fwk_voice - User Guide

Release: 0.5.1

Publication Date: 2023/03/20

Table of Contents

The Voice Framework Design Guide is written for system architects and engineers designing Far-field voice systems using the XCORE.AI processor. The document describes typical usage models, the processor architecture, key feature operation, and interface definitions. In conjunction with the product datasheet, these two documents provide all the information required for system design, from concept to production testing and verification.

It is expected that this document is read in conjunction with the relevant datasheet and that the user is familiar with basic voice processing terminology.

1 Audio Processing

At the core of the Voice Framework are high-performance audio processing algorithms. The algorithms are connected in a pipeline that takes its input from a pair of the microphone and executes a series of signal processing algorithms to extract a voice signal from a complex soundscape. The audio pipeline can accept a reference signal from a host system which is used to perform Acoustic Echo Cancellation (AEC) to remove audio being played by the host. The audio pipeline provides two different output channels - one that is optimized for Automatic Speech Recognition systems and the other for voice communications.

A flexible audio signal routing infrastructure and a range of digital inputs and outputs enables the Voice Framework to be integrated into a wide range of system configurations, that can be configured at start up and during operation through a set of control registers. In addition, all source code is provided to allow for full customization or the addition of other audio processing algorithms.

1.1 Audio Features

1.1.1 Acoustic Echo Canceller Library

`lib_aec` is a library which provides functions that can be put together to perform Acoustic Echo Cancellation (AEC) on input mic data using the input reference data to model the room echo characteristics. `lib_aec` library functions make use of functionality provided in `lib_xcore_math` to perform DSP operations. For more details refer to *AEC Overview*.

Repository Structure

- `modules/lib_aec` - The actual `lib_aec` library directory within `https://github.com/xmos/fwk_voice/`. Within `lib_aec`
  - `api/` - Headers containing the public API for `lib_aec`.
  - `doc/` - Library documentation source (for non-embedded documentation) and build directory.
  - `src/` - Library source code.

Requirements

`lib_aec` is included as part of the `fwk_voice` github repository and all requirements for cloning and building `fwk_voice` apply. `lib_aec` is compiled as a static library as part of overall `fwk_voice` build. It depends on lib_xcore_math.

API Structure

The API can be categorised into high level and low level functions.

High level API has fewer input arguments and is simpler. However, it provides limited options for calling functions in parallel across multiple threads. Keeping API simplicity in mind, most of the high level API functions accept a pointer to the AEC state structure as an input and modify the relevant part of the AEC state. API and example

documentation provides more details about the fields within the state modified when calling a given function. High level API functions allow 2 levels of parallelism:

- Single level of parallelism where for a given function, main and shadow filter processing can happen in parallel.
- Two levels of parallelism where a for a given function, processing across multiple channels as well as main and shadow filter can be done in parallel.

Low level API has more input arguments but allows more freedom for running in parallel across multiple threads. Low level API function names begin with a `aec_l2_` prefix. Depending on the low level API used, functions can be run in parallel to work over a range of bins or a range of phases. This API is still a work in progress and will be fully supported in the future.

Getting and Building

This repo is got as part of the parent `fwk_voice` repo clone. It is compiled as a static library as part of fwk_voice compilation process.

To include `lib_aec` in an application as a static library, the generated `libfwk_voice_module_lib_aec.a` can then be linked into the application. Be sure to also add `lib_aec/api` as an include directory for the application.

AEC Overview

The lib_aec library provides functions that can be put together to perform Automatic Echo Cancellation on input microphone data by using input reference data to model the echo characteristics of the room.

The echo canceller takes in one or more channels of microphone (mic) input and one or more channels of reference input data. The mic input is the input captured by the device microphones. Reference input is the audio that is played out of the device speakers. The echo canceller uses the reference input to model the room echo characteristics for each mic-loudspeaker pair and outputs an echo cancelled version of the mic input. AEC uses adaptive filters, one per mic-speaker pair to constantly remove echo from the the mic input. The filters continually adapt to the acoustic environment to accommodate changes in the room created by events such as doors opening or closing and people moving about.

Echo cancellation is performed on a frame by frame basis. Each frame is made of 15msec chunks of data, which is 240 samples at 16kHz input sampling frequency, per input channel. For example, for a 2 mic channel and 2 reference channel input configuration, an input frame is made of 2x240 samples of mic data and 2x240 samples of reference data. Input data is expected to be in fixed point 32bit 1.31 format. Further, in this example, there will be a total of 4 adaptive filters; $\hat{H}_{y0x0}$, $\hat{H}_{y0x1}$, $\hat{H}_{y1x0}$ and $\hat{H}_{y1x1}$, monitoring the echo seen in mic channel 0 from reference channel 0 and 1 and echo seen in mic channel 1 from reference channel 0 and 1.

Microphone data is referred to as y when in time domain and Y when in frequency domain. In general throughout the code, names starting with lower case represent time domain and those beginning with upper case represent frequency domain. For example $error$ is the filter error and $Error$ is the spectrum of the filter error. Reference input is referred to as x in time domain and X when in frequency domain. Filter is referred to as $\hat{h}$ in time domain and $\hat{H}$ in frequency domain.

A filter has multiple phases. The term phases refers to the tail length of the filter. A filter with more phases or a longer tail length will be able to model a more reverberant room response leading to better echo cancellation.

There are 2 types of adaptive filters used in the AEC. These are referred to as main filter and shadow filter. The main filter as the name suggests is the main filter that is used to generate the echo cancelled output of the AEC. Shadow filter is a filter that used to quickly detect and respond to changes in the room transfer function. There is one main filter and one shadow filter per x-y pair. Typically the main filter has more phases than the shadow

filter. Fewer phases in the shadow filter enable it to rapidly detect and respond to changes while more phases in main filter lead to deeper convergence and hence better echo cancellation at the AEC output.

Before starting AEC processing or every time there's a configuration change, the user needs to call aec_init() to initialise the echo canceller for a desired configuration. Once the AEC is initialised, the library functions can be called in a logical order to perform echo cancellation on a frame by frame basis. Refer to the aec_1_thread and aec_2_threads examples to see how the functions are called to perform echo cancellation using one thread or 2 threads.

API Reference

AEC Data Structure and Enum Definitions

group `aec_types`

Enums

enum `aec_adaption_e`

Values:

enumerator `AEC_ADAPTION_AUTO`

Compute filter adaption config every frame.

enumerator `AEC_ADAPTION_FORCE_ON`

Filter adaption always ON.

enumerator `AEC_ADAPTION_FORCE_OFF`

Filter adaption always OFF.

enum `shadow_state_e`

Values:

enumerator `LOW_REF`

Not much reference so no point in acting on AEC filter logic.

enumerator `ERROR`

something has gone wrong, zero shadow filter

enumerator `ZERO`

shadow filter has been reset multiple times, zero shadow filter

enumerator `RESET`

copy main filter to shadow filter

enumerator `EQUAL`

main filter and shadow filter are similar

enumerator `SIGMA`

shadow filter bit better than main, reset sigma_xx for faster convergence

enumerator `COPY`

shadow filter much better, copy to main

struct `coherence_mu_config_params_t`

Public Members

float_s32_t `coh_alpha`

Update rate of `coh`.

float_s32_t `coh_slow_alpha`

Update rate of `coh_slow`.

float_s32_t `coh_thresh_slow`

Adaption frozen if coh below (coh_thresh_slow*coh_slow)

float_s32_t `coh_thresh_abs`

Adaption frozen if coh below coh_thresh_abs.

float_s32_t `mu_scalar`

Scalefactor for scaling the calculated mu.

float_s32_t `eps`

Parameter to avoid divide by 0 in coh calculation.

float_s32_t `thresh_minus20dB`

-20dB threshold

float_s32_t `x_energy_thresh`

X_energy threshold used for determining if the signal has enough reference energy for sensible coherence mu calculation

unsigned `mu_coh_time`

Number of frames after low coherence, adaption frozen for.

unsigned `mu_shad_time`

Number of frames after shadow filter use, the adaption is fast for

aec_adaption_e `adaption_config`

Filter adaption mode. Auto, force ON or force OFF

int32_t `force_adaption_mu_q30`

Fixed mu value used when filter adaption is forced ON

struct `shadow_filt_config_params_t`

Public Members

float_s32_t `shadow_sigma_thresh`

threshold for resetting sigma_XX.

float_s32_t `shadow_copy_thresh`

threshold for copying shadow filter.

float_s32_t `shadow_reset_thresh`

threshold for resetting shadow filter.

float_s32_t `shadow_delay_thresh`

threshold for turning off shadow filter reset if reference delay is large

float_s32_t `x_energy_thresh`

X energy threshold used for deciding whether the system has enough reference energy for main and shadow filter comparison to make sense

float_s32_t `shadow_mu`

fixed mu value used during shadow filter adaption.

int32_t `shadow_better_thresh`

Number of times shadow filter needs to be better before it gets copied to main filter.

int32_t `shadow_zero_thresh`

Number of times shadow filter is reset by copying the main filter to it before it gets zeroed.

int32_t `shadow_reset_timer`

Number of frames between zeroing resets of shadow filter.

struct `aec_core_config_params_t`

Public Members

int `bypass`

bypass AEC flag.

int `gamma_log2`

parameter for deriving the gamma value that used in normalisation spectrum calculation. gamma is calculated as 2^gamma_log2

uint32_t `sigma_xx_shift`

parameter used for deriving the alpha value used while calculating EMA of X_energy to calculate sigma_XX.

float_s32_t `delta_adaption_force_on`

delta value used in normalisation spectrum computation when adaption is forced as always ON.

float_s32_t `delta_min`

Lower limit of delta computed using fractional regularisation.

uint32_t `coeff_index`

coefficient index used to track H_hat index when sending H_hat values over the host control interface.

uq2_30 `ema_alpha_q30`

alpha used while calculating y_ema_energy, x_ema_energy and error_ema_energy.

struct `aec_config_params_t`

#include <aec_state.h> AEC control parameters.

This structure contains control parameters that the user can modify at run time.

Public Members

coherence_mu_config_params_t `coh_mu_conf`

Coherence mu related control params.

shadow_filt_config_params_t `shadow_filt_conf`

Shadow filter related control params.

aec_core_config_params_t `aec_core_conf`

All AEC control params except those for coherence mu and shadow filter.

struct `coherence_mu_params_t`

Public Members

float_s32_t `coh`

Moving average coherence.

float_s32_t `coh_slow`

Slow moving average coherence.

int32_t `mu_coh_count`

Counter for tracking number of frames coherence has been low for.

int32_t `mu_shad_count`

Counter for tracking number of frames shadow filter has been used in.

float_s32_t `coh_mu`[*AEC_LIB_MAX_X_CHANNELS*]

Coherence mu.

struct `shadow_filter_params_t`

Public Members

int32_t `shadow_flag`[*AEC_LIB_MAX_Y_CHANNELS*]

shadow_state_e enum indicating shadow filter status

int `shadow_reset_count`[*AEC_LIB_MAX_Y_CHANNELS*]

counter for tracking shadow filter resets

int `shadow_better_count`[*AEC_LIB_MAX_Y_CHANNELS*]

counter for tracking shadow filter copy to main filter

struct `aec_shared_state_t`

#include <aec_state.h> AEC shared state structure.

Data structures holding AEC persistent state that is common between main filter and shadow filter. *aec_state_t::shared_state* for both main and shadow filter point to the common aec_shared_t structure. [*aec_shared_state_t*]

Public Members

bfp_complex_s32_t `X_fifo`[*AEC_LIB_MAX_X_CHANNELS*][*AEC_LIB_MAX_PHASES*]

BFP array pointing to the reference input spectrum phases. The term **phase** refers to the spectrum data for a frame. Multiple phases means multiple frames of data.

For example, 10 phases would mean the 10 most recent frames of data. Each phase spectrum, pointed to by X_fifo[i][j]->data is stored as a length AEC_FD_FRAME_LENGTH, complex 32bit array.

The phases are ordered from most recent to least recent in the X_fifo. For example, for an AEC configuration of 2 x-channels and 10 phases per x channel, 10 frames of X data spectrum is stored in the X_fifo. For a given x channel, say x channel 0, X_fifo[0][0] points to the most recent frame's X spectrum and X_fifo[0][9] points to the last phase, i.e the least recent frame's X spectrum.

bfp_complex_s32_t `X`[*AEC_LIB_MAX_X_CHANNELS*]

BFP array pointing to reference input signal spectrum. The X data values are stored as a length AEC_FD_FRAME_LENGTH complex 32bit array per x channel.

bfp_complex_s32_t `Y`[*AEC_LIB_MAX_Y_CHANNELS*]

BFP array pointing to mic input signal spectrum. The Y data values are stored as a length AEC_FD_FRAME_LENGTH complex 32bit array per y channel.

bfp_s32_t `y`[*AEC_LIB_MAX_Y_CHANNELS*]

BFP array pointing to time domain mic input processing block. The y data values are stored as length AEC_PROC_FRAME_LENGTH, 32bit integer array per y channel.

bfp_s32_t `x`[*AEC_LIB_MAX_X_CHANNELS*]

BFP array pointing to time domain reference input processing block. The x data values are stored as length AEC_PROC_FRAME_LENGTH, 32bit integer array per x channel.

bfp_s32_t `prev_y`[*AEC_LIB_MAX_Y_CHANNELS*]

BFP array pointing to time domain mic input values from the previous frame. These are put together with the new samples received in the current frame to make a AEC_PROC_FRAME_LENGTH processing block. The prev_y data values are stored as length (AEC_PROC_FRAME_LENGTH - AEC_FRAME_ADVANCE), 32bit integer array per y channel.

bfp_s32_t `prev_x`[*AEC_LIB_MAX_X_CHANNELS*]

BFP array pointing to time domain reference input values from the previous frame. These are put together with the new samples received in the current frame to make a AEC_PROC_FRAME_LENGTH processing block. The prev_x data values are stored as length (AEC_PROC_FRAME_LENGTH - AEC_FRAME_ADVANCE), 32bit integer array per x channel.

bfp_s32_t `sigma_XX`[*AEC_LIB_MAX_X_CHANNELS*]

BFP array pointing to sigma_XX values which are the weighted average of the X_energy signal. The sigma_XX data is stored as 32bit integer array of length AEC_FD_FRAME_LENGTH

float_s32_t `y_ema_energy`[*AEC_LIB_MAX_Y_CHANNELS*]

Exponential moving average of the time domain mic signal energy. This is calculated by calculating energy per sample and summing across all samples. Stored in a y channels array with every value stored as a 32bit integer mantissa and exponent.

float_s32_t `x_ema_energy`[*AEC_LIB_MAX_X_CHANNELS*]

Exponential moving average of the time domain reference signal energy. This is calculated by calculating energy per sample and summing across all samples. Stored in a x channels array with every value stored as a 32bit integer mantissa and exponent.

float_s32_t `overall_Y`[*AEC_LIB_MAX_Y_CHANNELS*]

Energy of the mic input spectrum. This is calculated by calculating the energy per bin and summing across all bins. Stored in a y channels array with every value stored as a 32bit integer mantissa and exponent.

float_s32_t `sum_X_energy`[*AEC_LIB_MAX_X_CHANNELS*]

Sum of the X_energy across all bins for a given x channel. Stored in a x channels array with every value stored as a 32bit integer mantissa and exponent.

coherence_mu_params_t `coh_mu_state`[*AEC_LIB_MAX_Y_CHANNELS*]

Structure containing coherence mu calculation related parameters.

shadow_filter_params_t `shadow_filter_params`

Structure containing shadow filter related parameters.

aec_config_params_t `config_params`

Structure containing AEC control parameters. These are initialised to the default values and can be changed at runtime by the user.

unsigned `num_y_channels`

Number of mic input channels that the AEC is configured for. This is the input parameter num_y_channels that *aec_init()* gets called with.

unsigned `num_x_channels`

Number of reference input channels that the AEC is configured for. This is the input parameter num_x_channels that *aec_init()* gets called with.

struct `aec_state_t`

#include <aec_state.h> [*aec_shared_state_t*]

AEC state structure.

Data structures holding AEC persistent state. There are 2 instances of *aec_state_t* maintained within AEC; one for main filter and one for shadow filter specific state. [*aec_state_t*]

Public Members

bfp_complex_s32_t `Y_hat`[*AEC_LIB_MAX_Y_CHANNELS*]

BFP array pointing to estimated mic signal spectrum. The Y_data data values are stored as length AEC_FD_FRAME_LENGTH, complex 32bit array per y channel.

bfp_complex_s32_t `Error`[*AEC_LIB_MAX_Y_CHANNELS*]

BFP array pointing to adaptive filter error signal spectrum. The Error data is stored as length AEC_FD_FRAME_LENGTH, complex 32bit array per y channel.

bfp_complex_s32_t `H_hat`[*AEC_LIB_MAX_Y_CHANNELS*][*AEC_LIB_MAX_PHASES*]

BFP array pointing to the adaptive filter spectrum. The filter spectrum is stored as a num_y_channels x total_phases_across_all_x_channels array where each H_hat[i][j] entry points to the spectrum of a single phase.

Number of phases in the filter refers to its tail length. A filter with more phases would be able to model a longer echo thereby causing better echo cancellation.

For example, for a 2 y-channels, 3 x-channels, 10 phases per x channel configuration, the filter spectrum phases are stored in a 2x30 array. For a given y channel, say y channel 0, H_hat[0][0] to H_hat[0][9] points to 10 phases of H_hat_{y0x0}, H_hat[0][10] to H_hat[0][19] points to 10 phases of H_hat_{y0x1} and H_hat[0][20] to H_hat[0][29] points to 10 phases of H_hat_{y0x2}.

Each filter phase data which is pointed to by H_hat[i][j].data is stored as AEC_FD_FRAME_LENGTH complex 32bit array.

bfp_complex_s32_t `X_fifo_1d`[*AEC_LIB_MAX_PHASES*]

BFP array pointing to all phases of reference input spectrum across all x channels. Here, the reference input spectrum is saved in a 1 dimensional array of phases, with x channel 0 phases followed by x channel 1 phases and so on. For example, for a 2 x-channels, 10 phases per x channel configuration, X_fifo_1d[0] to X_fifo_1d[9] points to the 10 phases for channel 0 and X_fifo[10] to X_fifo[19] points to the 10 phases for channel 1.

Each X data spectrum phase pointed to by X_fifo_1d[i][j].data is stored as length AEC_FD_FRAME_LENGTH complex 32bit array.

bfp_complex_s32_t `T`[*AEC_LIB_MAX_X_CHANNELS*]

BFP array pointing to T values which are stored as a length AEC_FD_FRAME_LENGTH, complex array per x channel.

bfp_s32_t `inv_X_energy`[*AEC_LIB_MAX_X_CHANNELS*]

BFP array pointing to the normalisation spectrum which are stored as a length AEC_FD_FRAME_LENGTH, 32bit integer array per x channel.

bfp_s32_t `X_energy`[*AEC_LIB_MAX_X_CHANNELS*]

BFP array pointing to the X_energy data which is the energy per bin of the X spectrum summed over all phases of the X data. X_energy data is stored as a length AEC_FD_FRAME_LENGTH, integer 32bit array per x channel.

bfp_s32_t `overlap`[*AEC_LIB_MAX_Y_CHANNELS*]

BFP array pointing to time domain overlap data values which are used in the overlap add operation done while calculating the echo canceller time domain output. Stored as a length 32, 32 bit integer array per y channel.

bfp_s32_t `y_hat`[*AEC_LIB_MAX_Y_CHANNELS*]

BFP array pointing to the time domain estimated mic signal. Stored as length AEC_PROC_FRAME_LENGTH, 32 bit integer array per y channel.

bfp_s32_t `error`[*AEC_LIB_MAX_Y_CHANNELS*]

BFP array pointing to the time domain adaptive filter error signal. Stored as length AEC_PROC_FRAME_LENGTH, 32 bit integer array per y channel.

float_s32_t `mu`[*AEC_LIB_MAX_Y_CHANNELS*][*AEC_LIB_MAX_X_CHANNELS*]

mu values for every x-y pair stored as 32 bit integer mantissa and 32 bit integer exponent

float_s32_t `error_ema_energy`[*AEC_LIB_MAX_Y_CHANNELS*]

Exponential moving average of the time domain adaptive filter error signal energy. Stored in an x channels array with every value stored as a 32bit integer mantissa and exponent.

float_s32_t `overall_Error`[*AEC_LIB_MAX_Y_CHANNELS*]

Energy of the adaptive filter error spectrum. Stored in a y channels array with every value stored as a 32bit integer mantissa and exponent.

float_s32_t `max_X_energy`[*AEC_LIB_MAX_X_CHANNELS*]

Maximum X energy across all values of X_energy for a given x channel. Stored in an x channels array with every value stored as a 32bit integer mantissa and exponent.

float_s32_t `delta_scale`

fractional regularisation scalefactor.

float_s32_t `delta`

delta parameter used in the normalisation spectrum calculation.

aec_shared_state_t *`shared_state`

pointer to the state data shared between main and shadow filter.

unsigned `num_phases`

Number of filter phases per x-y pair that AEC filter is configured for. This is the input argument num_main_filter_phases or num_shadow_filter_phases, depending on which filter the *aec_state_t* is instantiated for, passed in *aec_init()* call.

AEC #define constants

group `aec_defines`

Defines

`AEC_LIB_MAX_Y_CHANNELS`

Maximum number of microphone input channels supported in the library. Microphone input to the AEC refers to the input from the device's microphones from which AEC removes the echo created in the room by the device's loudspeakers.

AEC functions follow the convention of using y and Y for referring to time domain and frequency domain representation of microphone input.

The `num_y_channels` passed into *aec_init()* call should be less than or equal to AEC_LIB_MAX_Y_CHANNELS. This define is only used for defining data structures in the aec_state. The library code implementation uses only the num_y_channels aec is initialised for in the *aec_init()* call.

`AEC_LIB_MAX_X_CHANNELS`

Maximum number of reference input channels supported in the library. Reference input to the AEC refers to a copy of the device's speaker output audio that is also sent as an input to the AEC. It is used to model the echo characteristics between a mic-loudspeaker pair.

AEC functions follow the convention of using x and X for referring to time domain and frequency domain representation of reference input.

The `num_x_channels` passed into *aec_init()* call should be less than or equal to AEC_LIB_MAX_X_CHANNELS. This define is only used for defining data structures in the aec_state. The library code implementation uses only the num_x_channels aec is initialised for in the *aec_init()* call.

`AEC_FRAME_ADVANCE`

AEC frame size This is the number of samples of new data that the AEC works on every frame. 240 samples at 16kHz is 15msec. Every frame, the echo canceller takes in 15msec of mic and reference data and generates 15msec of echo cancelled output.

`AEC_PROC_FRAME_LENGTH`

Time domain samples block length used internally in AEC's block LMS algorithm

`AEC_FD_FRAME_LENGTH`

Number of bins of spectrum data computed when doing a DFT of a AEC_PROC_FRAME_LENGTH length time domain vector. The AEC_FD_FRAME_LENGTH spectrum values represent the bins from DC to Nyquist.

`AEC_LIB_MAX_PHASES`

Maximum total number of phases supported in the AEC library This is the maximum number of total phases supported in the AEC library. Total phases are calculated by summing phases across adaptive filters for all x-y pairs.

For example. for a 2 y-channels, 2 x-channels, 10 phases per x channel configuration, there are 4 adaptive filters, H_hat_{y0x0}, H_hat_{y0x1}, H_hat_{y1x0} and H_hat_{y1x1}, each filter having 10 phases, so the total number of phases is 40. When *aec_init()* is called to initialise the AEC, the num_y_channels, num_x_channels and num_main_filter_phases parameters passed in should be such that num_y_channels * num_x_channels * num_main_filter_phases is less than equal to AEC_LIB_MAX_PHASES.

This define is only used when defining data structures within the AEC state structure. The AEC algorithm implementation uses the num_main_filter_phases and num_shadow_filter_phases values that are passed into *aec_init()*.

`AEC_UNUSED_TAPS_PER_PHASE`

Overlap data length

`AEC_FFT_PADDING`

Extra 2 samples you need to allocate in time domain so that the full spectrum (DC to nyquist) can be stored after the in-place FFT. NOT USER MODIFIABLE.

AEC API

AEC High Level API Functions

group `aec_func`

Functions

void `aec_init`(*aec_state_t* *main_state, *aec_state_t* *shadow_state, *aec_shared_state_t* *shared_state, uint8_t *main_mem_pool, uint8_t *shadow_mem_pool, unsigned num_y_channels, unsigned num_x_channels, unsigned num_main_filter_phases, unsigned num_shadow_filter_phases)

Initialise AEC data structures.

This function initializes AEC data structures for a given configuration. The configuration parameters num_y_channels, num_x_channels, num_main_filter_phases and num_shadow_filter_phases are passed in as input arguments.

This function needs to be called at startup to first initialise the AEC and subsequently whenever the AEC configuration changes.

`main_state`, `shadow_state` and shared_state structures must start at double word aligned addresses.

main_mem_pool and shadow_mem_pool must point to memory buffers big enough to support main and shadow filter processing. AEC state *aec_state_t* and shared state *aec_shared_state_t* structures contain only the BFP data structures used in the AEC. The memory these BFP structures will point to needs to be provided by the user in the memory pool main and shadow filters memory pool. An example memory pool structure is present in aec_memory_pool_t and aec_shadow_filt_memory_pool_t.

main_mem_pool and shadow_mem_pool must also start at double word aligned addresses.

Example

```
#include "aec_memory_pool.h"
aec_state_t DWORD_ALIGNED main_state;
aec_state_t DWORD_ALIGNED shadow_state;
aec_shared_state_t DWORD_ALIGNED aec_shared_state;
uint8_t DWORD_ALIGNED aec_mem[sizeof(aec_memory_pool_t)];
uint8_t DWORD_ALIGNED aec_shadow_mem[sizeof(aec_shadow_filt_memory_pool_t)];
unsigned y_chans = 2, x_chans = 2;
unsigned main_phases = 10, shadow_phases = 5;
// There is one main and one shadow filter per x-y channel pair, so for this␣
↪example there will be 4 main and 4
// shadow filters. Each main filter will have 10 phases and each shadow filter␣
↪will have 5 phases.
aec_init(&main_state, &shadow_state, &shared_state, aec_mem, aec_shadow_mem, y_
↪chans, x_chans, main_phases, shadow_phases);
```

Parameters

- `main_state` – **[inout]** AEC state structure for holding main filter specific state
- `shadow_state` – **[inout]** AEC state structure for holding shadow filter specific state
- `shared_state` – **[inout]** Shared state structure for holding state that is common to main and shadow filter
- `main_mem_pool` – **[inout]** Memory pool containing main filter memory buffers
- `shadow_mem_pool` – **[inout]** Memory pool containing shadow filter memory buffers
- `num_y_channels` – **[in]** Number of mic input channels
- `num_x_channels` – **[in]** Number of reference input channels
- `num_main_filter_phases` – **[in]** Number of phases in the main filter

- `num_shadow_filter_phases` – **[in]** Number of phases in the shadow filter

void `aec_frame_init`(*aec_state_t* *main_state, *aec_state_t* *shadow_state, const int32_t (*y_data)[*AEC_FRAME_ADVANCE*], const int32_t (*x_data)[*AEC_FRAME_ADVANCE*])

Initialise AEC data structures for processing a new frame.

This is the first function that is called when a new frame is available for processing. It takes the new samples as input and combines the new samples and previous frame's history to create a processing block on which further processing happens. It also initialises some data structures that need to be initialised at the beginning of a frame.

Note:

y_data and x_data buffers memory is free to be reused after this function call.

Parameters

- `main_state` – **[inout]** main filter state
- `shadow_state` – **[inout]** shadow filter state
- `y_data` – **[in]** pointer to mic input buffer
- `x_data` – **[in]** pointer to reference input buffer

void `aec_calc_freq_domain_energy`(float_s32_t *fd_energy, const bfp_complex_s32_t *input)

Calculate energy in the spectrum.

This function calculates the energy of frequency domain data used in the AEC. Frequency domain data in AEC is in the form of complex 32bit vectors and energy is calculated as the squared magnitude of the input vector.

Parameters

- `fd_energy` – **[out]** energy of the input spectrum
- `input` – **[in]** input spectrum BFP structure

void `aec_calc_time_domain_ema_energy`(float_s32_t *ema_energy, const bfp_s32_t *input, unsigned start_offset, unsigned length, const *aec_config_params_t* *conf)

Calculate exponential moving average (EMA) energy of a time domain (TD) vector.

This function calculates the EMA energy of AEC time domain data which is in the form of real 32bit vectors.

This function can be called to calculate the EMA energy of subsets of the input vector as well.

Parameters

- `ema_energy` – **[out]** EMA energy of the input
- `input` – **[in]** time domain input BFP structure
- `start_offset` – **[in]** offset in the input vector from where to start calculating EMA energy
- `length` – **[in]** length over which to calculate EMA energy
- `conf` – **[in]** AEC configuration parameters.

void `aec_forward_fft`(bfp_complex_s32_t *output, bfp_s32_t *input)

Calculate Discrete Fourier Transform (DFT) spectrum of an input time domain vector.

This function calculates the spectrum of a real 32bit time domain vector. It calculates an N point real DFT where N is the length of the input vector to output a complex N/2+1 length complex 32bit vector. The N/2+1 complex output values represent spectrum samples from DC up to the Nyquist frequency.

The DFT calculation is done in place. After this function call the input and output BFP structures `data` fields point to the same memory. Since DFT is calculated in place, use of the input BFP struct is undefined after this function.

To allow for inplace transform from N real 32bit values to N/2+1 complex 32bit values, the input vector should have 2 extra real 32bit samples worth of memory. This means that `input->data` should point to a buffer of length `input->length`+2

After this function `input->data` and `output->data` point to the same memory address.

Parameters

- `output` – **[out]** DFT output BFP structure
- `input` – **[in]** DFT input BFP structure

void `aec_inverse_fft`(bfp_s32_t *output, bfp_complex_s32_t *input)

Calculate inverse Discrete Fourier Transform (DFT) of an input spectrum.

This function calculates a N point inverse real DFT of a complex 32bit where N is 2*(length-1) where length is the length of the input vector. The output is a real 32bit vector of length N.

The inverse DFT calculation is done in place. After this operation the input and the output BFP structures `data` fields point to the same memory. Since the calculation is done in place, use of input BFP struct after this function is undefined.

After this function `input->data` and `output->data` point to the same memory address.

Parameters

- `output` – **[out]** inverse DFT output BFP structure
- `input` – **[in]** inverse DFT input BFP structure

void `aec_calc_X_fifo_energy`(*aec_state_t* *state, unsigned ch, unsigned recalc_bin)

Calculate total energy of the X FIFO.

`X FIFO` is a FIFO of the most recent `X` frames, where `X` is spectrum of one frame of reference input. There's a common X FIFO that is shared between main and shadow filters. It holds `num_main_filter_phases` most recent X frames and the shadow filter uses `num_shadow_filter_phases` most recent frames out of it.

This function calculates the energy per X sample index summed across the X FIFO phases. This function also calculates the maximum energy across all samples indices of the output energy vector

---

Note:

This function implements some speed optimisations which introduce quantisation error. To stop quantisation error build up, in every call of this function, energy for one sample index, which is specified in the `recalc_bin` argument, is recalculated without the optimisations. There are a total of

AEC_FD_FRAME_LENGTH samples in the energy vector, so recalc_bin keeps cycling through indexes 0 to AEC_PROC_FRAME_LENGTH/2.

Parameters

- `state` – **[inout]** AEC state. state->X_energy[ch] and state->max_X_energy[ch] are updated
- `ch` – **[in]** channel index for which energy calculations are done
- `recalc_bin` – **[in]** The sample index for which energy is recalculated to eliminate quantisation errors

void `aec_update_X_fifo_and_calc_sigmaXX`(*aec_state_t* *state, unsigned ch)

Update X FIFO with the current X frame.

This function updates the X FIFO by removing the oldest X frame from it and adding the current X frame to it. This function also calculates sigmaXX which is the exponential moving average of the current X frame energy

Parameters

- `state` – **[inout]** AEC state structure. state->shared_state->X_fifo[ch] and state->shared_state->sigma_XX[ch] are updated.
- `ch` – **[in]** X channel index for which to update X FIFO

void `aec_calc_Error_and_Y_hat`(*aec_state_t* *state, unsigned ch)

Calculate error spectrum and estimated mic signal spectrum.

This function calculates the error spectrum (`Error`) and estimated mic input spectrum (`Y_hat`) `Y_hat` is calculated as the sum of all phases of the adaptive filter multiplied by the respective phases of the reference input spectrum. Error is calculated by subtracting `Y_hat` from the mic input spectrum `Y`

Parameters

- `state` – **[inout]** AEC state structure. state->Error[ch] and state->Y_hat[ch] are updated
- `ch` – **[in]** mic channel index for which to compute Error and Y_hat

void `aec_calc_coherence`(*aec_state_t* *state, unsigned ch)

Calculate coherence.

This function calculates the average coherence between mic input signal (`y`) and estimated mic signal (`y_hat`). A metric is calculated using `y` and `y_hat` and the moving average (`coh`) and a slow moving average (`coh_slow`) of that metric is calculated. The coherence values are used to distinguish between situations when filter adaption should continue or freeze and update mu accordingly.

Parameters

- `state` – **[inout]** AEC state structure. `state->shared_state->coh_mu_state[ch].coh` and `state->shared_state->coh_mu_state[ch].coh_slow` are updated
- `ch` – **[in]** mic channel index for which to calculate average coherence

void `aec_calc_output`(*aec_state_t* *state, int32_t (*output)[*AEC_FRAME_ADVANCE*], unsigned ch)

Calculate AEC filter output signal.

This function is responsible for windowing the filter `error` signal and creating AEC filter output that can be propagated to downstream stages. `output` is calculated by overlapping and adding current frame's windowed error signal with the previous frame windowed error. This is done to smooth discontinuities in the output as the filter adapts.

Parameters

- `state` – **[inout]** AEC state structure. `state->error[ch]`
- `output` – **[out]** pointer to the output buffer
- `ch` – **[in]** mic channel index for which to calculate output

void `aec_calc_normalisation_spectrum`(*aec_state_t* *state, unsigned ch, unsigned is_shadow)

Calculate normalisation spectrum.

This function calculates the normalisation spectrum of the reference input signal. This normalised spectrum is later used during filter adaption to scale the adaption to the size of the input signal. The normalisation spectrum is calculated as a time and frequency smoothed energy of the reference input spectrum.

The normalisation spectrum is calculated differently for main and shadow filter, so a flag indicating whether this calculation is being done for the main or shadow filter is passed as an input to the function

Parameters

- `state` – **[inout]** AEC state structure. state->inv_X_energy[ch] is updated
- `ch` – **[in]** reference channel index for which to calculate normalisation spectrum
- `is_shadow` – **[in]** flag indicating filter type. 0: Main filter, 1: Shadow filter

void `aec_compare_filters_and_calc_mu`(*aec_state_t* *main_state, *aec_state_t* *shadow_state)

Compare and update filters. Calculate the adaption step size mu.

This function has 2 responsibilities. First, it compares the energies in the error spectrums of the main and shadow filter with each other and with the mic input spectrum energy, and makes an estimate of how well the filters are performing. Based on this, it optionally modifies the filters by either resetting the filter coefficients or copying one filter into another. Second, it uses the coherence values calculated in aec_calc_coherence as well as information from filter comparison done in step 1 to calculate the adaption step size mu.

Parameters

- `main_state` – **[inout]** AEC state structure for the main filter
- `shadow_state` – **[inout]** AEC state structure for the shadow filter

void `aec_calc_T`(*aec_state_t* *state, unsigned y_ch, unsigned x_ch)

Calculate the parameter `T`

This function calculates a parameter referred to as `T` that is later used to scale the reference input spectrum in the filter update step. `T` is a function of the adaption step size `mu`, normalisation spectrum `inv_X_energy` and the filter error spectrum `Error`.

Parameters

- `state` – **[inout]** AEC state structure. `state->T[x_ch]` is updated
- `y_ch` – **[in]** mic channel index
- `x_ch` – **[in]** reference channel index

void `aec_filter_adapt`(*aec_state_t* *state, unsigned y_ch)

Update filter.

This function updates the adaptive filter spectrum (H_hat). It calculates the delta update that is applied to the filter by scaling the X FIFO with the T values computed in `aec_compute_T()` and applies the delta update to `H_hat`. A gradient constraint FFT is then applied to constrain the length of each phase of the filter to avoid wrapping when calculating `y_hat`

Parameters

- `state` – **[inout]** AEC state structure. `state->H_hat[y_ch]` is updated
- `y_ch` – **[in]** mic channel index

void `aec_update_X_fifo_1d`(*aec_state_t* *state)

Update the X FIFO alternate BFP structure.

The X FIFO BFP structure is maintained in 2 forms - as a 2 dimensional [x_channels][num_phases] and as a [x_channels * num_phases] 1 dimensional array. This is done in order to optimally access the X FIFO as needed in different functions. After the X FIFO is updated with the current X frame, this function is called in order to copy the 2 dimensional BFP structure into it's 1 dimensional counterpart.

Parameters

- `state` – **[inout]** AEC state structure. `state->X_fifo_1d` is updated

float_s32_t `aec_calc_corr_factor`(*aec_state_t* *state, unsigned ch)

Calculate a correlation metric between the microphone input and estimated microphone signal.

This function calculates a metric of resemblance between the mic input and the estimated mic signal. The correlation metric, along with reference signal energy is used to infer presence of near and far end signals in the AEC mic input.

Parameters

- `state` – **[in]** AEC state structure. `state->y` and `state->y_hat` are used to calculate the correlation metric
- `ch` – **[in]** mic channel index for which to calculate the metric

Returns

correlation metric in float_s32_t format

float_s32_t `aec_calc_max_input_energy`(const int32_t (*input_data)[*AEC_FRAME_ADVANCE*], int num_channels)

Calculate the energy of the input signal.

This function calculates the sum of the energy across all samples of the time domain input channel and returns the maximum energy across all channels.

Parameters

- `input_data` – **[in]** Pointer to the input data buffer. The input is assumed to be in Q1.31 fixed point format.
- `num_channels` – **[in]** Number of input channels.

Returns

Maximum energy in float_s32_t format.

void `aec_reset_state`(*aec_state_t* *main_state, *aec_state_t* *shadow_state)

Reset parts of aec state structure.

This function resets parts of AEC state so that the echo canceller starts adapting from a zero filter.

Parameters

- `pointer` – **[in]** to AEC main filter state structure.
- `pointer` – **[in]** to AEC shadow filter state structure

uint32_t `aec_detect_input_activity`(const int32_t (*input_data)[*AEC_FRAME_ADVANCE*], float_s32_t active_threshold, int32_t num_channels)

Detect activity on input channels.

This function implements a quick check for detecting activity on the input channels. It detects signal presence by checking if the maximum sample in the time domain input frame is above a given threshold.

Parameters

- `input_data` – **[in]** Pointer to input data frame. Input is assumed to be in Q1.31 fixed point format.
- `active_threshold` – **[in]** Threshold for detecting signal activity
- `num_channels` – **[in]** Number of input data channels

Returns

0 if no signal activity on the input channels, 1 if activity detected on the input channels

AEC Low Level API Functions (STILL WIP)

group `aec_low_level_func`

Functions

void `aec_l2_calc_Error_and_Y_hat`(bfp_complex_s32_t *Error, bfp_complex_s32_t *Y_hat, const bfp_complex_s32_t *Y, const bfp_complex_s32_t *X_fifo, const bfp_complex_s32_t *H_hat, unsigned num_x_channels, unsigned num_phases, unsigned start_offset, unsigned length, int32_t bypass_enabled)

Calculate Error and Y_hat for a channel over a range of bins.

void `aec_l2_adapt_plus_fft_gc`(bfp_complex_s32_t *H_hat_ph, const bfp_complex_s32_t *X_fifo_ph, const bfp_complex_s32_t *T_ph)

Adapt one phase of the adaptive filter.

void `aec_l2_bfp_complex_s32_unify_exponent`(bfp_complex_s32_t *chunks, int32_t *final_exp, uint32_t *final_hr, const uint32_t *mapping, uint32_t array_len, uint32_t desired_index, uint32_t min_headroom)

Unify bfp_complex_s32_t chunks into a single exponent and headroom.

void `aec_l2_bfp_s32_unify_exponent`(bfp_s32_t *chunks, int32_t *final_exp, uint32_t *final_hr, const uint32_t *mapping, uint32_t array_len, uint32_t desired_index, uint32_t min_headroom)

Unify bfp_s32_t chunks into a single exponent and headroom.

lib_aec Header Files

aec_defines.h

page `page_aec_defines_h`

This header contains lib_aec public defines

aec_state.h

page `page_aec_state_h`

This header contains definitions for data structures and enums used in lib_aec.

aec_api.h

page `page_aec_api_h`

lib_aec public functions API.

On GitHub

`lib_aec` is present as part of `fwk_voice`. Get the latest version of `fwk_voice` from `https://github.com/xmos/fwk_voice`. `lib_aec` is present within the *modules/lib_aec* directory in `fwk_voice`

API

To use the functions in this library in an application, include *aec_api.h* in the application source file

1.1.2 Noise Suppression Library

`lib_ns` is a library which performs Noise Suppression (NS), by estimating the noise and subtracting it from frame. `lib_ns` library functions make use of functionality provided in `lib_xcore_math` to perform DSP operations. For more details, refer to *NS Overview*.

Repository Structure

- `modules/lib_ns` - The actual `lib_ns` library directory within `https://github.com/xmos/fwk_voice/`. Within `lib_ns`
  - `api/` - Headers containing the public API for `lib_ns`.
  - `doc/` - Library documentation source (for non-embedded documentation) and build directory.
  - `src/` - Library source code.

Requirements

`lib_ns` is included as part of the `fwk_voice` github repository and all requirements for cloning and building `fwk_voice` apply. `lib_ns` is compiled as a static library as part of the overall `fwk_voice` build. It depends on lib_xcore_math.

Getting and Building

This module is part of the parent `fwk_voice` repo clone. It is compiled as a static library as part of `fwk_voice` compilation process.

To include `lib_ns` in an application as a static library, the generated `libfwk_voice_module_lib_ns.a` can then be linked into the application. Add `lib_ns/api` to the include directories when building the application.

NS Overview

The `lib_ns` library provides an API to implement Noise Suppression within an application.

The noise suppressor estimates the probability of speech presence and dynamically adapts its coefficients to estimate the noise levels to subtract from the input. The filter will automatically reset its noise estimations every 10 frames.

The NS takes as input a frame of data from an audio channel. This could be the microphone input or the output of another module in the application.

Noise Suppression is performed on a frame-by-frame basis. Each frame consists of 15ms of data, which is 240 samples at 16kHz input sampling frequency. Input data is expected to be in a fixed-point 32-bit 1.31 format.

Before processing any frames, the application must configure and initialise the NS instance by calling `ns_init()`. Then for each frame, `ns_process_frame()` will update the NS instance's internal state and produce the output frame by applying the NS algorithm to the input frame.

If multiple channels need to be processed by the application, or multiple outputs are required, an independent instance of the NS must be run for each channel.

API Reference

NS API Functions

group `ns_func`

Functions

void `ns_init`(*ns_state_t* *ns)

Initialise the NS.

This function initialises the NS state with the provided configuration. It must be called at startup to initialise the NS before processing any frames, and can be called at any time after that to reset the NS instance, returning the internal NS state to its defaults.

Example

```
ns_state_t ns;
ns_init(&ns);
```

Parameters

- `ns` – **[out]** NS state structure

void `ns_process_frame`(*ns_state_t* *ns, int32_t output[*NS_FRAME_ADVANCE*], const int32_t input[*NS_FRAME_ADVANCE*])

Perform NS processing on a frame of input data.

This function updates the NS's internal state based on the input 1.31 frame, and returns an output 1.31 frame containing the result of the NS algorithm applied to the input.

The `input` and `output` pointers can be equal to perform the processing in-place.

Example

```
int32_t input[NS_FRAME_ADVANCE];
int32_t output[NS_FRAME_ADVANCE];
ns_state_t ns;
ns_init(&ns);
ns_process_frame(&ns, output, input);
```

Parameters

- `ns` – **[inout]** NS state structure
- `output` – **[out]** Array to return the resulting frame of data
- `input` – **[in]** Array of frame data on which to perform the NS

NS API Structure Definitions

group `ns_defs`

Defines

`NS_FRAME_ADVANCE`

Length of the frame of data on which the NS will operate.

`NS_PROC_FRAME_LENGTH`

Time domain samples block length used internally.

`NS_PROC_FRAME_BINS`

Number of bins of spectrum data computed when doing a DFT of a NS_PROC_FRAME_LENGTH length time domain vector. The NS_PROC_FRAME_BINS spectrum values represent the bins from DC to Nyquist.

`NS_INT_EXP`

The exponent used internally to keep q1.31 format.

`NS_WINDOW_LENGTH`

The length of the window applied in time domain

struct `ns_state_t`

#include <ns_state.h> NS state structure.

This structure holds the current state of the NS instance and members are updated each time that *`ns_process_frame()`* runs. Many of these members are exponentially-weighted moving averages (EWMA) which influence the behaviour of the NS filter. The user should not directly modify any of these members.

Public Members

bfp_s32_t `S`
: BFP structure to hold the local energy.

bfp_s32_t `S_min`
: BFP structure to hold the minimum local energy within 10 frames.

bfp_s32_t `S_tmp`
: BFP structure to hold the temporary local energy.

bfp_s32_t `p`
: BFP structure to hold the conditional signal presence probability

bfp_s32_t `alpha_d_tilde`
: BFP structure to hold the time-varying smoothing parameter.

bfp_s32_t `lambda_hat`
: BFP structure to hold the noise estimation.

int32_t `data_S`[*NS_PROC_FRAME_BINS*]
: int32_t array to hold the data for S.

int32_t `data_S_min`[*NS_PROC_FRAME_BINS*]
: int32_t array to hold the data for S_min.

int32_t `data_S_tmp`[*NS_PROC_FRAME_BINS*]
: int32_t array to hold the data for S_tmp.

int32_t `data_p`[*NS_PROC_FRAME_BINS*]
: int32_t array to hold the data for p.

int32_t `data_adt`[*NS_PROC_FRAME_BINS*]
: int32_t array to hold the data for alpha_d_tilde.

int32_t `data_lambda_hat`[*NS_PROC_FRAME_BINS*]
: int32_t array to hold the data for lambda_hat.

bfp_s32_t `prev_frame`
: BFP structure to hold the previous frame.

bfp_s32_t `overlap`
: BFP structure to hold the overlap.

bfp_s32_t `wind`
: BFP structure to hold the first part of the window.

bfp_s32_t `rev_wind`
: BFP structure to hold the second part of the window.

int32_t `data_prev_frame`[*NS_PROC_FRAME_LENGTH* - *NS_FRAME_ADVANCE*]
: int32_t array to hold the data for prev_frame.

int32_t `data_ovelap`[*NS_FRAME_ADVANCE*]
: int32_t array to hold the data for overlap.

int32_t `data_rev_wind`[*NS_WINDOW_LENGTH* / 2]
: int32_t array to hold the data for rev_wind.

float_s32_t `delta`
: EWMA of the energy ratio to calculate p.

float_s32_t `alpha_d`
: EWMA of the smoothing parameter for alpha_d_tilde.

float_s32_t `alpha_s`
: EWMA of the smoothing parameter for S.

float_s32_t `alpha_p`
: EWMA of the smoothing parameter for p.

float_s32_t `one_minus_aplha_d`
: EWMA of the 1 - alpha_d parameter.

float_s32_t `one_minus_alpha_s`
: EWMA of the 1 - alpha_s parameter.

float_s32_t `one_minus_alpha_p`
: EWMA of the 1 - alpha_p parameter.

unsigned `reset_period`
: Filter reset period value for auto-reset.

unsigned `reset_counter`
: Filter reset counter.

NS Header Files

ns_api.h

page `page_ns_api_h`

This header should be included in application source code to gain access to the lib_ns public functions API.

ns_state.h

page `page_ns_state_h`

This header contains definitions for data structure and defines.

This header is automatically included by `ns_api.h`

On GitHub

`lib_ns` is present as part of `fwk_voice`. Get the latest version of `fwk_voice` from `https://github.com/xmos/fwk_voice`. `lib_ns` is present within the *modules/lib_ns* directory in `fwk_voice`.

API

To use the functions in this library in an application, include *ns_api.h* in the application source file.

1.1.3 Automatic Gain Control Library

`lib_agc` is a library which performs Automatic Gain Control (AGC), with support for Loss Control. For more details, refer to *AGC Overview*.

Repository Structure

- `modules/lib_agc` - The actual `lib_agc` library directory within `https://github.com/xmos/fwk_voice/`. Within `lib_agc`
  - `api/` - Headers containing the public API for `lib_agc`.
  - `doc/` - Library documentation source (for non-embedded documentation) and build directory.
  - `src/` - Library source code.

Requirements

`lib_agc` is included as part of the `fwk_voice` github repository and all requirements for cloning and building `fwk_voice` apply. `lib_agc` is compiled as a static library as part of the overall `fwk_voice` build. It depends on lib_xcore_math.

Getting and Building

This module is part of the parent `fwk_voice` repo clone. It is compiled as a static library as part of `fwk_voice` compilation process.

To include `lib_agc` in an application as a static library, the generated `libfwk_voice_module_lib_agc.a` can then be linked into the application. Add `lib_agc/api` to the include directories when building the application.

AGC Overview

The `lib_agc` library provides an API to implement Automatic Gain Control within an application. The goal of the AGC algorithm is to provide consistent output levels for voice audio.

The gain control can adapt to maintain the amplitude of the peak of the frame within an upper and lower bound configured for the AGC instance. When used in an application with a Voice to Noise Ratio estimator (VNR), the AGC will adapt only when voice activity is detected, so that speech in the input signal is amplified above other sounds.

The AGC also has a Loss Control feature which can be used when the application has an Acoustic Echo Canceller (AEC). This feature uses data from the AEC to adjust the gain applied to reduce residual echoes by attenuating the audio when near-end speech is not present.

The AGC takes as input a frame of data from an audio channel. This could be the microphone input or the output of another module in the application.

Gain control is performed on a frame-by-frame basis. Each frame consists of 15ms of data, which is 240 samples at 16kHz input sampling frequency. Input data is expected to be in a fixed-point 32-bit 1.31 format.

Before processing any frames, the application must configure and initialise the AGC instance by calling `agc_init()`. Then for each frame, `agc_process_frame()` will update the AGC instance's internal state and produce the output frame by applying the AGC algorithm to the input frame.

The gain values in this module for AGC gain and Loss Control gain are multiplicative factors that are applied to scale the input frame. Therefore, a fixed gain value of 1.0 (without loss control) will create no change to the input.

If multiple channels need to be processed by the application, or multiple outputs are required, an independent instance of the AGC must be run for each channel.

API Reference

AGC API Functions

group `agc_func`

Functions

void `agc_init`(*agc_state_t* *agc, *agc_config_t* *config)

Initialise the AGC.

This function initialises the AGC state with the provided configuration. It must be called at startup to initialise the AGC before processing any frames, and can be called at any time after that to reset the AGC instance, returning the internal AGC state to its defaults.

Example with an unmodified profile

```
agc_state_t agc;
agc_init(&agc, &AGC_PROFILE_ASR);
```

Example with modification to the profile

```
agc_config_t conf = AGC_PROFILE_FIXED_GAIN;
conf.gain = f32_to_float_s32(100);
agc_state_t agc;
agc_init(&agc, &conf);
```

Parameters

- `agc` – **[out]** AGC state structure
- `config` – **[in]** Initial configuration values

void `agc_process_frame`(*agc_state_t* *agc, int32_t output[*AGC_FRAME_ADVANCE*], const int32_t input[*AGC_FRAME_ADVANCE*], *agc_meta_data_t* *meta_data)

Perform AGC processing on a frame of input data.

This function updates the AGC's internal state based on the input frame and meta-data, and returns an output containing the result of the AGC algorithm applied to the input.

The `input` and `output` pointers can be equal to perform the processing in-place.

Example

```
int32_t input[AGC_FRAME_ADVANCE];
int32_t output[AGC_FRAME_ADVANCE];
agc_meta_data md;
md.vnr_flag = AGC_META_DATA_NO_VNR;
md.aec_ref_power = AGC_META_DATA_NO_AEC;
md.aec_corr_factor = AGC_META_DATA_NO_AEC;
agc_process_frame(&agc, output, input, &md);
```

Parameters

- `agc` – **[inout]** AGC state structure
- `output` – **[out]** Array to return the resulting frame of data
- `input` – **[in]** Array of frame data on which to perform the AGC
- `meta_data` – **[in]** Meta-data structure with VNR/AEC data

AGC Pre-Defined Profiles

group `agc_profiles`

Defines

`AGC_PROFILE_ASR`

AGC profile tuned for Automatic Speech Recognition (ASR).

`AGC_PROFILE_FIXED_GAIN`

AGC profile tuned to apply a fixed gain.

AGC API Structure Definitions

group `agc_defs`

Defines

`AGC_FRAME_ADVANCE`

Length of the frame of data on which the AGC will operate.

`AGC_META_DATA_NO_VNR`

If the application has no VNR, `adapt_on_vnr` must be disabled in the configuration. This pre-processor definition can be assigned to the `vnr_flag` in *agc_meta_data_t* in that situation to make it clear in the code that there is no VNR.

`AGC_META_DATA_NO_AEC`

If the application has no AEC, `lc_enabled` must be disabled in the configuration. This pre-processor definition can be assigned to the `aec_ref_power` and `aec_corr_factor` in *agc_meta_data_t* in that situation to make it clear in the code that there is no AEC.

struct `agc_config_t`

#include <agc_api.h> AGC configuration structure.

This structure contains configuration settings that can be changed to alter the behaviour of the AGC instance.

Members with the "lc_" prefix are parameters for the Loss Control feature.

Public Members

int `adapt`

Boolean to enable AGC adaption; if enabled, the gain to apply will adapt based on the peak of the input frame and the upper/lower threshold parameters.

int `adapt_on_vnr`

Boolean to enable adaption based on the VNR meta-data; if enabled, adaption will always be performed when voice activity is detected. This must be disabled if the application doesn't have a VNR.

int `soft_clipping`
: Boolean to enable soft-clipping of the output frame.

float_s32_t `gain`
: The current gain to be applied, not including loss control.

float_s32_t `max_gain`
: The maximum gain allowed when adaption is enabled.

float_s32_t `min_gain`
: The minimum gain allowed when adaption is enabled.

float_s32_t `upper_threshold`
: The upper limit for the gained peak of the frame when adaption is enabled.

float_s32_t `lower_threshold`
: The lower limit for the gained peak of the frame when adaption is enabled.

float_s32_t `gain_inc`
: Factor by which to increase the gain during adaption.

float_s32_t `gain_dec`
: Factor by which to decrease the gain during adaption.

int `lc_enabled`
: Boolean to enable loss control. This must be disabled if the application doesn't have an AEC.

int `lc_n_frame_far`
: Number of frames required to consider far-end audio active.

int `lc_n_frame_near`
: Number of frames required to consider near-end audio active.

float_s32_t `lc_corr_threshold`
: Threshold for far-end correlation above which to indicate far-end activity only.

float_s32_t `lc_bg_power_gamma`
: Gamma coefficient for estimating the power of the far-end background noise.

float_s32_t `lc_gamma_inc`
: Factor by which to increase the loss control gain when less than target value.

float_s32_t `lc_gamma_dec`
: Factor by which to decrease the loss control gain when greater than target value.

float_s32_t `lc_far_delta`

Delta multiplier used when only far-end activity is detected.

float_s32_t `lc_near_delta`

Delta multiplier used when only near-end activity is detected.

float_s32_t `lc_near_delta_far_active`

Delta multiplier used when both near-end and far-end activity is detected.

float_s32_t `lc_gain_max`

Loss control gain to apply when near-end activity only is detected.

float_s32_t `lc_gain_double_talk`

Loss control gain to apply when double-talk is detected.

float_s32_t `lc_gain_silence`

Loss control gain to apply when silence is detected.

float_s32_t `lc_gain_min`

Loss control gain to apply when far-end activity only is detected.

struct `agc_state_t`

#include <agc_api.h> AGC state structure.

This structure holds the current state of the AGC instance and members are updated each time that *`agc_process_frame()`* runs. Many of these members are exponentially-weighted moving averages (EWMA) which influence the adaption of the AGC gain or the loss control feature. The user should not directly modify any of these members, except the config.

Public Members

agc_config_t `config`

The current configuration of the AGC. Any member of this configuration structure can be modified and that change will take effect on the next run of *`agc_process_frame()`*.

float_s32_t `x_slow`

EWMA of the frame peak, which is used to identify the overall trend of a rise or fall in the input signal.

float_s32_t `x_fast`

EWMA of the frame peak, which is used to identify a rise or fall in the peak of frame.

float_s32_t `x_peak`

EWMA of `x_fast`, which is used when adapting to the *`agc_config_t::upper_threshold`*.

int `lc_t_far`

Timer counting down until enough frames with far-end activity have been processed.

int `lc_t_near`

Timer counting down until enough frames with near-end activity have been processed.

float_s32_t `lc_near_power_est`

EWMA of estimates of the near-end power.

float_s32_t `lc_far_power_est`

EWMA of estimates of the far-end power.

float_s32_t `lc_near_bg_power_est`

EWMA of estimates of the power of near-end background noise.

float_s32_t `lc_gain`

Loss control gain applied on top of the AGC gain in *agc_config_t*.

float_s32_t `lc_far_bg_power_est`

EWMA of estimates of the power of far-end background noise.

float_s32_t `lc_corr_val`

EWMA of the far-end correlation for detecting double-talk.

struct `agc_meta_data_t`

#include <agc_api.h> AGC meta data structure.

This structure holds meta-data about the current frame to be processed, and must be updated to reflect the current frame before calling *agc_process_frame()*.

Public Members

int `vnr_flag`

Boolean to indicate the detection of voice activity in the current frame.

float_s32_t `aec_ref_power`

The power of the most powerful reference channel.

float_s32_t `aec_corr_factor`

Correlation factor between the microphone input and the AEC's estimated microphone signal.

AGC Header Files

agc_api.h

page `page_agc_api_h`

This header should be included in application source code to gain access to the lib_agc public functions API.

agc_profiles.h

page `page_agc_profiles_h`

This header contains pre-defined profiles for AGC configurations. These profiles can be used to initialise the *agc_config_t* data for use with *agc_init()*.

This header is automatically included by `agc_api.h`.

On GitHub

`lib_agc` is present as part of `fwk_voice`. Get the latest version of `fwk_voice` from `https://github.com/xmos/fwk_voice`. `lib_agc` is present within the *modules/lib_agc* directory in `fwk_voice`.

API

To use the functions in this library in an application, include *agc_api.h* in the application source file.

1.1.4 Automatic Delay Estimation and Correction Library

`lib_adec` is a library which provides functions for measuring and correcting delay offsets between the reference and loudspeaker signals. `lib_adec` depends on `lib_aec` and `lib_xcore_math` libraries. For more details about the ADEC, refer to *ADEC Overview*

Repository Structure

- `modules/lib_adec` - The actual `lib_adec` library directory within `https://github.com/xmos/fwk_voice/`. Within `lib_adec`
  - `api/` - Headers containing the public API for `lib_adec`.
  - `doc/` - Library documentation source (for non-embedded documentation) and build directory.
  - `src/` - Library source code.

Getting and Building

`lib_adec` is included as part of the `fwk_voice` github repository and all requirements for cloning and building `fwk_voice` apply. `lib_adec` is compiled as a static library as part of overall `fwk_voice` build. To include `lib_adec` in an application as a static library, the generated `libfwk_voice_module_lib_adec.a` can then be linked into the application. Be sure to also add `lib_adec/api` as an include directory for the application.

ADEC Overview

The ADEC module provides functions to estimate and automatically correct for delay offsets between the reference and the loudspeakers.

Acoustic echo cancellation is an adaptive filtering process which compares the reference audio to that received from the microphones. It models the reverberation time of a room, i.e. the time it takes for acoustic reflections to decay to insignificance. The time window modelled by the AEC is finite, and to maximise its performance it is important to ensure that the reference audio is presented to the AEC time aligned to the audio being reproduced by the loudspeakers. The reference audio path delay and the audio reproduction path delay may be significantly different, requiring additional delay to be inserted into one of the two paths, to correct this delay difference.

The ADEC module provides functionality for

- Measuring the current delay
- Using the measured delay along with AEC performance related metadata collected from the echo canceller to monitor AEC and make decisions about reconfiguring the AEC and correcting bulk delay offsets.

The metadata collected from AEC contains statistics such as the ERLE, the peak power seen in the adaptive filter and the peak power to average power ratio of the adaptive filter.

The ADEC algorithm works in 2 modes - normal mode and delay estimation mode. In its normal mode ADEC monitors the AEC performance and requests small delay corrections. Using the statistics from the AEC, the ADEC estimates a metric called the AEC goodness which is an estimate of how well the echo canceller is performing. Based on the estimated AEC goodness and the current measured delay, the ADEC can request for a delay correction to be applied at the input of the echo canceller.

If the AEC is seen as consistently bad, the ADEC transitions to a delay estimation mode and requests for

- A special delay to be applied at AEC input that will enable measuring the actual delay in both delay scenarios; microphone input arriving at the AEC earlier in time than the reference input as well as microphone input arriving late in time wrt reference input.
- A restart of AEC in a new configuration that has more adaptive filter phases, in order of have a longer filter tail length that is suitable for delay estimation.

Once the ADEC has a measure of the new delay, it requests a delay correction and a reconfiguration of the AEC back to its normal mode and goes back to its normal mode of monitoring AEC performance and correcting for small delay offsets.

Before processing any frames, the application must configure and initialise the ADEC instance by calling adec_init(). Then for each frame, adec_estimate_delay() will estimate the current delay and adec_process_frame() will use the current frame's AEC statistics and the estimated delay to monitor the AEC and request possible AEC and delay configuration changes.

API Reference

ADEC API Functions

group `adec_func`

Functions

void `adec_init`(*adec_state_t* *state, *adec_config_t* *config)

Initialise ADEC data structures.

This function initialises ADEC state for a given configuration. It must be called at startup to initialise the ADEC data structures before processing any frames, and can be called at any time after that to reset the ADEC instance, returning the internal ADEC state to its defaults.

Example with ADEC configured for delay estimation only at startup

```
adec_state_t adec_state;
adec_config_t adec_conf;
adec_conf.bypass = 1; // Bypass automatic DE correction
adec_conf.force_de_cycle_trigger = 1; // Force a delay correction cycle, so␣
↪that delay correction happens once after initialisation
adec_init(&adec_state, &adec_conf);
// Application needs to ensure that adec_state->adec_config.force_de_cycle_
↪trigger is set to 0 after ADEC has requested a transition to delay␣
↪estimation mode once in order to ensure that delay is corrected only at␣
↪startup.
```

Example with ADEC configured for automatic delay estimation and correction

```
adec_state_t adec_state;
adec_conf.bypass = 0;
adec_conf.force_de_cycle_trigger = 0;
adec_init(&adec_state, &adec_conf);
```

Parameters

- `state` – **[out]** Pointer to ADEC state structure
- `config` – **[in]** Pointer to ADEC configuration structure.

void `adec_process_frame`(*adec_state_t* *state, *adec_output_t* *adec_output, const *adec_input_t* *adec_in)

Perform ADEC processing on an input frame of data.

This function takes information about the latest AEC processed frame and the latest measured delay estimate as input, and decides if a delay correction between input microphone and reference signals is required. If a correction is needed, it outputs a new requested input delay, optionally accompanied with a request for AEC restart in a different configuration. It updates the internal ADEC state structure to reflect the current state of the ADEC process.

Parameters

- `state` – **[inout]** ADEC internal state structure
- `adec_output` – **[out]** ADEC output structure
- `adec_in` – **[in]** ADEC input structure

void `adec_estimate_delay`(*de_output_t* *de_output, const bfp_complex_s32_t *H_hat, unsigned num_phases)

Estimate microphone delay.

This function measures the microphone signal delay wrt the reference signal. It does so by looking for the phase with the peak energy among all AEC filter phases and uses the peak energy phase index as the estimate of the microphone delay. Along with the measured delay, it also outputs information about

the peak phase energy that can then be used to gauge the AEC filter convergence and the reliability of the measured delay.

Parameters

- `de_state` – **[out]** Delay estimator output structure
- `H_hat` – **[in]** bfp_complex_s32_t array storing the AEC filter spectrum
- `Number` – **[in]** of phases in the AEC filter

ADEC #define constants

group `adec_defines`

Defines

`ADEC_PEAK_TO_AVERAGE_HISTORY_DEPTH`

Number of frames far we look back to smooth the peak to average filter power ratio history.

`ADEC_PEAK_LINREG_HISTORY_SIZE`

Number of frames of peak power history we look at while computing AEC goodness metric. Not NOT USER MODIFIABLE.

ADEC Data Structure and Enum definitions

group `adec_types`

Enums

enum `adec_mode_t`

Values:

enumerator `ADEC_NORMAL_AEC_MODE`

ADEC processing mode where it monitors AEC performance and requests small delay correction.

enumerator `ADEC_DELAY_ESTIMATOR_MODE`

ADEC processing mode for bulk delay correction in which it measures for a new delay offset.

struct `adec_config_t`

#include <adec_state.h> ADEC configuration structure.

This is used to provide configuration when initialising ADEC at startup. A copy of this structure is present in the ADEC state structure and available to be modified by the application for run time control of ADEC configuration.

Public Members

int32_t `bypass`

Bypass ADEC decision making process. When set to 1, ADEC evaluates the current input frame metrics but doesn't make any delay correction or aec reset and reconfiguration requests

int32_t `force_de_cycle_trigger`

Force trigger a delay estimation cycle. When set to 1, ADEC bypasses the ADEC monitoring process and transitions to delay estimation mode for measuring delay offset.

struct `de_output_t`

#include <adec_state.h> Delay estimator output structure.

Public Members

int32_t `measured_delay_samples`

Estimated microphone delay in time domain samples.

int32_t `peak_power_phase_index`

Phase index of peak energy AEC filter phase.

float_s32_t `peak_phase_power`

Maximum per phase energy across all AEC filter phases.

float_s32_t `sum_phase_powers`

Sum of filter energy across all filter phases.

float_s32_t `peak_to_average_ratio`

Ratio of peak filter phase energy to average filter phase energy. Used to evaluate how well the filter has converged.

float_s32_t `phase_power`[*AEC_LIB_MAX_PHASES*]

Phase energy of all AEC filter phases.

struct `adec_output_t`

#include <adec_state.h> ADEC output structure.

Public Members

int32_t `delay_change_request_flag`

Flag indicating if ADEC is requesting an input delay correction

int32_t `requested_mic_delay_samples`

Mic delay in samples requested by ADEC. Relevant when delay_change_request_flag is 1. Note that this value is a signed integer. A positive `requested_mic_delay_samples` requires the microphone to be delayed so the application needs to delay the input mic signal by `requested_mic_delay_samples` samples. A negative `requested_mic_delay_samples` means ADEC is requesting the input mic signal to be moved earlier in time. This, the application should do my delaying the input reference signal by `abs(requested_mic_delay_samples)` samples.

int32_t `reset_aec_flag`

flag indicating ADEC's request for a reset of part of the AEC state to get AEC filter to start adapting from a 0 filter. ADEC requests this when a small delay correction needs to be applied that doesn't require a full reset of the AEC.

int32_t `delay_estimator_enabled_flag`

Flag indicating if AEC needs to be run configured in delay estimation mode.

int32_t `requested_delay_samples_debug`

Requested delay samples without clamping to +- MAX_DELAY_SAMPLES. Used only for debugging.

struct `aec_to_adec_t`

#include <adec_state.h> Input structure containing current frame's information from AEC.

Public Members

float_s32_t `y_ema_energy_ch0`

EWMA energy of AEC input mic signal channel 0

float_s32_t `error_ema_energy_ch0`

EWMA energy of AEC filter error output signal channel 0

int32_t `shadow_flag_ch0`

shadow_flag value for the current frame computed within the AEC

struct `adec_input_t`

#include <adec_state.h> ADEC input structure.

Public Members

de_output_t `from_de`

ADEC input from the delay estimator

aec_to_adec_t `from_aec`

ADEC input from AEC

int32_t `far_end_active_flag`

Flag indicating if there is activity on reference input channels.

struct `adec_state_t`

#include <adec_state.h> ADEC state structure.

This structure holds the current state of the ADEC instance and members are updated each time that *`adec_process_frame()`* runs. Many of these members are statistics from tracking the AEC performance. The user should not directly modify any of these members, except the config.

Public Members

float_s32_t `max_peak_to_average_ratio_since_reset`

Maximum peak to average AEC filter phase energy ratio seen since a delay correction was last requested.

float_s32_t `peak_to_average_ratio_history`[*ADEC_PEAK_TO_AVERAGE_HISTORY_DEPTH* + 1]

Last ADEC_PEAK_TO_AVERAGE_HISTORY_DEPTH frames peak_to_average_ratio of phase energies.

float_s32_t `peak_power_history`[*ADEC_PEAK_LINREG_HISTORY_SIZE*]

Last ADEC_PEAK_LINREG_HISTORY_SIZE frames peak phase power.

float_s32_t `aec_peak_to_average_good_aec_threshold`

Threshold was considering peak to average ratio as good.

q8_24 `agm_q24`

AEC goodness metric indicating a measure of how well AEC filter is performing.

q8_24 `erle_bad_bits_q24`

log2 of threshold below which AEC output's measured ERLE is considered bad

q8_24 `erle_good_bits_q24`

log2 of threshold above which AEC output's measured ERLE is considered good

q8_24 `peak_phase_energy_trend_gain_q24`

Multiplier used for scaling agm's sensitivity to peak phase energy trend.

q8_24 `erle_bad_gain_q24`

Multiplier determining how steeply we reduce aec's goodness when measured erle falls below the bad erle threshold.

adec_mode_t `mode`

ADEC's mode of operation. Can be operating in normal AEC or delay estimation mode.

int32_t `gated_milliseconds_since_mode_change`

milliseconds elapsed since a delay change was last requested. Used to ensure that delay corrections are not requested too early without allowing enough time for aec filter to converge.

int32_t `last_measured_delay`

Last measured delay.

int32_t `peak_power_history_idx`

index storing the head of the peak_power_history circular buffer

int32_t `peak_power_history_valid`

Flag indicating whether the peak_power_history buffer has been filled at least once.

int32_t `sf_copy_flag`

Flag indicating if shadow to main filter copy has happened at least once in the AEC.

int32_t `convergence_counter`

Counter indicating number of frames the AEC shadow filter has been attempting to converge.

int32_t `shadow_flag_counter`

Counter indicating number of frame the AEC shadow filter has been better than the main filter.

adec_config_t `adec_config`

ADEC configuration parameters structure. Can be modified by application at run-time to reconfigure ADEC.

ADEC Header Files

adec_defines.h

page `page_adec_defines_h`

This header contains lib_adec public defines

adec_state.h

page `page_adec_state_h`

This header contains definitions for data structures and enums used in lib_adec.

adec_api.h

page `page_adec_api_h`

lib_adec public functions API.

On GitHub

`lib_adec` is present as part of `fwk_voice`. Get the latest version of `fwk_voice` from `https://github.com/xmos/fwk_voice`. `lib_adec` is present within the *modules/lib_adec* directory in `fwk_voice`

API

To use the functions in this library in an application, include *adec_api.h* in the application source file

1.1.5 Interference Canceller Library

`lib_ic` is a library which provides functions that together perform Interference Cancellation (IC) on two channel input mic data by adapting to and modelling the room transfer characteristics. `lib_ic` library functions make use of functionality provided in `lib_aec` for the core normalised LMS blocks which in turn uses `lib_xcore_math` to perform DSP low-level optimised operations. For more details refer to *IC Overview*.

Repository Structure

- `modules/lib_ic` - The actual `lib_ic` library directory within `https://github.com/xmos/fwk_voice/`. Within `lib_ic`:
  - `api/` - Headers containing the public API for `lib_ic`.
  - `doc/` - Library documentation source (for non-embedded documentation) and build directory.
  - `src/` - Library source code.

Requirements

`lib_ic` is included as part of the `fwk_voice` github repository and all requirements for cloning and building `fwk_voice` apply. `lib_ic` is compiled as a static library as part of overall `fwk_voice` build. It depends on `lib_aec` and `lib_xcore_math`.

API Structure

The API is presented as three simple functions. These are initialisation, filtering and adaption. Initialisation is called once at startup and filtering and adaption is called once per frame of samples. The performance requirement is relative low (around 12MIPS) and as such is supplied as a single threaded implementation only.

Getting and Building

This repo is obtained as part of the parent `fwk_voice` repo clone. It is compiled as a static library as part of `fwk_voice` compilation process.

To include `lib_ic` in an application as a static library, the generated `libfwk_voice_module_lib_ic.a` can then be linked into the application. Be sure to also add `lib_ic/api` as an include directory for the application.

IC Overview

The Interference Canceller (IC) suppresses static noise from point sources such as cooker hoods, washing machines, or radios for which there is no reference audio signal available. When the Voice to Noise Ratio estimator (VNR) input indicates the absence of voice, the IC adapts to remove noise from point sources in the environment. When the VNR signal indicates the presence of voice, the IC suspends adaptation which allows the voice source to be passed but maintains suppression of the interfering noise sources which have been previously adapted to.

It can offer much greater, and automatic, cancellation of broad-band noise sources when compared to beam forming techniques.

It is designed to work at a sample rate of 16kHz and has a fixed configuration of two input microphones and a single output channel.

The interference canceller is based on an AEC architecture and attempts to cancel one microphone signal from the other in the absence of voice. In this way, it builds an estimate of the difference in transfer functions between the two microphones for any present noise sources. Since the transfer function includes spatial information about the noise sources, applying this filter to the mic input allows any signals originating from the noise source to be cancelled.

The IC uses an adaptive filter which continually adapts to the acoustic environment to accommodate changes in the room created by events such as doors opening or closing and people moving about. However, it will hold the current transfer function in the presence of voice meaning it does not adapt to desired audio sources, which can be a person speaking.

The cancellation is performed on a frame by frame basis. Each frame is made of 15msec chunks of data, which is 240 new samples at 16kHz input sampling frequency, per input channel. This is combined with previous audio data to form a 512 sample frame which allows for sufficient overlap for effective operation of the filter.

The first channel of input microphone data is referred to as `y` when in time domain and `Y` when in frequency domain. The second channel of input microphone data is referred to as `x` when in time domain and `X` when in frequency domain. The y signal is effectively used as the signal containing noise that needs to be cancelled and the x signal is the reference from which the transfer function is estimated and consequently the noise signal estimated before it is subtracted from y.

In general throughout the code, names starting with lower case represent time domain and those beginning with upper case represent frequency domain. For example `error` is the filter error and `Error` is the spectrum of the filter error. The filter coefficient array referred to as `h_hat` in time domain and `H_hat` in frequency domain.

The filter has multiple phases each of 15ms. The term phases refers to the tail length of the filter. A filter with more phases or a longer tail length will be able to model a more reverberant room response leading to better interference cancellation but, as with all normalised LMS based architectures, will be slower to converge in the case of a transfer function change.

Before starting the IC processing the user must call ic_init() to initialise the IC. If the configuration parameters are to be set to non-defaults please modify these after ic_init() or in the *lib_ic API Definitions* file. Once the IC is initialised, the library functions can be called in a order to perform interference cancellation on a frame by frame basis.

API Reference

lib_ic API Functions

group `ic_func`

Functions

int32_t `ic_init`(*ic_state_t* *state)

Initialise IC and VNR data structures and set parameters according to ic_defines.h.

This is the first function that must called after creating an *ic_state_t* instance.

Parameters

- `state` – **[inout]** pointer to IC state structure

Returns

Error status of the VNR inference engine initialisation that is done as part of ic_init. 0 if no error, one of TfLiteStatus error enum values in case of error.

void `ic_filter`(*ic_state_t* *state, int32_t y_data[*IC_FRAME_ADVANCE*], int32_t x_data[*IC_FRAME_ADVANCE*], int32_t output[*IC_FRAME_ADVANCE*])

Filter one frame of audio data inside the IC.

This should be called once per new frame of IC_FRAME_ADVANCE samples. The y_data array contains the microphone data that is to have the noise subtracted from it and x_data is the noise reference source which is internally delayed before being fed into the adaptive filter. Note that the y_data input array is internally delayed by the call to *ic_filter()* and so contains the delayed y_data afterwards. Typically it does not matter which mic channel is connected to x or y_data as long as the separation is appropriate. The performance of this filter has been optimised for a 71mm mic separation distance.

Parameters

- `state` – **[inout]** pointer to IC state structure
- `y_data` – **[inout]** array reference of mic 0 input buffer. Modified during call
- `x_data` – **[in]** array reference of mic 1 input buffer
- `output` – **[out]** array reference containing IC processed output buffer

void `ic_calc_vnr_pred`(*ic_state_t* *state, float_s32_t *input_vnr_pred, float_s32_t *output_vnr_pred)

Calculate voice to noise ratio estimation for the input and output of the IC.

This function can be called after each call to ic_filter. It will calculate voice to noise ratio which can be used to give information to ic_adapt and to the AGC.

Parameters

- `state` – **[inout]** pointer to IC state structure
- `input_vnr_pred` – **[inout]** voice to noise estimate of the IC input
- `output_vnr_pred` – **[inout]** voice to noise estimate of the IC output

void `ic_adapt`(*ic_state_t* *state, float_s32_t vnr)

Adapts the IC filter according to previous frame's statistics and VNR input.

This function should be called after each call to ic_filter. Filter and adapt functions are separated so that the external VNR can operate on each frame.

Parameters

- `state` – **[inout]** pointer to IC state structure
- `vnr` – **[in]** VNR Voice-to-Noise ratio estimation

lib_ic API State Structure

group `ic_state`

Enums

enum `adaption_config_e`

Values:

enumerator `IC_ADAPTION_AUTO`

enumerator `IC_ADAPTION_FORCE_ON`

enumerator `IC_ADAPTION_FORCE_OFF`

enum `control_flag_e`

Values:

enumerator `HOLD`

enumerator `ADAPT`

enumerator `ADAPT_SLOW`

enumerator `UNSTABLE`

enumerator `FORCE_ADAPT`

enumerator `FORCE_HOLD`

struct `ic_config_params_t`

#include <ic_state.h> IC configuration structure.

This structure contains configuration settings that can be changed to alter the behaviour of the IC instance. An instance of this structure is is automatically included as part of the IC state.

It controls the behaviour of the main filter and normalisation thereof. The initial values for these configuration parameters are defined in ic_defines.h and are initialised by *ic_init()*.

Public Members

uint8_t `bypass`

Boolean to control bypassing of filter stage and adaption stage. When set the delayed y audio samples are passed unprocessed to the output. It is recommended to perform an initialisation of the instance after bypass is set as the room transfer function may have changed during that time.

int32_t `gamma_log2`

Up scaling factor for X energy calculation used for normalisation.

uint32_t `sigma_xx_shift`

Down scaling factor for X energy for used for normalisation.

q2_30 `ema_alpha_q30`

Alpha used for calculating error_ema_energy in adapt.

float_s32_t `delta`

Delta value used in denominator to avoid large values when calculating inverse X energy.

struct `ic_adaption_controller_config_t`

#include <ic_state.h> IC adaption controller configuration structure.

This structure contains configuration settings that can be changed to alter the behaviour of the adaption controller. This includes processing of the raw VNR probability input and optional stability controller logic. It is automatically included as part of the IC state and initialised by *ic_init()*.

The initial values for these configuration parameters are defined in ic_defines.h.

Public Members

q2_30 `energy_alpha_q30`

Alpha for EMA input/output energy calculation.

float_s32_t `fast_ratio_threshold`

Fast ratio threshold to detect instability.

float_s32_t `high_input_vnr_hold_leakage_alpha`

Setting of H_hat leakage which gets set if vnr detects high voice probability.

float_s32_t `instability_recovery_leakage_alpha`

Setting of H_hat leakage which gets set if fast ratio exceeds a threshold.

float_s32_t `input_vnr_threshold`

VNR input threshold which decides whether to hold or adapt the filter.

float_s32_t `input_vnr_threshold_high`

VNR high threshold to leak the filter is the speech level is high.

float_s32_t `input_vnr_threshold_low`

VNR low threshold to adapt faster when the speech level is low.

uint32_t `adapt_counter_limit`

Limits number of frames for which mu and leakage_alpha could be adapted.

uint8_t `enable_adaption`

Boolean which controls whether the IC adapts when *ic_adapt()* is called.

adaption_config_e `adaption_config`

Enum which controls the way mu and leakage_alpha are being adjusted.

struct `ic_adaption_controller_state_t`

#include <ic_state.h> IC adaption controller state structure.

This structure contains state used for the instance of the adaption controller logic. It is automatically included as part of the IC state and initialised by *ic_init()*.

Public Members

float_s32_t `input_energy`

EMWA of input frame energy.

float_s32_t `output_energy`

EMWA of output frame energy.

float_s32_t `fast_ratio`

Ratio between output and input EMWA energies.

uint32_t `adapt_counter`

Adaption counter which counts number of frames has been adapted.

control_flag_e `control_flag`

Flag that represents the state of the filter.

ic_adaption_controller_config_t `adaption_controller_config`

Configuration parameters for the adaption controller.

struct `ic_state_t`

#include <ic_state.h> IC state structure.

This is the main state structure for an instance of the Interference Canceller. Before use it must be initialised using the *ic_init()* function. It contains everything needed for the IC instance including configuration and internal state of both the filter, adaption logic and adaption controller.

Public Members

bfp_s32_t `y_bfp`[*IC_Y_CHANNELS*]

BFP array pointing to the time domain y input signal.

bfp_complex_s32_t `Y_bfp`[*IC_Y_CHANNELS*]

BFP array pointing to the frequency domain Y input signal.

int32_t `y`[*IC_Y_CHANNELS*][*IC_FRAME_LENGTH* + *FFT_PADDING*]

Storage for y and Y mantissas. Note FFT is done in-place so the y storage is reused for Y.

bfp_s32_t `x_bfp`[*IC_X_CHANNELS*]

BFP array pointing to the time domain x input signal.

bfp_complex_s32_t `X_bfp`[*IC_X_CHANNELS*]

BFP array pointing to the frequency domain X input signal.

int32_t `x`[*IC_X_CHANNELS*][*IC_FRAME_LENGTH* + *FFT_PADDING*]

Storage for x and X mantissas. Note FFT is done in-place so the x storage is reused for X.

bfp_s32_t `prev_y_bfp`[*IC_Y_CHANNELS*]

BFP array pointing to previous y samples which are used for framing.

int32_t `y_prev_samples`[*IC_Y_CHANNELS*][*IC_FRAME_LENGTH* - *IC_FRAME_ADVANCE*]

Storage for previous y mantissas.

bfp_s32_t `prev_x_bfp`[*IC_X_CHANNELS*]

BFP array pointing to previous x samples which are used for framing.

int32_t `x_prev_samples`[*IC_X_CHANNELS*][*IC_FRAME_LENGTH* - *IC_FRAME_ADVANCE*]

Storage for previous x mantissas.

bfp_complex_s32_t `Y_hat_bfp`[*IC_Y_CHANNELS*]

BFP array pointing to the estimated frequency domain Y signal.

complex_s32_t `Y_hat`[*IC_Y_CHANNELS*][*IC_FD_FRAME_LENGTH*]

Storage for Y_hat mantissas.

bfp_complex_s32_t `Error_bfp`[*IC_Y_CHANNELS*]

BFP array pointing to the frequency domain Error output.

bfp_s32_t `error_bfp`[*IC_Y_CHANNELS*]

BFP array pointing to the time domain Error output.

complex_s32_t `Error`[*IC_Y_CHANNELS*][*IC_FD_FRAME_LENGTH*]

Storage for Error and error mantissas. Note IFFT is done in-place so the Error storage is reused for error.

bfp_complex_s32_t `H_hat_bfp`[*IC_Y_CHANNELS*][*IC_X_CHANNELS* * *IC_FILTER_PHASES*]

BFP array pointing to the frequency domain estimate of transfer function.

complex_s32_t `H_hat`[*IC_Y_CHANNELS*][*IC_FILTER_PHASES* * *IC_X_CHANNELS*][*IC_FD_FRAME_LENGTH*]

Storage for H_hat mantissas.

bfp_complex_s32_t `X_fifo_bfp`[*IC_X_CHANNELS*][*IC_FILTER_PHASES*]

BFP array pointing to the frequency domain X input history used for calculating normalisation.

bfp_complex_s32_t `X_fifo_1d_bfp`[*IC_X_CHANNELS* * *IC_FILTER_PHASES*]

1D alias of the frequency domain X input history used for calculating normalisation.

complex_s32_t `X_fifo`[*IC_X_CHANNELS*][*IC_FILTER_PHASES*][*IC_FD_FRAME_LENGTH*]

Storage for X_fifo mantissas.

bfp_complex_s32_t `T_bfp`[*IC_X_CHANNELS*]

BFP array pointing to the frequency domain T used for adapting the filter coefficients (H). Note there is no associated storage because we re-use the x input array as a memory optimisation.

bfp_s32_t `inv_X_energy_bfp`[*IC_X_CHANNELS*]

BFP array pointing to the inverse X energies used for normalisation.

int32_t `inv_X_energy`[*IC_X_CHANNELS*][*IC_FD_FRAME_LENGTH*]

Storage for inv_X_energy mantissas.

bfp_s32_t `X_energy_bfp`[*IC_X_CHANNELS*]

BFP array pointing to the X energies.

int32_t `X_energy`[*IC_X_CHANNELS*][*IC_FD_FRAME_LENGTH*]

Storage for X_energy mantissas.

unsigned `X_energy_recalc_bin`

Index state used for calculating energy across all X bins.

bfp_s32_t `overlap_bfp`[*IC_Y_CHANNELS*]
: BFP array pointing to the overlap array used for windowing and overlap operations.

int32_t `overlap`[*IC_Y_CHANNELS*][IC_FRAME_OVERLAP]
: Storage for overlap mantissas.

int32_t `y_input_delay`[*IC_Y_CHANNELS*][*IC_Y_CHANNEL_DELAY_SAMPS*]
: FIFO for delaying y channel (w.r.t x) to enable adaptive filter to be effective.

uint32_t `y_delay_idx`[*IC_Y_CHANNELS*]
: Index state used for keeping track of y delay FIFO.

float_s32_t `mu`[*IC_Y_CHANNELS*][*IC_X_CHANNELS*]
: Mu value used for controlling adaption rate.

float_s32_t `leakage_alpha`
: Alpha used for leaking away H_hat, allowing filter to slowly forget adaption.

float_s32_t `max_X_energy`[*IC_X_CHANNELS*]
: Used to keep track of peak X energy.

bfp_s32_t `sigma_XX_bfp`[*IC_X_CHANNELS*]
: BFP array pointing to the EMA filtered X input energy.

int32_t `sigma_XX`[*IC_X_CHANNELS*][*IC_FD_FRAME_LENGTH*]
: Storage for sigma_XX mantissas.

float_s32_t `sum_X_energy`[*IC_X_CHANNELS*]
: X energy sum used for maintaining the X FIFO.

ic_config_params_t `config_params`
: Configuration parameters for the IC.

ic_adaption_controller_state_t `ic_adaption_controller_state`
: State and configuration parameters for the IC adaption controller.

vnr_pred_state_t `vnr_pred_state`
: Input and Output VNR Prediction related state

lib_ic API Definitions

group `ic_defines`

Defines

`IC_INIT_MU`

Initial MU value applied on startup. MU controls the adaption rate of the IC and is normally adjusted by the adaption rate controller during operation.

`IC_INIT_EMA_ALPHA`

Alpha used for calculating y_ema_energy, x_ema_energy and error_ema_energy.

`IC_INIT_LEAKAGE_ALPHA`

Alpha used for leaking away H_hat, allowing filter to slowly forget adaption. This value is adjusted by the adaption rate controller if instability is detected.

`IC_FILTER_PHASES`

The number of filter phases supported by the IC. Each filter phase represents 15ms of filter length. Hence a 10 phase filter will allow cancellation of noise sources with up to 150ms of echo tail length. There is a tradeoff between adaption speed and maximum cancellation of the filter; increasing the number of phases will increase the maximum cancellation at the cost of increased xCORE resource usage and slower adaption times.

`IC_Y_CHANNEL_DELAY_SAMPS`

This is the delay, in samples that one of the microphone signals is delayed in order for the filter to be effective. A larger number increases the delay through the filter but may improve cancellation. The group delay through the IC filter is 32 + this number of samples.

`IC_INIT_SIGMA_XX_SHIFT`

Down scaling factor for X energy calculation used for normalisation.

`IC_INIT_GAMMA_LOG2`

Up scaling factor for X energy calculation for used for LMS normalisation.

`IC_INIT_DELTA`

Delta value used in denominator to avoid large values when calculating inverse X energy.

`IC_INIT_FAST_RATIO_THRESHOLD`

Fast ratio threshold to detect instability.

`IC_INIT_ENERGY_ALPHA`

Alpha for EMA input/output energy calculation.

`IC_INIT_HIGH_INPUT_VNR_HOLD_LEAKAGE_ALPHA`

Leakage alpha used in case vnr detects high voice probability.

`IC_INIT_INSTABILITY_RECOVERY_LEAKAGE_ALPHA`

Leakage alpha used in the case where instability is detected. This allows the filter to stabilise without completely forgetting the adaption.

`IC_INIT_ADAPT_COUNTER_LIMIT`

Limits number of frames for which mu and leakage_alpha could be adapted.

`IC_INIT_INPUT_VNR_THRESHOLD`

VNR input threshold which decides whether to hold or adapt the filter.

`IC_INIT_INPUT_VNR_THRESHOLD_HIGH`

VNR high threshold to leak the filter is the speech level is high.

`IC_INIT_INPUT_VNR_THRESHOLD_LOW`

VNR low threshold to adapt faster when the speech level is low.

`IC_INIT_VNR_PRED_ALPHA`

Alpha for EMA VNR prediction calculation.

`IC_INIT_INPUT_VNR_PRED`

Initial value for the input VNR prediction.

`IC_INIT_OUTPUT_VNR_PRED`

Initial value for the output VNR prediction.

`IC_Y_CHANNELS`

Number of Y channels input. This is fixed at 1 for the IC. The Y channel is delayed and used to generate the estimated noise signal to subtract from X. In practical terms it does not matter which microphone is X and which is Y. NOT USER MODIFIABLE.

`IC_X_CHANNELS`

Number of X channels input. This is fixed at 1 for the IC. The X channel is the microphone from which the estimated noise signal is subtracted. In practical terms it does not matter which microphone is X and which is Y. NOT USER MODIFIABLE.

`IC_FRAME_LENGTH`

Time domain samples block length used internally in the IC's block LMS algorithm. NOT USER MODIFIABLE.

`IC_FRAME_ADVANCE`

IC new samples frame size This is the number of samples of new data that the IC works on every frame. 240 samples at 16kHz is 15msec. Every frame, the IC takes in 15msec of mic data and generates 15msec of interference cancelled output. NOT USER MODIFIABLE.

`IC_FD_FRAME_LENGTH`

Number of bins of spectrum data computed when doing a DFT of a IC_FRAME_LENGTH length time domain vector. The IC_FD_FRAME_LENGTH spectrum values represent the bins from DC to Nyquist. NOT USER MODIFIABLE.

`FFT_PADDING`

Extra 2 samples you need to allocate in time domain so that the full spectrum (DC to nyquist) can be stored after the in-place FFT. NOT USER MODIFIABLE.

lib_ic Header Files

ic_defines.h

page `page_ic_defines_h`

This header contains lib_ic public defines that are used to configure the interference canceller when *ic_init()* is called.

ic_state.h

page `page_ic_state_h`

This header contains definitions for data structures used in lib_ic. It also contains the configuration sub-structures which control the operation of the interference canceller during run-time.

ic_api.h

page `page_ic_api_h`

lib_ic public functions API.

On GitHub

`lib_ic` is present as part of `fwk_voice`. Get the latest version of `fwk_voice` from `https://github.com/xmos/fwk_voice`. The `lib_ic` module can be found in the *modules/lib_ic* directory in `fwk_voice`.

API

To use the functions in this library in an application, include *ic_api.h* in the application source file

1.1.6 Voice To Noise Ratio Estimator Library

`lib_vnr` is a library which estimates the ratio of speech signal in noise for an input audio stream. `lib_vnr` library functions uses `lib_xcore_math` to perform DSP using low-level optimised operations, and `lib_tflite_micro` and `lib_nn` to perform inference using an optimised TensorFlow Lite model.

Repository Structure

- `modules/lib_vnr` - The `lib_vnr` library directory within `https://github.com/xmos/fwk_voice/`. Within `lib_vnr`:
  - `api/` - Header files containing the public API for `lib_vnr`.
  - `doc/` - Library documentation source (for non-embedded documentation) and build directory.
  - `src/` - Library source code.

Requirements

`lib_vnr` is included as part of the `fwk_voice` github repository and all requirements for cloning and building `fwk_voice` apply. It depends on `lib_xcore_math`, `lib_tflite_micro` and `lib_nn`.

API Structure

The API is split into 2 parts; feature extraction and inference. The feature extraction API processes an input audio frame to extract features that are input to the inference stage. The inference API has functions for running inference using the VNR TensorFlow Lite model to predict the speech to noise ratio. Both feature extraction and inference APIs have initialisation functions that are called only once at device initialisation and processing functions that are called every frame. The performance requirement is relative low, around 5 MIPS for initialisation and 3 MIPS for processing, and as such is supplied as a single threaded implementation only.

Getting and Building

The VNR estimator module is obtained as part of the parent `fwk_voice` repo clone. It is present in `fwk_voice/modules/lib_vnr`

The feature extraction part of lib_vnr can be compiled as a static library. The application can link against `libfwk_voice_module_lib_vnr_features.a` and add `lib_vnr/api/features` and `lib_vnr/api/common` as include directories. VNR inference engine compilation however, requires the runtime HW target to be specified, information about which is not available at library compile time. To include VNR inference engine in an application, it needs to compile the VNR inference related files from source. lib_vnr module CMake file demonstrates the VNR inference engine compiled as an INTERFACE library and if compiling using CMake, the application can simply *link* against the fwk_voice::vnr::inference library. For an example of compiling an application with VNR using CMake, refer to VNR example CMake file.

VNR Inference Model

The VNR estimator module uses a neural network model to predict the SNR of speech in noise for incoming data. The model used is a pre trained TensorFlow Lite model that has been optimised for the XCORE architecture using the xmos-ai-tools xformer. The optimised model is compiled as part of the VNR Inference Engine. Changing the model at runtime is not supported. If changing to a different model, the application needs to generate the model related files, copy them to the appropriate directory within the VNR module and recompile. Part of this process is automated through a python script, as described below.

Integrating a TensorFlow Lite model into the VNR module This document describes the process for integrating a TensorFlow Lite model into the VNR module. Starting with an unoptimised model, follow the steps below to optimise it for XCORE by running it through the xmos-ai-tools xformer and integrate it into the VNR module.

1. Use the xformer to optimise the model for XCORE architecture.
2. Run the tflite_micro_compiler on the XCORE optimised model to generate the compiled .cpp and .h files that can be integrated in the VNR module.
3. Update the TensorFlow Lite 8-bit quantization spec for the new model in vnr_quant_spec_defines.h.

The xform_model.py script automates the above steps. It creates the files mentioned in steps 1-3 above and copies them to the VNR module directory. In addition to that, it also lists down the steps that the user is expected to do manually post running this script. These steps include things making sure any old model files if present are

deleted and the new files are added to git and all changes are committed. The script does provide a list of files that need removing and adding to git before committing to make this manual step easier.

Ensure you have installed Python 3 and the python requirements listed in requirements.txt in order to run the script. To use the script, run,

```
$ python xform_model.py <Unoptimised TensorFlow Lite model> --copy-files --module-path
↪<path to model related files in lib_vnr module>
```

The above command will generate the relevant files and copy them into the VNR module.

For example, to run it for the existing model that we have, run,

```
$ python xform_model.py fwk_voice/modules/lib_vnr/python/model/model_output/trained_model.
↪tflite --copy-files --module-path=fwk_voice/modules/lib_vnr/src/inference/model/
```

The process described above only generates an optimised model that would run on a single core.

Also worth mentioning is, since the feature extraction code is fixed and compiled as part of the VNR module, any new models replacing the existing one should have the same set of input features, input and output size and data types as the existing model.

VNR Overview

The VNR (Voice to Noise Ratio) estimator predicts the signal to noise ratio of a speech signal in noise, using a pre-trained neural network. The VNR neural network model outputs a value between 0 and 1, with 1 indicating the strongest speech, and 0, the weakest speech compared to noise in a frame of audio data.

The VNR module processes *VNR_FRAME_ADVANCE* new audio pcm samples every frame. The time domain input is transformed to frequency domain using a 512 point DFT. A MEL filterbank is then applied to compress the DFT output spectrum into fewer data points. The MEL filter outputs of *VNR_PATCH_WIDTH* most recent frames are normalised and fed as input features to the VNR prediction model which runs an inference over the features to output the VNR estimate value.

VNR estimations can be very helpful in voice processing pipelines. Applications for VNR include intelligent power management, control of adaptive filters for reducing noise sources and improved performance of AGC (Automatic Gain Control) blocks that provide a more natural listening experience.

The VNR API is split into 2 parts; feature extraction and inference. This is done to allow multiple sets of features to use the same inference engine. The VNR feature extraction is further split into 2 parts; a function to form the input frame that the feature extraction can run on, and a function to do the actual feature extraction. The function for forming the input frame starts from *VNR_FRAME_ADVANCE* new pcm samples and creates the DFT output that is used as input to the MEL filterbank. This has been separated from the rest of the feature extraction to support cases where the VNR might be using the DFT output computed in another module for extracting features.

The pre-trained, optimised for XCORE TensorFlow Lite model, that is used for VNR inference has been compiled as part of the VNR inference static library. There's no support for providing a new model to the inference engine at run time.

Before starting the feature extraction, the user must call `vnr_input_state_init()` and `vnr_feature_state_init()` to initialise the form input frame and feature extraction state. Before starting inference, the user must call `vnr_inference_init()` to initialise the inference engine.

There are no user configurable parameters within the VNR and so no arguments are required and no configuration structures need be tuned.

Once the VNR is initialised, the `vnr_form_input_frame()`, `vnr_extract_features()` and `vnr_inference()` functions should be called on a frame by frame basis.

API Reference

lib_vnr feature extraction API Functions

group `vnr_features_api`

Functions

void `vnr_input_state_init`(*vnr_input_state_t* *input_state)

Initialise previous frame samples buffer that is used when creating an input frame for processing through the VNR estimator.

This function should be called once at device startup.

Parameters

- `input_state` – **[inout]** pointer to the VNR input state structure

void `vnr_form_input_frame`(*vnr_input_state_t* *input_state, bfp_complex_s32_t *X, complex_s32_t X_data[*VNR_FD_FRAME_LENGTH*], const int32_t new_x_frame[*VNR_FRAME_ADVANCE*])

Create the input frame for processing through the VNR estimator.

This function takes in VNR_FRAME_ADVANCE new samples, combines them with previous frame's samples to form a VNR_PROC_FRAME_LENGTH samples input frame of time domain data, and outputs the DFT spectrum of the input frame. The DFT spectrum is output in the BFP structure and data memory provided by the user.

The frequency spectrum output from this function is processed through the VNR feature extraction stage.

If sharing the DFT spectrum calculated in some other module, *vnr_form_input_frame()* is not needed.

Example

```
#include "vnr_features_api.h"
complex_s32_t DWORD_ALIGNED input_frame[VNR_FD_FRAME_LENGTH];
bfp_complex_s32_t X;
vnr_form_input_frame(&vnr_input_state, &X, input_frame, new_data);
```

Parameters

- `input_state` – **[inout]** pointer to the VNR input state structure
- `X` – **[out]** pointer to a variable of type bfp_complex_s32_t that the user allocates. The user doesn't need to initialise this bfp variable. After this function, X is updated to point to the DFT output spectrum and can be passed as input to the feature extraction stage.
- `X_data` – **[out]** pointer to VNR_FD_FRAME_LENGTH values of type complex_s32_t that the user allocates. After this function, the DFT spectrum values are written to this array, and `X->data` points to X_data memory.
- `new_x_frame` – **[in]** Pointer to VNR_FRAME_ADVANCE new time domain samples

void `vnr_feature_state_init`(*vnr_feature_state_t* *feature_state)

Initialise the state structure for the VNR feature extraction stage.

This function is called once at device startup.

Parameters

- `feature_state` – **[inout]** pointer to the VNR feature extraction state structure

void `vnr_extract_features`(*vnr_feature_state_t* *vnr_feature_state, bfp_s32_t *feature_patch, int32_t feature_patch_data[*VNR_PATCH_WIDTH* * *VNR_MEL_FILTERS*], const bfp_complex_s32_t *X)

Extract features.

This function takes in DFT spectrum of the VNR input frame and does the feature extraction. The features are written to the feature_patch BFP structure and feature_patch_data memory provided by the user. The feature output from this function are passed as input to the VNR inference engine.

Parameters

- `vnr_feature_state` – **[inout]** Pointer to the VNR feature extraction state structure
- `feature_patch` – **[out]** Pointer to the bfp_s32_t structure allocated by the user. The user doesn't need to initialise this BFP structure before passing it to this function. After this function call feature_patch will be updated and will point to the extracted features. It can then be passed to the inference stage.
- `feature_patch_data` – **[out]** Pointer to the VNR_PATCH_WIDTH * VNR_MEL_FILTERS int32_t values allocated by the user. The extracted features will be written to the feature_patch_data array and the BFP structure's `feature_patch->data` will point to this array.

lib_vnr inference engine API Functions

group `vnr_inference_api`

Functions

int32_t `vnr_inference_init`()

Initialise the inference_engine object and load the VNR model into the inference engine.

This function calls lib_tflite_micro functions to initialise the inference engine and load the VNR model into it. It is called once at startup. The memory required for the inference engine object as well as the tensor arena size required for inference is statically allocated as global buffers in the VNR module. The VNR model is compiled as part of the VNR module.

void `vnr_inference`(float_s32_t *vnr_output, bfp_s32_t *features)

Run model prediction on a feature patch.

This function invokes the inference engine. It takes in a set of features corresponding to an input frame of data and outputs the VNR prediction value. The VNR output is a single value ranging between 0 and 1 returned in float_s32_t format, with 0 being the lowest SNR and 1 being the strongest possible SNR in speech compared to noise.

Parameters

- `vnr_output` – **[out]** VNR prediction value.

- `features` – **[in]** Input feature vector. Note that this is not passed as a const pointer and the feature memory is overwritten as part of the inference computation.

lib_vnr #defines common to feature extraction and inference

group `vnr_defines`

Defines

`VNR_MEL_FILTERS`

Number of filters in the MEL filterbank used in the VNR feature extraction.

`VNR_PATCH_WIDTH`

Number of frames that make up a full set of features for the inference to run on.

lib_vnr feature extraction #defines and data structure definitions

group `vnr_features_state`

Defines

`VNR_PROC_FRAME_LENGTH`

Time domain samples block length used internally in VNR DFT computation. NOT USER MODIFIABLE.

<>

`VNR_FRAME_ADVANCE`

VNR new samples frame size This is the number of samples of new data that the VNR processes every frame. 240 samples at 16kHz is 15msec. NOT USER MODIFIABLE.

`VNR_FD_FRAME_LENGTH`

Number of bins of spectrum data computed when doing a DFT of a VNR_PROC_FRAME_LENGTH length time domain vector. The VNR_FD_FRAME_LENGTH spectrum values represent the bins from DC to Nyquist. NOT USER MODIFIABLE.

struct `vnr_input_state_t`

#include <vnr_features_state.h> VNR form_input state structure.

Public Members

int32_t `prev_input_samples`[*VNR_PROC_FRAME_LENGTH* - *VNR_FRAME_ADVANCE*]

Previous frame time domain input samples which are combined with VNR_FRAME_ADVANCE new samples to form the VNR input frame.

struct `vnr_feature_config_t`

#include <vnr_features_state.h> VNR feature extraction config structure.

Public Members

int32_t `enable_highpass`

Enable highpass filtering of VNR MEL filter output. Disabled by default

struct `vnr_feature_state_t`

#include <vnr_features_state.h> State structure used in VNR feature extraction.

Public Members

int32_t `feature_buffers`[*VNR_PATCH_WIDTH*][*VNR_MEL_FILTERS*]

Feature buffer containing the most recent VNR_MEL_FILTERS frames' MEL frequency spectrum.

lib_vnr Header Files

vnr_features_api.h

page `page_vnr_features_api_h`

This header contains lib_vnr features extraction API functions.

vnr_inference_api.h

page `page_vnr_inference_api_h`

This header contains lib_vnr inference engine API functions.

vnr_defines.h

page `page_vnr_defines_h`

This header contains the lib_vnr public #defines that are common to both feature extraction and inference.

vnr_features_state.h

page `page_vnr_features_state_h`

This header contains lib_vnr feature extraction related public #defines and data structure definitions

On GitHub

`lib_vnr` is present as part of `fwk_voice`. Get the latest version of `fwk_voice` from `https://github.com/xmos/fwk_voice`. The `lib_vnr` module can be found in the *modules/lib_vnr* directory in `fwk_voice`.

2 Example Applications

Several examples are provided to demonstrate processing of audio using the audio processing algorithms individually as well as put together in a pipeline.

2.1 Building Examples

After configuring the CMake project (with the `BUILD_EXAMPLES` enabled), all the examples can be built by using the `make` command within the build directory. Individual examples can be built using `make EXAMPLE_NAME`, where `EXAMPLE_NAME` is the example to build.

2.2 Running Examples

In order to access binary files on the host from the XCore device over xscope, the examples make use of the xscope_fileio utility, which needs to be installed before running the example application. To install xscope_fileio, run the following command from the top level *fwk_voice* directory in a terminal where XMOS XTC tools are sourced. Make sure that cmake build step has been completed prior to this.

```
pip install -e build/fwk_voice_deps/xscope_fileio/
```

2.2.1 aec_1_thread

This example demonstrates how AEC functions are called on a single thread to process data through the AEC stage of a pipeline.

In it, a 32-bit, 4 channel wav file input.wav is read and processed through the AEC stage frame by frame. The AEC is configured for 2 mic input channels, 2 reference input channels, 10 phase main filter and a 5 phase shadow filter. The input file input.wav has 2 channels of mic input followed by 2 channels of reference input. Echo cancelled version of the mic input is generated as the AEC output and written to the output.wav file.

Building

Run the following commands in the fwk_voice/build folder to build the firmware for the XCORE-AI-EXPLORER board as a target:

```
cmake -S.. -DCMAKE_TOOLCHAIN_FILE=../xmos_cmake_toolchain/xs3a.cmake
make fwk_voice_example_bare_metal_aec_1_thread
```

```
# make sure you have the patch command available
cmake -G "NMake Makefiles" -S.. -DCMAKE_TOOLCHAIN_FILE=../xmos_cmake_toolchain/xs3a.cmake
nmake fwk_voice_example_bare_metal_aec_1_thread
```

Running

From the fwk_voice/build folder run:

```
pip install -e fwk_voice_deps/xscope_fileio
cd ../examples/bare-metal/aec_1_thread
python ../shared_src/python/run_xcoreai.py ../../../build/examples/bare-metal/aec_1_thread/
↪bin/fwk_voice_example_bare_metal_aec_1_thread.xe --input ../shared_src/test_streams/aec_
↪example_input.wav
```

```
pip install -e fwk_voice_deps/xscope_fileio
cd fwk_voice_deps/xscope_fileio/host
cmake -G "NMake Makefiles" .
nmake
cd ../../../../examples/bare-metal/aec_1_thread
python ../shared_src/python/run_xcoreai.py ../../../build/examples/bare-metal/aec_1_thread/
↪bin/fwk_voice_example_bare_metal_aec_1_thread.xe --input ../shared_src/test_streams/aec_
↪example_input.wav
```

Output

The output file output.wav is generated in the *fwk_voice/examples/bare-metal/aec_1_thread* directory. The input file input.wav is also present in the same directory. View output.wav and input.wav in Audacity to compare the echo cancelled output against the microphone input.

2.2.2 aec_2_threads

This example demonstrates how AEC functions are called on 2 threads to process data through the AEC stage of a pipeline.

In it, a 32-bit, 4 channel wav file input.wav is read and processed through the AEC stage frame by frame. The AEC is configured for 2 mic input channels, 2 reference input channels, 10 phase main filter and a 5 phase shadow filter.

The input file input.wav has 2 channels of mic input followed by 2 channels of reference input. Echo cancelled version of the mic input is generated as the AEC output and written to the output.wav file.

Building

Run the following commands in the fwk_voice/build folder to build the firmware for the XCORE-AI-EXPLORER board as a target:

```
cmake -S.. -DCMAKE_TOOLCHAIN_FILE=../xmos_cmake_toolchain/xs3a.cmake
make fwk_voice_example_bare_metal_aec_2_thread
```

```
# make sure you have the patch command available
cmake -G "NMake Makefiles" -S.. -DCMAKE_TOOLCHAIN_FILE=../xmos_cmake_toolchain/xs3a.cmake
nmake fwk_voice_example_bare_metal_aec_2_thread
```

Running

From the fwk_voice/build folder run:

```
pip install -e fwk_voice_deps/xscope_fileio
cd ../examples/bare-metal/aec_2_thread
python ../shared_src/python/run_xcoreai.py ../../../build/examples/bare-metal/aec_2_thread/
↪bin/fwk_voice_example_bare_metal_aec_2_thread.xe --input ../shared_src/test_streams/aec_
↪example_input.wav
```

```
pip install -e fwk_voice_deps/xscope_fileio
cd fwk_voice_deps/xscope_fileio/host
cmake -G "NMake Makefiles" .
nmake
cd ../../../../examples/bare-metal/aec_2_thread
python ../shared_src/python/run_xcoreai.py ../../../build/examples/bare-metal/aec_2_thread/
↪bin/fwk_voice_example_bare_metal_aec_2_thread.xe --input ../shared_src/test_streams/aec_
↪example_input.wav
```

Output

The output file output.wav is generated in the *fwk_voice/examples/bare-metal/aec_2_threads* directory. The input file input.wav is also present in the same directory. View output.wav and input.wav in Audacity to compare the echo cancelled output against the microphone input.

2.2.3 vnr

This example demonstrates how the VNR functions are called on a single thread to generate the Voice to Noise Ratio (VNR) estimates for an input audio stream.

In this example, a 32-bit, 1 channel wav file test_stream_1.wav is read and processed through the VNR frame by frame. The neural network inference model used in the VNR is pre-trained to estimate the voice to noise ratio. It outputs a number between 0 and 1, 1 being the strongest voice with respect to noise and 0 being the lowest voice compared to noise ratio.

Building

After configuring the CMake project, the following commands can be used from the fwk_voice/examples/bare-metal/vnr directory to build and run this example application using the XCORE-AI-EXPLORER board as a target:

```
cd ../../../build
make fwk_voice_example_bare_metal_vnr_fileio
cd ../examples/bare-metal/vnr
python host_app.py test_stream_1.wav vnr_out.bin --run-with-xscope-fileio --show-plot
```

Alternatively, to not have the VNR output plot displayed on the screen, run,

```
python host_app.py test_stream_1.wav vnr_out.bin --run-with-xscope-fileio
```

Output

The output from the VNR is written into the *vnr_out.bin* file. For every frame, VNR outputs its estimate in the form of a floating point value between 0 and 1. The floating point value is written as a 32bit mantissa, followed by a 32bit exponent in the vnr_out.bin file. Additionally, these estimates are plotted, with the plot displayed on screen when run with the `--show-plot` argument. Irrespective of whether on not `--show-plot` is used as an option, the plotted figure is saved in the *vnr_example_plot_test_stream_1.png* file.

2.2.4 ic

This example demonstrates how the IC functions are called to process data through the IC stage of a voice pipeline.

A 32-bit, 2 channel wav file input.wav is read and processed through the IC stage frame by frame. The input file consists of 2 channels of mic input consisting of a *Alexa* utterances with a point noise source consisting of pop music. The signal and noise sources in input.wav come from different spatial locations.

The interference cancelled version of the mic input is generated as the IC output and written to the output.wav file. In this example, a VNR is not used and so the VNR signal is set to 0 to indicate that voice is not present, meaning adaption will occur. In a practical system, the VNR voice to noise ratio would increase during the utterances to ensure the IC does not adapt to the voice and cause it to be attenuated. The test file has only a few short voice utterances and so the example works and demonstrates the IC operation.

Building

Run the following commands in the fwk_voice/build folder to build the firmware for the XCORE-AI-EXPLORER board as a target:

```
cmake -S.. -DCMAKE_TOOLCHAIN_FILE=../xmos_cmake_toolchain/xs3a.cmake
make fwk_voice_example_bare_metal_ic

# make sure you have the patch command available
cmake -G "NMake Makefiles" -S.. -DCMAKE_TOOLCHAIN_FILE=../xmos_cmake_toolchain/xs3a.cmake
nmake fwk_voice_example_bare_metal_ic
```

Running

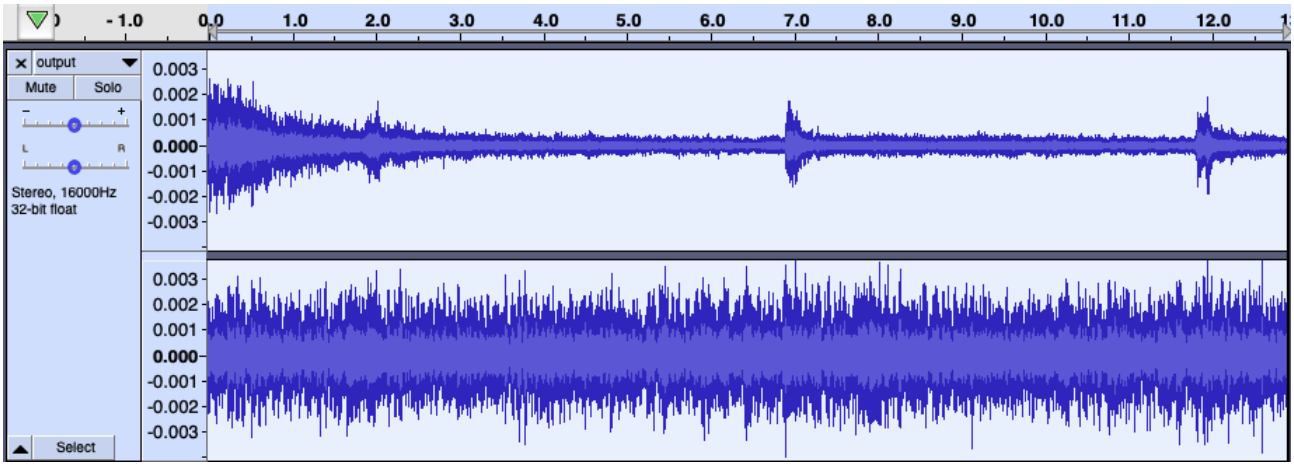
From the fwk_voice/build folder run:

```
pip install -e fwk_voice_deps/xscope_fileio
cd ../examples/bare-metal/ic
python ../shared_src/python/run_xcoreai.py ../../../build/examples/bare-metal/ic/bin/fwk_
↪voice_example_bare_metal_ic.xe

pip install -e fwk_voice_deps/xscope_fileio
cd fwk_voice_deps/xscope_fileio/host
cmake -G "NMake Makefiles" .
nmake
cd ../../../../examples/bare-metal/ic
python ../shared_src/python/run_xcoreai.py ../../../build/examples/bare-metal/ic/bin/fwk_
↪voice_example_bare_metal_ic.xe
```

Output

The output file output.wav is generated in the *fwk_voice/examples/bare-metal/ic* directory. When viewing output.wav in a visual audio tool, such as Audacity, you can see a stark difference between the channels emitted. Channel 0 is the IC output and is suitable for increasing the SNR in automatic speech recognition (ASR) applications. Channel 1 is the *simple beamformed* (average of mic 0 and mic 1 inputs) which may be preferable in comms (human to human) applications. The logic for channel 1 is contained in the `ic_test_task.c` file and is not part of the IC library.s



You can see the drastic reduction of the amplitude of the music noise source in channel 0 after just just a few seconds whilst the voice signal source is preserved. By zooming in to the start of the waveform, you can also see the 212 sample (180 y_delay + 32 framing delay) latency through the IC, when compared with the averaged output of channel 1.

2.2.5 agc

This example demonstrates how AGC functions are called on a single thread to process data through the AGC stage of a pipeline. A single AGC instance is run using the profile that is tuned for communication with a human listener.

Since this example application only demonstrates the AGC module, without a VNR or an AEC, adaption based on voice activity and the loss control feature are both disabled.

The input is a single channel, 32-bit wav file, which is read and processed through the AGC frame-by-frame.

Building

Run the following commands in the fwk_voice/build folder to build the firmware for the XCORE-AI-EXPLORER board as a target:

```
cmake -S.. -DCMAKE_TOOLCHAIN_FILE=../xmos_cmake_toolchain/xs3a.cmake
make fwk_voice_example_bare_metal_agc

# make sure you have the patch command available
cmake -G "NMake Makefiles" -S.. -DCMAKE_TOOLCHAIN_FILE=../xmos_cmake_toolchain/xs3a.cmake
nmake fwk_voice_example_bare_metal_agc
```

Running

From the fwk_voice/build folder run:

```
pip install -e fwk_voice_deps/xscope_fileio
cd ../examples/bare-metal/agc
python ../shared_src/python/run_xcoreai.py ../../../build/examples/bare-metal/agc/bin/fwk_
↪voice_example_bare_metal_agc.xe --input ../shared_src/test_streams/agc_example_input.wav

pip install -e fwk_voice_deps/xscope_fileio
cd fwk_voice_deps/xscope_fileio/host
cmake -G "NMake Makefiles" .
nmake
cd ../../../../examples/bare-metal/agc
python ../shared_src/python/run_xcoreai.py ../../../build/examples/bare-metal/agc/bin/fwk_
↪voice_example_bare_metal_agc.xe --input ../shared_src/test_streams/agc_example_input.wav
```

Output

The output file output.wav is generated in the *fwk_voice/examples/bare-metal/agc* directory. The provided input *agc_example_input.wav* is low-volume white-noise and the effect of the AGC can be heard in the output by listening to the two wav files.

2.2.6 pipeline_single_threaded

This example demonstrates how the audio processing stages are put together in a pipeline

In it, a 32-bit, 4 channel wav file input.wav is read and processed through the pipeline stages frame by frame. The example currently demonstrates a pipeline having AEC, IC, NS and AGC stages. It also demonstrates the use of ADEC module to do a one time estimation and correction for possible reference and loudspeaker delay offsets at start up in order to maximise AEC performance. ADEC processing happens on the same thread as the AEC. The VNR is introduced to give the IC and the AGC information about the speech presence in a frame.

The AEC is configured for 2 mic input channels, 2 reference input channels, 10 phase main filter and a 5 phase shadow filter. The AEC gets reconfigured as a 1 mic input channel, 1 reference input channel, 30 main filter phases and no shadow filter, when ADEC goes in delay estimation mode. This allows it to measure the room delay. During this process, the AEC output is ignored and the mic input is directly sent to output. Once the new delay has been measured and the delay correction is applied, the AEC gets configured back to its original configuration and starts adapting and cancellation. This example supports a maximum of 150ms of delay correction, in either direction, between the reference and microphone input. The AEC stage generates the echo cancelled version of the mic input that is then sent for processing through the IC.

The IC only processes a two channel input. It will use the second channel as the reference to the first to output one channel of interference cancelled output. In this manner, it tries to cancel the room noise. However, to avoid cancelling the wanted signal, it only adapts in the absence of voice. Hence the VNR is called to calculate the voice to noise ratio estimation. The output of the VNR will allow IC to modulate the rate at which it adapts it's coefficients. The output of the IC is copied to the second channel as well.

The NS is a single channel API, so two instances of NS should be initialised for 2 channel processing. The NS is configured the same way for both the channels. It will try to predict the background noise and cancel it from the frame before passing it to AGC.

The AGC is configured for ASR engine suitable gain control on both the channels. The output of the AGC stage is the pipeline output which is written into a 2 channel output wav file. The AGC also takes the output of the VNR to adapt it's coefficients.

The pipeline is run on a single thread. To run the pipeline on a single xcore-AI thread a minimum thread speed of 140MHz is recommended, for a typical pipeline configuration.

The input file input.wav has a total of 4 channels and should have bit depth of 32b. Due to the microphone inputs being very low amplitude, 16b data would result in severe quantisation of the data. The first 2 channels in the 4 channel file are the mic inputs followed by 2 channels of reference input. Output is written to the output.wav file consisting of 2 channels.

Building

Run the following commands in the fwk_voice/build folder to build the firmware for the XCORE-AI-EXPLORER board as a target:

```
cmake -S.. -DCMAKE_TOOLCHAIN_FILE=../xmos_cmake_toolchain/xs3a.cmake
make fwk_voice_example_bare_metal_pipeline_single_thread
```

```
# make sure you have the patch command available
cmake -G "NMake Makefiles" -S.. -DCMAKE_TOOLCHAIN_FILE=../xmos_cmake_toolchain/xs3a.cmake
nmake fwk_voice_example_bare_metal_pipeline_single_thread
```

Running

From the fwk_voice/build folder run:

```
pip install -e fwk_voice_deps/xscope_fileio
cd ../examples/bare-metal/pipeline_single_threaded
python ../shared_src/python/run_xcoreai.py ../../../build/examples/bare-metal/pipeline_
↪single_threaded/bin/fwk_voice_example_bare_metal_pipeline_single_thread.xe --input ../
↪shared_src/test_streams/pipeline_example_input.wav
```

```
pip install -e fwk_voice_deps/xscope_fileio
cd fwk_voice_deps/xscope_fileio/host
cmake -G "NMake Makefiles" .
nmake
cd ../../../../examples/bare-metal/pipeline_single_threaded
python ../shared_src/python/run_xcoreai.py ../../../build/examples/bare-metal/pipeline_
↪single_threaded/bin/fwk_voice_example_bare_metal_pipeline_single_thread.xe --input ../
↪shared_src/test_streams/pipeline_example_input.wav
```

2.2.7 pipeline_multi_threaded

This example demonstrates how the audio processing stages are put together in a pipeline where stages are run in parallel on separate hardware threads.

In it, a 32-bit, 4 channel wav file input.wav is read and processed through the pipeline stages frame by frame. The example currently demonstrates a pipeline having AEC, IC, NS and AGC stages. It also demonstrates the use of ADEC module to do a one time estimation and correction for possible reference and loudspeaker delay offsets at

start up in order to maximise AEC performance. ADEC processing happens on the same thread as the AEC. The VNR is introduced to give the IC and the AGC information about the speech presence in a frame.

The AEC is configured for 2 mic input channels, 2 reference input channels, 10 phase main filter and a 5 phase shadow filter. This example calls AEC functions using 2 threads to process a frame through the AEC stage. The AEC gets reconfigured as a 1 mic input channel, 1 reference input channel, 30 main filter phases and no shadow filter, when ADEC goes in delay estimation mode. This allows it to measure the room delay. During this process, the AEC output is ignored and the mic input is directly sent to output. Once the new delay has been measured and the delay correction is applied, the AEC gets configured back to its original configuration and starts adapting and cancellation. This example supports a maximum of 150ms of delay correction, in either direction, between the reference and microphone input. The AEC stage generates the echo cancelled version of the mic input that is then sent for processing through the IC.

The IC only processes a two channel input. It will use the second channel as the reference to the first to output one channel of interference cancelled output. In this manner, it tries to cancel the room noise. However, to avoid cancelling the wanted signal, it only adapts in the absence of voice. Hence the VNR is called to calculate the voice to noise ratio estimation. The output of the VNR will allow IC to modulate the rate at which it adapts it's coefficients. The output of the IC is copied to the second channel as well.

The NS is a single channel API, so two instances of NS should be initialised for 2 channel processing. The NS is configured the same way for both channels. It will try to predict the background noise and cancel it from the frame before passing it to AGC.

The AGC is configured for ASR engine suitable gain control on both channels. The output of AGC stage is the pipeline output which is written into a 2 channel output wav file. The AGC also takes the output of the VNR to control when to adapt. This avoids noise being amplified during the absence of voice.

In total, the audio processing stages consume 5 hardware threads; 2 for AEC stage, 1 for IC and VNR, 1 for NS stage and 1 for AGC stage. Note that it is possible to run the full pipeline in as little as two 75MHz threads if required using one thread for stage 1 and a second thread for all remaining blocks.

The input file input.wav has a total of 4 channels and should have bit depth of 32b. Due to the microphone inputs being very low amplitude, 16b data would result in severe quantisation of the data. The first 2 channels in the 4 channel file are the mic inputs followed by 2 channels of reference input. Output is written to the output.wav file consisting of 2 channels.

Building

Run the following commands in the fwk_voice/build folder to build the firmware for the XCORE-AI-EXPLORER board as a target:

```
cmake -S.. -DCMAKE_TOOLCHAIN_FILE=../xmos_cmake_toolchain/xs3a.cmake
make fwk_voice_example_bare_metal_pipeline_multi_thread
```

```
# make sure you have the patch command available
cmake -G "NMake Makefiles" -S.. -DCMAKE_TOOLCHAIN_FILE=../xmos_cmake_toolchain/xs3a.cmake
nmake fwk_voice_example_bare_metal_pipeline_multi_thread
```

Running

From the fwk_voice/build folder run:

```
pip install -e fwk_voice_deps/xscope_fileio
cd ../examples/bare-metal/pipeline_multi_threaded
python ../shared_src/python/run_xcoreai.py ../../../build/examples/bare-metal/pipeline_
↪multi_threaded/bin/fwk_voice_example_bare_metal_pipeline_multi_thread.xe --input ../
↪shared_src/test_streams/pipeline_example_input.wav

pip install -e fwk_voice_deps/xscope_fileio
cd fwk_voice_deps/xscope_fileio/host
cmake -G "NMake Makefiles" .
nmake
cd ../../../../examples/bare-metal/pipeline_multi_threaded
python ../shared_src/python/run_xcoreai.py ../../../build/examples/bare-metal/pipeline_
↪multi_threaded/bin/fwk_voice_example_bare_metal_pipeline_multi_thread.xe --input ../
↪shared_src/test_streams/pipeline_example_input.wav
```

Output

The output file output.wav is generated in the *fwk_voice/examples/bare-metal/pipeline_multi_threaded* directory. The input file input.wav is also present in the same directory. View output.wav and input.wav in Audacity to compare the pipeline output against the microphone input.

2.2.8 pipeline_alt_arch

This example demonstrates how the audio processing stages are put together in an alternate implementation of the pipeline, which is different from sequentially calling the stages one after the other. In this pipeline form, the AEC and the IC frame processing are selectively enabled and disabled based on the presence of reference input signal. Acoustic Echo Cancellation is performed only if activity is detected on the reference input channels and disabled otherwise. Interference Cancellation is performed only when AEC is disabled so in the absence of reference channel activity and disabled otherwise.

In this example, a 32-bit, 4 channel wav file input.wav is read and processed through the pipeline modules frame by frame. The example currently demonstrates a pipeline having AEC, IC, NS and AGC stages. It also demonstrates the use of ADEC module to do a one time estimation and correction for possible reference and loudspeaker delay offsets at start up in order to maximise AEC performance. ADEC processing happens on the same thread as the AEC. The VNR is introduced to give the IC and the AGC information about the speech presence in a frame.

The AEC is configured for 1 mic input channel, 2 reference input channels, 15 phase main filter and a 5 phase shadow filter giving an extended tail length of 225ms which is suitable for highly reverberant environments. The AEC gets reconfigured as a 1 mic input channel, 1 reference input channel, 30 main filter phases and no shadow filter, when ADEC goes in delay estimation mode. This allows it to measure the room delay. During this process, the AEC output is ignored and the mic input is directly sent to output. Once the new delay has been measured and the delay correction is applied, the AEC gets configured back to its original configuration and starts adapting and cancellation. This example supports a maximum of 150ms of delay correction, in either direction, between the reference and microphone input.

In the absence of activity on the reference channels, when the AEC is disabled, the mic input is copied directly to the output of the AEC.

When enabled, the IC processes the two channel input. It will use the second channel as the reference to the first to output one channel of interference cancelled output. In this manner, it tries to cancel the room noise. However, to avoid cancelling the wanted signal, it only adapts in the absence of voice. Hence the VNR is called to calculate the voice to noise ratio estimation in a frame. The output of the VNR will allow IC to modulate the rate at which

it adapts it's coefficients. The output of the IC is copied to the second channel as well. When disabled in the presence of reference channel activity, the IC stage configured in bypass mode.

The NS is a single channel API, so two instances of NS should be initialised for 2 channel processing. The NS is configured the same way for both channels. It will try to predict the background noise and cancel it from the frame before passing it to AGC.

The AGC is configured for ASR engine suitable gain control on both channels. The output of AGC stage is the pipeline output which is written into a 2 channel output wav file. The AGC also takes the output of the VNR to control when to adapt. This avoids noise being amplified during the absence of voice.

The example build outputs 2 executables, a single thread and a multi-thread implementation of the pipeline. The single thread version does the entire pipeline processing on a single thread. In the multi-thread version, the audio processing consumes 5 hardware threads; 2 for the AEC stage, 1 for the IC and VAD, 1 for the NS stage and 1 for the AGC stage. Note that it is possible to run the full pipeline in as little as two 75MHz threads if required using one thread for stage 1 and a second thread for all remaining blocks. Alternatively, a single 150MHz thread may support all stages of the pipeline within a single thread.

The input file input.wav has a total of 4 channels and should have bit depth of 32b. Due to the microphone inputs being very low amplitude, 16b data would result in severe quantisation of the data. The first 2 channels in the 4 channel file are the mic inputs followed by 2 channels of reference input. Output is written to the output.wav file consisting of 2 channels.

Building

Run the following commands in the fwk_voice/build folder to build the multi-threaded firmware for the XCORE-AI-EXPLORER board as a target:

```
cmake -S.. -DCMAKE_TOOLCHAIN_FILE=../xmos_cmake_toolchain/xs3a.cmake
make fwk_voice_example_bare_metal_pipeline_alt_arch_mt


# make sure you have the patch command available
cmake -G "NMake Makefiles" -S.. -DCMAKE_TOOLCHAIN_FILE=../xmos_cmake_toolchain/xs3a.cmake
nmake fwk_voice_example_bare_metal_pipeline_alt_arch_mt
```

To build the single-threaded firmware use fwk_voice_example_bare_metal_pipeline_alt_arch_st cmake target when doing make(nmake).

Running

To run the multi-threaded application run these commands from the fwk_voice/build folder:

```
pip install -e fwk_voice_deps/xscope_fileio
cd ../examples/bare-metal/pipeline_alt_arch
 python ../shared_src/python/run_xcoreai.py ../../../build/examples/bare-metal/pipeline_alt_
↪arch/bin/fwk_voice_example_bare_metal_pipeline_alt_arch_mt.xe --input ../shared_src/test_
↪streams/pipeline_example_input.wav


pip install -e fwk_voice_deps/xscope_fileio
cd fwk_voice_deps/xscope_fileio/host
cmake -G "NMake Makefiles" .
nmake
cd ../../../../examples/bare-metal/pipeline_alt_arch
```

(continues on next page)

(continued from previous page)

```
python ../shared_src/python/run_xcoreai.py ../../../build/examples/bare-metal/pipeline_alt_
↪arch/bin/fwk_voice_example_bare_metal_pipeline_alt_arch_mt.xe --input ../shared_src/test_
↪streams/pipeline_example_input.wav
```

To run the single-threaded application use fwk_voice_example_bare_metal_pipeline_alt_arch_st.xe as an executable for the python script.

Output

The output file output.wav is generated in the *fwk_voice/examples/bare-metal/pipeline_alt_arch* directory. The input file input.wav is also present in the same directory. View output.wav and input.wav in Audacity to compare the pipeline output against the microphone input.

2 Index

A

C

D

F

I

N

S

V