lib_audio_dsp - Tool User Guide

Release: 1.0.0
Publication Date: 2024/07/30
Document Number: XM-015034-PC

Table of Contents

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

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 a console sometimes does not work as expected on Windows. Ensure that each line is copied and executed separately.

1.1 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)

1.2 Software Requirements

- Graphviz: this software must installed and the `dot` executable must be on the system path.
- XTC 15.3.0
- Python 3.10
- Jupyter 7.2.1
- CMake

Additionally, on Windows the following is required:

- ninja-build

1.3 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:

      ```
      "C:\Program Files (x86)\XMOS\XTC\15.3.0\SetEnv"
      ```

      or similar

   On Linux and macOS:

   1. Open a terminal

   2. Activate the XTC environment using *SetEnv*

2. Create a sandbox folder with the command below:

```
mkdir lib_audio_dsp_sandbox
```

3. Clone the library inside *lib_audio_dsp_sandbox*:

```
git clone git@github.com:xmos/lib_audio_dsp.git
```

4. Get the sandbox 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 requirements file inside *lib_audio_dsp_sandbox*.

   On Windows:

```
echo -e lib_audio_dsp/python > requirements.txt
echo notebook >> requirements.txt
```

   On Linux or macOS:

```
echo "-e lib_audio_dsp/python" > requirements.txt
echo notebook >> requirements.txt
chmod 644 requirements.txt
```

6. Create a Python virtualenv inside *lib_audio_dsp_sandbox*.

   On Windows:

```
python -m venv .venv
.venv\Scripts\activate.bat
pip install -Ur requirements.txt
cd ..
```

   On Linux or macOS:

```
python -m venv .venv
source .venv/bin/activate
pip install -Ur requirements.txt
cd ..
```

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

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 following page should open:

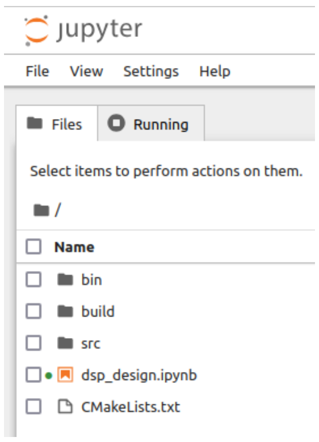


Fig. 1.1: Top-level page of the Jupyter Notebook

9. Run all the cells from the browser. From the menu at the top of the page click *Run -> Run all cells*:

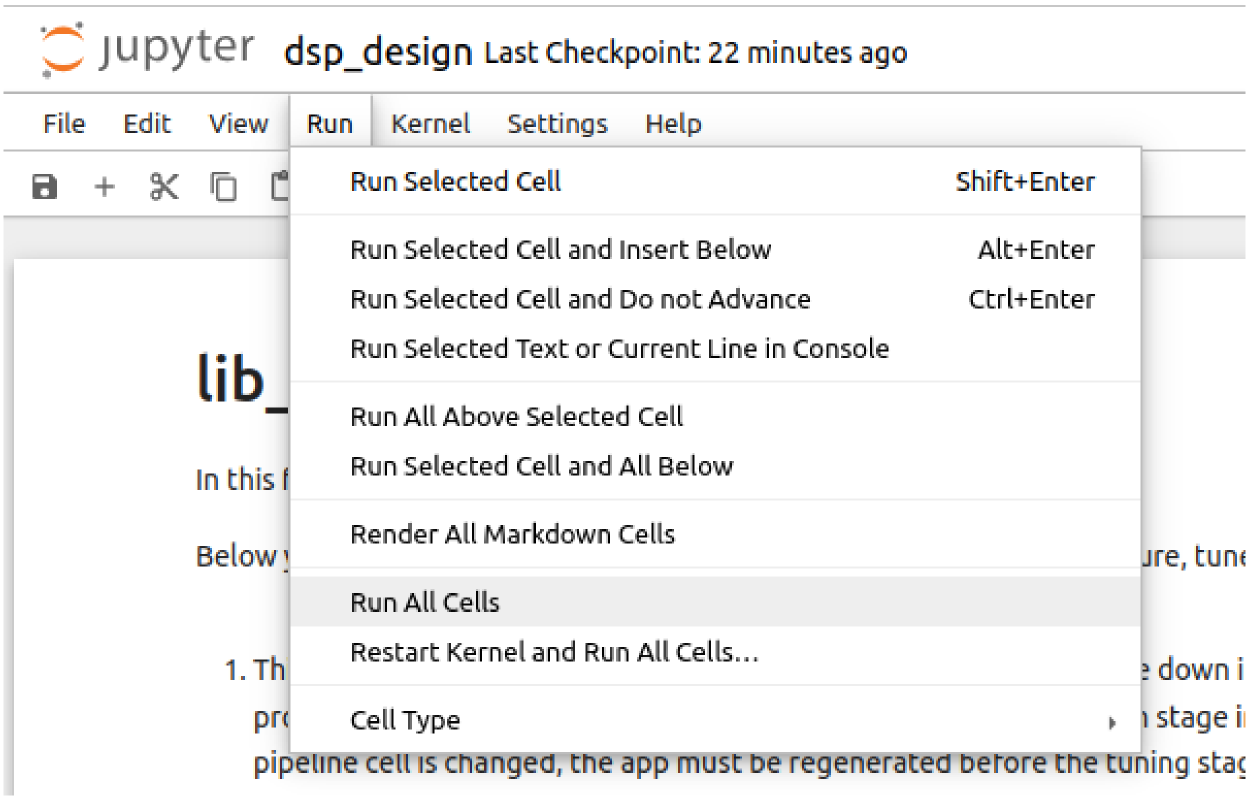
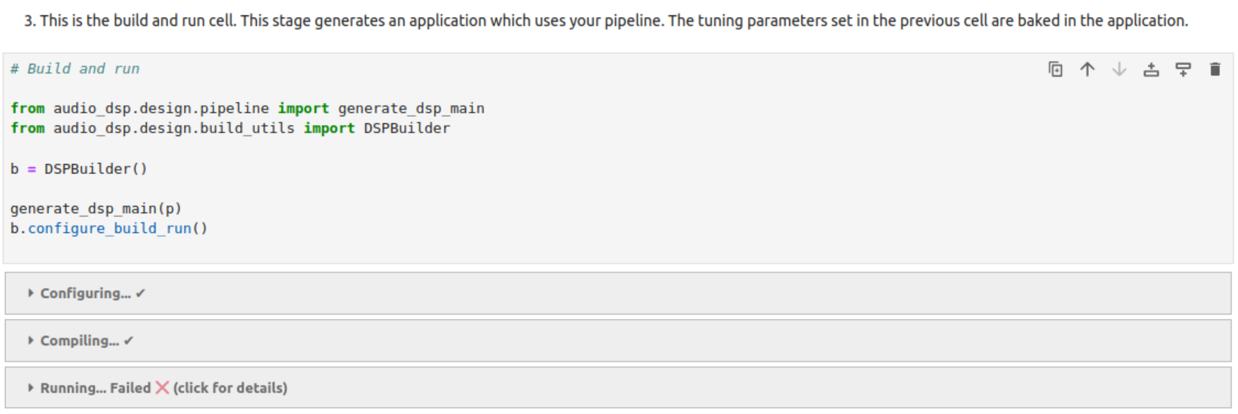

jupyter dsp_design Last Checkpoint: 22 minutes ago
File Edit View Run Kernel Settings Help
Run Selected Cell Shift+Enter
Run Selected Cell and Insert Below Alt+Enter
Run Selected Cell and Do not Advance Ctrl+Enter
Run Selected Text or Current Line in Console
Run All Above Selected Cell
Run Selected Cell and All Below
Render All Markdown Cells
Run All Cells
Restart Kernel and Run All Cells...
Cell Type

Fig. 1.2: Run menu of the Jupyter Notebook

This creates the pipeline and builds the app. Wait for all the cells to finish

Any configuration or compilation errors will be displayed in the notebook in the *Build and run* cell, as in the example below:



3. This is the build and run cell. This stage generates an application which uses your pipeline. The tuning parameters set in the previous cell are baked in the application.

```python
# Build and run

from audio_dsp.design.pipeline import generate_dsp_main
from audio_dsp.design.build_utils import DSPBuilder

b = DSPBuilder()

generate_dsp_main(p)
b.configure_build_run()
```

```
Configuring... ✓
Compiling... ✓
Running... Failed ✗ (click for details)
```

Fig. 1.3: Run error of the Jupyter Notebook

10. Update and run *Pipeline design stage* to add the desired audio processing blocks. A diagram will be generated showing the pipeline IO mapping.

    A simple pipeline example is shown in Fig. 1.4:

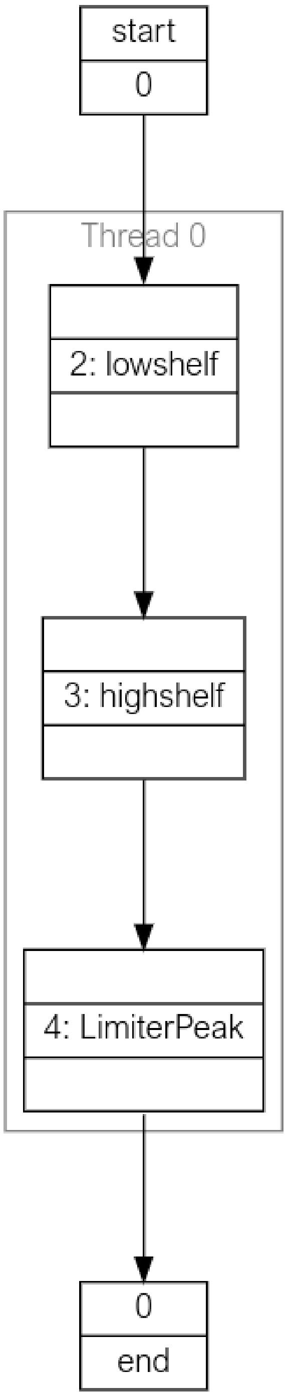


Fig. 1.4: Diagram of a simple audio pipeline

See the top of the notebook for more information about this stage.

11. Update and run the *Tuning Stage* cell to change the parameters before building. See the top of the notebook for more information about this stage.

1.4 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.
   - Enable the Python Virtual Environment: this is checked by running the command `echo %VIRTUAL_ENV%` on Windows, or `echo $VIRTUAL_ENV` on Linux or macOS. The path should have been set.
2. Open the notebook by running `jupyter notebook lib_audio_dsp/examples/app_simple_audio_dsp_integration/dsp_design.ipynb` from `lib_audio_dsp_sandbox`, as described in the *Setup Steps* section.

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.

2.1 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 following processes:

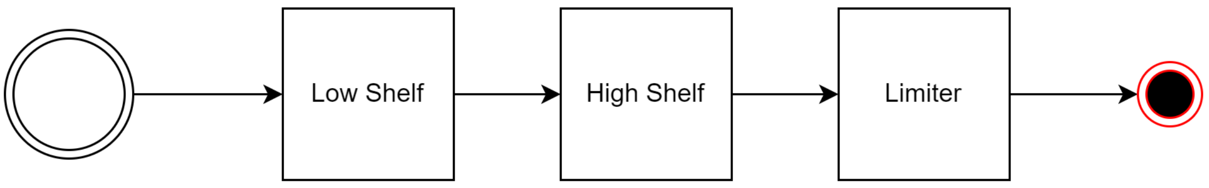


Fig. 2.1: The target pipeline

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`.

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

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

# 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")
```

(continues on next page)

(continued from previous page)

```
# 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()
```

When running the above snippet in a Jupyter Notebook it will output the following image which illustrates the pipeline which has been designed:

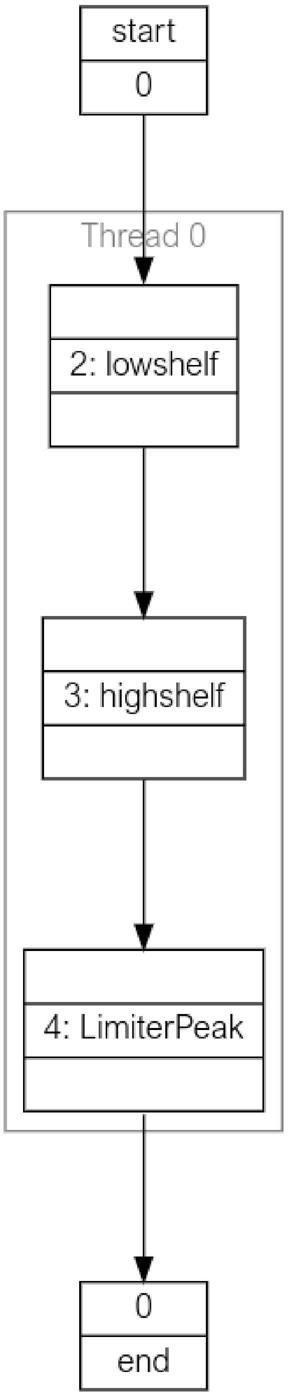


Fig. 2.2: Generated pipeline diagram

2.2 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. 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()
```

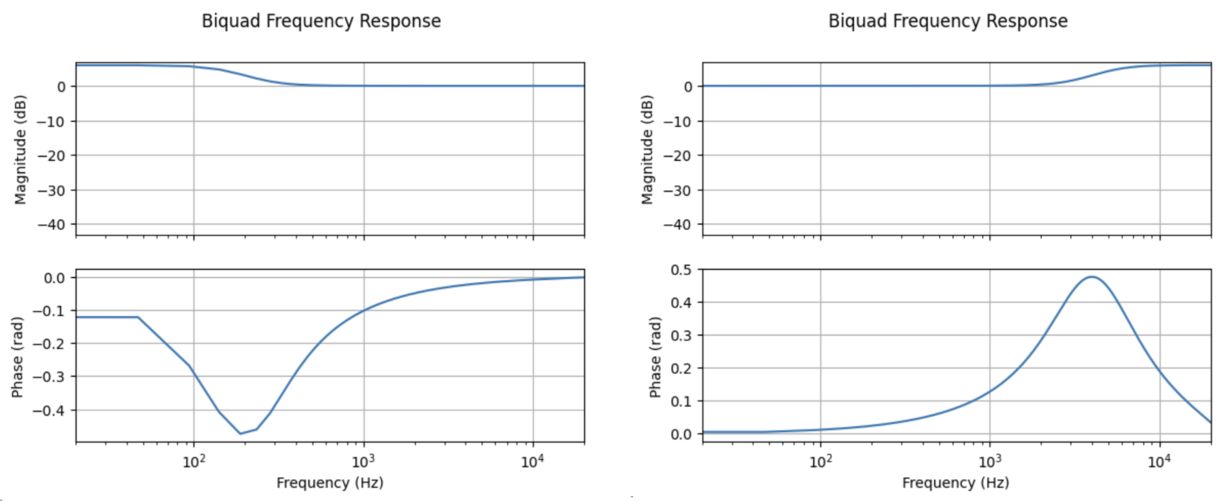


Fig. 2.3: Frequency response of the biquads (low shelf left, high shelf right)

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

2.2.1 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.

With that the tuned DSP pipeline will be running on the xcore device and 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.

2.3 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 created, every stage's initialiser returns an instance of `StageOutputList`, a container of `StageOutput`. The stage's outputs can be selected from the StageOutputList by indexing into it, creating a new StageOutputList, which can be concatenated with other StageOutputList instances using the + operator. When creating a stage, it will require a StageOutputList as its inputs.

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

```
# split the pipeline inputs
i0 = p.stage(Biquad, i[0:2])      # use the first 2 inputs
i1 = p.stage(Biquad, i[2])        # use the third input (index 2)
i2 = p.stage(Biquad, i[3, 5, 6])  # use the inputs at index 3, 5, and 6
# join biquad outputs
i3 = p.stage(Biquad, i0 + i1 + i2[0]) # pass all of i0 and i1, as well as the first channel in i2

p.set_outputs(i3 + i2[1:]) # The pipeline output will be all i3 channels and the 2nd and 3rd channel
↪from i2.
```

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)
```

3 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()`.

Once the code is generated use the functions defined in *stages/adsp_pipeline.h* to read and write samples to the pipeline and update configuration fields.

3.1 C Design API

3.1.1 stages/adsp_control.h

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

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**

3.1.2 stages/adsp_module.h

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**

3.1.3 stages/adsp_pipeline.h

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**

3.2 Python Design API

3.2.1 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: Optional[Union[str, Path]] = None, build_dir: Optional[Union[str, Path]] = None, bin_dir: Optional[Union[str, Path]] = None, project_name: Optional[str] = None, config_name: Optional[str] = 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`() → 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!".

`run`() → int

Invoke xrun with the options specified in this class instance. Invokation will be of the form "xrun <binary>", where the path to the binary is constructed as per this class' docstring.

Returns

returncode
Return code from the invokation of xrun. 0 if success.

3.2.2 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: Optional[str] = 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.

3.2.3 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.

3.2.4 audio_dsp.design.host_app

Global host app management, to provide easy access to the host app.

exception `audio_dsp.design.host_app.`**`DeviceConnectionError`**

Raised when the host app cannot connect to the device.

exception `audio_dsp.design.host_app.`**`InvalidHostAppError`**

Raised when there is a issue with the configured host app.

`audio_dsp.design.host_app.`**`send_control_cmd`**(*instance_id*, **args*, *verbose=False*)

Send a control command from the host to the device.

Parameters

instance_id
[int | str] Instance id of the stage to which this command is sent

***args**
[list[str]] Command + arguments for this control command

verbose
[bool] When set to true, print the full command that gets issued

Raises

InvalidHostAppError
If set_host_app() hasn't been called to set the host app and transport protocol before calling this function. If the transport protocol is 'xscope', and port num has not been set by calling set_host_app_xscope_port() before calling this function.

DeviceConnectionError
If the device is not programmed and/or not connected to the host

`audio_dsp.design.host_app.`**`set_host_app`**(*host_app*, *transport_protocol='usb'*)

Set the host_app and the transport protocol to use for control.

Parameters

host_app
[str] Host app file

transport_protocol
[str] Protocol to use for control. Supported transport protocols are usb and xscope

Raises

InvalidHostAppError
If an invalid host app or transport protocol is selected.

`audio_dsp.design.host_app.`**`set_host_app_xscope_port`**(*port_num*)

Set the port number on which to communicate with the device when doing control over xscope.

Parameters

host_app
[int] Port number

Returns

The return value from the subprocess.run(). The caller can use this to check the returncode, stdout etc.

Raises

InvalidHostAppError
If the port is set before calling set_host_app() or if the port is set when transport protocol is not xscope

3.2.5 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.

3.2.6 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*)
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

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*)

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: Optional[Path] = 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

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: Optional[str] = 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.**profile_pipeline**(*pipeline: Pipeline*)

Profiles the DSP threads that are a part of the pipeline. Make sure set_host_app() is called before calling this to set a valid host app.

Parameters

pipeline
[Pipeline] A designed and optionally tuned pipeline

audio_dsp.design.pipeline.**send_config_to_device**(*pipeline: Pipeline*)

Send the current config for all stages to the device. Make sure set_host_app() is called before calling this to set a valid host app.

Parameters

pipeline
[Pipeline] A designed and optionally tuned pipeline

`audio_dsp.design.pipeline.`**`validate_pipeline_checksum`**`(`*pipeline: Pipeline*`)`

Check if Python and device pipeline checksums match. Raise a runtime error if the checksums are not equal. The check is performed only if the host application can connect to the device.

Parameters

pipeline
[Python pipeline for which to validate checksum against the device pipeline]

3.2.7 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[str, Path]] = 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[str, Path]] = 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[str, Path]] = 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.Stage]], inputs: list[audio_dsp.design.stage.StageOutput], outputs: list[audio_dsp.design.stage.StageOutput]*)

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

3.2.8 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.

3.2.9 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[str, Path]] = 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.

`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=512*) → 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=512*)

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: Optional[list[audio_dsp.design.stage.StageOutput | 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]
```

(continues on next page)

(continued from previous page)

```
# 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.

3.2.10 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.