lib_audio_dsp - DSP Run-time Control guide

Release: 1.1.0
Publication Date: 2024/09/23
Document Number: XM-015034-PC

Table of Contents

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.

1 Control Interface Walkthrough

1.1 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.

```
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)
```

This code snippet will generate the pipeline diagram shown in Fig. 1.1.

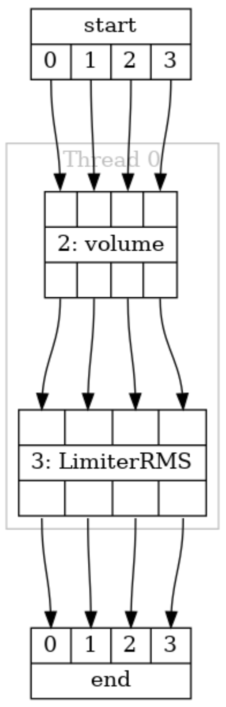

Fig. 1.1: The example pipeline diagram

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:

```
#pragma once
```

(continues on next page)

(continued from previous page)

```
#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.

1.2 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 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 *dsp/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 Tool User Guide, in the pipeline_design_api 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.

```c
#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)
  );
}
```

1.3 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.

```c
int32_t read_volume_gain(adsp_controller_t* control) {
```

(continues on next page)

(continued from previous page)

```
  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;
}
```

1.3.1 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.

2 Run-Time 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.

2.1 Biquad helpers

Functions

void **`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

void **`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

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

void **`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

void **`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

void **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

void **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

void **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

void **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

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

void **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

2.2 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

2.3 Reverb helpers

Functions

static inline int32_t **`adsp_reverb_float2int`**(float x)

Convert a floating point value to the Q_VERB format, saturate out of range values. Accepted range is 0 to 1

Parameters

- **`x`** – A floating point number

Returns

postive Q_VERB int32_t value

static inline int32_t **`adsp_reverb_db2int`**(float db)

Convert a floating point gain in decibels into a linear Q_VERB value for use in controlling the reverb gains.

Parameters

- **`db`** – Floating point value in dB, values above 0 will be clipped.

Returns

Q_VERB fixed point linear gain.

static inline int32_t **`adsp_reverb_calculate_damping`**(float damping)

Convert a user damping value into a Q_VERB fixed point value suitable for passing to a reverb.

Parameters

- **`damping`** – The chose value of damping.

Returns

Damping as a Q_VERB fixed point integer, clipped to the accepted range.

static inline int32_t **`adsp_reverb_calculate_feedback`**(float decay)

Calculate a Q_VERB 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_VERB fixed point integer.

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

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 mic 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`** – Lenght 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

2.4 Signal chain helpers

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