupyter notebook to listen to the audio.

> **Warning:** This will not work outside of a jupyter notebook.

Parameters

channel
The channel to listen to.

plot(*path: Optional[Union[Path, str]] = None*)

Display a time domain plot of the result. Save to file if path is not None.

Parameters

path
If path is not none then the plot will be saved to a file and not shown.

plot_magnitude_spectrum(*path: Optional[Union[Path, str]] = None*)

Display a spectrum plot of the result. Save to file if path is not None.

Parameters

path
If path is not none then the plot will be saved to a file and not shown.

plot_spectrogram(*path: Optional[Union[Path, str]] = None*)

Display a spectrogram plot of the result. Save to file if path is not None.

Parameters

path
If path is not none then the plot will be saved to a file and not shown.

to_wav(*path: str | pathlib.Path*)

Save output to a wav file.

class audio_dsp.design.pipeline_executor.**PipelineExecutor**(*graph: Graph[Stage], view_getter: Callable[[], PipelineView]*)

Utility for simulating the pipeline.

Parameters

graph
The pipeline graph to simulate

log_chirp(*length_s: float = 0.5, amplitude: float = 1, start: float = 20, stop: Optional[float] = None*) → ExecutionResult

Generate a logarithmic chirp of constant amplitude and play through the simulated pipeline.

Parameters

length_s
Length of generated chirp in seconds.

amplitude
Amplitude of the generated chirp, between 0 and 1.

start
Start frequency.

stop
Stop frequency. Nyquist if not set

Returns

ExecutionResult
The output wrapped in a helpful container for viewing, saving, processing, etc.

`process`(*data: ndarray*) → ExecutionResult

Process the DSP pipeline on the host.

Parameters

data
Pipeline input to process through the pipeline. The shape must match the number of channels that the pipeline expects as an input; if this is 1 then it may be a 1 dimensional array. Otherwise, it must have shape (num_samples, num_channels).

Returns

ExecutionResult
A result object that can be used to visualise or save the output.

`class audio_dsp.design.pipeline_executor.PipelineView`(*stages: Optional[list[audio_dsp.design.stage.S*
inputs: list[audio_dsp.design.stage.StageO
outputs: list[audio_dsp.design.stage.StageO

A view of the DSP pipeline that is used by PipelineExecutor.

`inputs: list[audio_dsp.design.stage.StageOutput]`
Alias for field number 1

`outputs: list[audio_dsp.design.stage.StageOutput]`
Alias for field number 2

`stages: Optional[list[audio_dsp.design.stage.Stage]]`
Alias for field number 0

audio_dsp.design.plot

Helper functions for displaying plots in the jupyter notebook pipeline design.

`audio_dsp.design.plot.plot_frequency_response`(*f*, *h*, *name=''*, *range=50*)

Plot the frequency response.

Parameters

f
[numpy.ndarray] Frequencies (The X axis)

h
[numpy.ndarray] Frequency response at the corresponding frequencies in `f`

name
: [str] String to include in the plot title, if not set there will be no title.

range
: [int | float] Set the Y axis lower limit in dB, upper limit will be the maximum magnitude.

audio_dsp.design.stage

The edges and nodes for a DSP pipeline.

class audio_dsp.design.stage.**PropertyControlField**(*get*, *set=None*)

For stages which have internal state they can register callbacks for getting and setting control fields.

property value

The current value of this control field.
Determined by executing the getter method.

class audio_dsp.design.stage.**Stage**(*inputs: StageOutputList*, *config: Optional[Union[Path, str]] = None*, *name: Optional[str] = None*, *label: Optional[str] = None*)

Base class for stages in the DSP pipeline. Each subclass should have a corresponding C implementation. Enables code generation, tuning and simulation of a stage.

The stages config can be written and read using square brackets as with a dictionary. This is shown in the below example, note that the config field must have been declared in the stages yaml file.

self["config_field"] = 2 assert self["config_field"] == 2

Parameters

config
: [str | Path] Path to yaml file containing the stage definition for this stage. Config parameters are derived from this config file.

inputs
: [Iterable[StageOutput]] Pipeline edges to connect to self

name
: [str] Name of the stage. Passed instead of config when the stage does not have an associated config yaml file

label
: [str] User defined label for the stage. Used for autogenerating a define for accessing the stage's index in the device code

Attributes

i
: [list[StageOutput]] This stages inputs.

fs
: [int | None] Sample rate.

frame_size
: [int | None] Samples in frame.

name
: [str] Stage name determined from config file

yaml_dict
[dict] config parsed from the config file

label
[str] User specified label for the stage

n_in
[int] number of inputs

n_out
[int] number of outputs

details
[dict] Dictionary of descriptive details which can be displayed to describe current tuning of this stage

dsp_block
[None | audio_dsp.dsp.generic.dsp_block] This will point to a dsp block class (e.g. biquad etc), to be set by the child class

add_to_dot(*dot*)

Add this stage to a diagram that is being constructed. Does not add the edges.

Parameters

dot
[graphviz.Diagraph] dot instance to add edges to.

property constants

Get a copy of the constants for this stage.

create_outputs(*n_out*)

Create this stages outputs.

Parameters

n_out
[int] number of outputs to create.

get_config()

Get a dictionary containing the current value of the control fields which have been set.

Returns

dict
current control fields

get_frequency_response(*nfft=32768*) → tuple[numpy.ndarray, numpy.ndarray]

Return the frequency response of this instance's dsp_block attribute.

Parameters

nfft
The length of the FFT

Returns

ndarray, ndarray
Frequency values, Frequency response for this stage.

get_required_allocator_size()

Calculate the required statically-allocated memory in bytes for this stage. Formats this into a compile-time determinable expression.

Returns

compile-time determinable expression of required allocator size.

`property o: StageOutputList`

This stage's outputs. Use this object to connect this stage to the next stage in the pipeline. Subclass must call self.create_outputs() for this to exist.

`plot_frequency_response(`*nfft=32768*`)`

Plot magnitude and phase response of this stage using matplotlib. Will be displayed inline in a jupyter notebook.

Parameters

nfft
[int] Number of frequency bins to calculate in the fft.

`process(`*in_channels*`)`

Run dsp object on the input channels and return the output.

Args:
in_channels: list of numpy arrays

Returns

list of numpy arrays.

`set_constant(`*field*, *value*, *value_type*`)`

Define constant values in the stage. These will be hard coded in the autogenerated code and cannot be changed at runtime.

Parameters

field
[str] name of the field

value
[ndarray or int or float or list] value of the constant. This can be an array or scalar

`set_control_field_cb(`*field*, *getter*, *setter=None*`)`

Register callbacks for getting and setting control fields, to be called by classes which implement stage.

Parameters

field
[str] name of the field

getter
[function] A function which returns the current value

setter
[function] A function which accepts 1 argument that will be used as the new value

`class audio_dsp.design.stage.StageOutput(`*fs=48000*, *frame_size=1*`)`

The Edge of a dsp pipeline.

Parameters

fs
[int] Edge sample rate Hz

frame_size
[int] Number of samples per frame

Attributes

source
[audio_dsp.design.graph.Node] Inherited from Edge

dest
[audio_dsp.design.graph.Node] Inherited from Edge

source_index
[int | None] The index of the edge connection to source.

fs
[int] see fs parameter

frame_size
[int] see frame_size parameter

property dest_index: int | None
The index of the edge connection to the dest.

class audio_dsp.design.stage.StageOutputList(*edges: list[audio_dsp.design.stage.StageOutput | None] | None = None*)

A container of StageOutput.

A stage output list will be created whenever a stage is added to the pipeline. It is unlikely that a StageOutputList will have to be explicitly created during pipeline design. However the indexing and combining methods shown in the example will be used to create new StageOutputList instances.

Parameters

edges
list of StageOutput to create this list from.

Examples

This example shows how to combine StageOutputList in various ways:

```
# a and b are StageOutputList
a = some_stage.o
b = other_stage.o

# concatenate them
a + b

# Choose a single channel from 'a'
a[0]

# Choose channels 0 and 3 from 'a'
a[0, 3]

# Choose a slice of channels from 'a', start:stop:step
a[0:10:2]

# Combine channels 0 and 3 from 'a', and 2 from 'b'
a[0, 3] + b[2]

# Join 'a' and 'b', with a placeholder "None" in between
a + None + b
```

Attributes

edges
[list[StageOutput]] To access the actual edges contained within this list then read from the edges attribute. All methods in this class return new StageOutputList instances (even when the length is 1).

class audio_dsp.design.stage.**ValueControlField**(*value=None*)

Simple field which can be updated directly.

audio_dsp.design.stage.**all_stages**() → dict[str, Type[audio_dsp.design.stage.Stage]]

Get a dict containing all stages in scope.

audio_dsp.design.stage.**find_config**(*name*)

Find the config yaml file for a stage by looking for it in the default directory for built in stages.

Parameters

name
[str] Name of stage, e.g. a stage whose config is saved in "biquad.yaml" should pass in "biquad".

Returns

Path
Path to the config file.

audio_dsp.design.thread

Contains classes for adding a thread to the DSP pipeline.

class audio_dsp.design.thread.**DSPThreadStage**(***kwargs*)

Stage for the DSP thread. Does not support processing of data through it. Only used for DSP thread level control commands, for example, querying the max cycles consumed by the thread.

add_to_dot(*dot*)

Exclude this stage from the dot diagram.

Parameters

dot
[graphviz.Diagraph] dot instance to add edges to.

class audio_dsp.design.thread.**Thread**(*id: int*, ***kwargs*)

A composite stage used to represent a thread in the pipeline. Create using Pipeline.thread rather than instantiating directly.

Parameters

id
[int] Thread index

kwargs
[dict] forwarded to __init__ of CompositeStage

Attributes

id
[int] Thread index
thread_stage
[Stage] DSPThreadStage stage

add_thread_stage()
Add to this thread the stage which manages thread level commands.

Copyright & Disclaimer