



lib_voice: Voice processing library

Publication Date: 2026/3/31

Document Number: XM-010445-UG v1.0.1

IN THIS DOCUMENT

1	Audio processing	4
1.1	Acoustic Echo Celler	4
1.2	Interference Celler	9
1.3	Voice to Noise Ratio Estimator	11
1.4	Noise Suppressor	13
1.5	Automatic Gain Control	14
1.6	Automatic Delay Estimation and Correction	19
1.7	Pipeline Stage 1	25
2	Examples	30
2.1	AEC example	30
2.2	VNR example	30
2.3	Pipeline example	31
2.4	Building the example	32
2.5	Running the example	33
3	Resource usage	34
3.1	Memory	34
3.2	CPU	34
4	API reference	36
4.1	AEC	36
4.2	IC	54
4.3	VNR	63
4.4	NS	67
4.5	AGC	70
4.6	ADEC	76
4.7	Stage1	81



Introduction

lib_voice is a collection of DSP components used to build a front-end voice processing pipeline.

At its core, the library provides high-performance audio processing algorithms that are combined into a configurable pipeline. The pipeline takes input from a pair of microphones and applies a sequence of signal processing stages to extract a clean voice signal from complex acoustic environments. An optional reference signal from a host system can be provided to enable Acoustic Echo Cancellation (AEC), removing echo from the microphone signal.

The pipeline produces two output streams: one optimized for Automatic Speech Recognition (ASR) systems and another suitable for voice communications.

lib_voice includes a flexible audio routing infrastructure and supports a range of digital inputs and outputs, allowing it to be integrated into a wide variety of system configurations. The pipeline can be configured at startup and adjusted during operation via a set configuration parameters. All source code is provided, enabling full customization and the integration of additional audio processing algorithms.

Usage

lib_voice is intended to be used with the [XCommon CMake](#), the XMOS application build and dependency management system.

To use this library in an application include **lib_voice** in the application's **APP_DEPENDENT_MODULES** list in *CMakeLists.txt*, for example:

```
set(APP_DEPENDENT_MODULES "lib_voice")
```

Note

Dependent modules should be pinned to release versions where possible, otherwise the latest commit on the *develop* branch will be used. For further details on managing modules, pinning to a release version and other options, please see the page [xcommon-cmake Dependency Management](#).

All **lib_voice** functions can be accessed via the **voice.h** header file, for example:

```
#include "voice.h"
```

Documentation structure

This document will cover the following topics:

1. [Audio processing](#) : A walkthrough of the DSP modules supported by **lib_voice**.
2. [Examples](#) : Simple demonstrations of how to put APIs together and get them running.
3. [Resource usage](#) : Memory and CPU requirements for **lib_voice** DSP components
4. [API reference](#) : References to all DSP components' APIs.



1 Audio processing

This section describes the functionality of each **lib_voice** modules, diving into the details of the underlying DSP.

All the modules have been designed to be used together in a voice processing pipeline, but they can also be used independently or in custom combinations. The modules have been designed to work with a 16 kHz sample rates, but may work at other samples rates if the time domain parameters are adjusted correctly.

Many of the modules in **lib_voice** use block floating point (BFP) arithmetic to achieve high performance on fixed-point hardware. In BFP, a block of data shares a common exponent, allowing for efficient scaling and dynamic range management. For more information on BFP, see the [XCore BFP documentation](#).

1.1 Acoustic Echo Canceller

An acoustic echo canceller (AEC) removes signal that is played through a device's loudspeaker, into the room, and picked up again by its microphones. The difference between the loudspeaker signal, referred to as the *reference signal*, and the microphone signal is used by the AEC to model the acoustic paths between loudspeakers and microphones. Using this model, the AEC predicts the resulting echo and subtracts it from the captured microphone signal in real time. By eliminating this feedback, the AEC ensures clear communication and prevents far-end listeners from hearing their own voice echoed back.

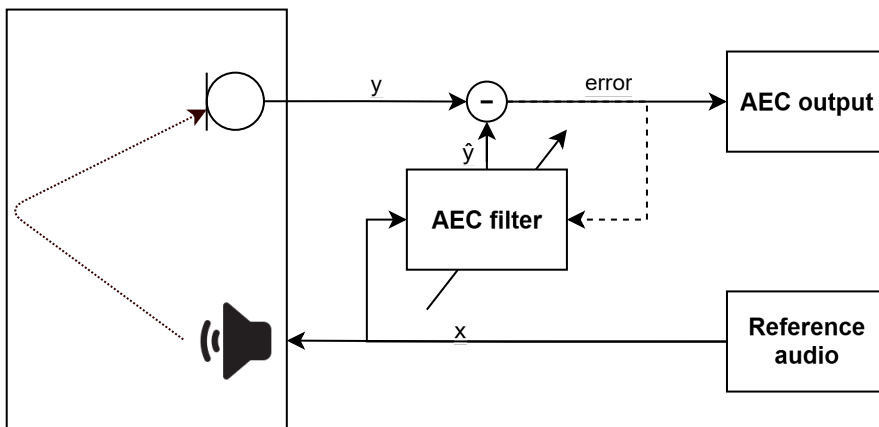


Fig. 1: A basic AEC filter.

Overview

The AEC component in **lib_voice** processes one or more channels of microphone input together with one or more channels of reference input. Microphone input is the audio captured by the device microphones. Reference input is the audio signal sent to the device loudspeakers. Using the reference input, the AEC estimates how sound propagates through the acoustic environment and removes the resulting echo from the microphone signal. The resulting output is the *error signal*, which represents the echo-cancelled microphone signal.

Echo cancellation is performed independently for each microphone–loudspeaker pair. For a system with M microphone channels and N reference channels, the AEC maintains $M \times N$ adaptive filters, each modeling the acoustic path from a particular loudspeaker to a particular microphone. 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.

Signal Representation

Processing is performed on a frame-by-frame basis. Each frame consists of 15 ms of audio, corresponding to 240 samples at a 16 kHz sampling rate, per channel. For example, a configuration with two microphone channels and two reference channels processes 2×240 samples of microphone data and 2×240 samples of reference data per frame.

Adaptive Filters

The AEC uses frequency-domain adaptive filters to estimate and remove echo. Each filter has a configurable number of phases, where the number of phases determines the effective tail length of the filter. Longer filters can model more reverberant acoustic environments and generally provide improved echo suppression, at the cost of increased computation and slower adaptation.

Two types of adaptive filters are used:

- Main filter
- Shadow filter

Each microphone-reference pair has one main filter and one shadow filter.

The main filter is used to generate the echo-cancelled output of the AEC. It typically has a longer tail length, allowing it to converge to a more accurate estimate of the room impulse response and achieve deeper echo cancellation. In larger rooms with more reverberation, a longer tail length may be necessary to achieve good echo cancellation, as the echo path is longer. This is shown in [Fig. 2](#).

The shadow filter has fewer phases and is designed to adapt more quickly. It is used to detect changes in the acoustic environment, such as people moving or doors opening and closing. When the shadow filter outperforms the main filter, its coefficients can be promoted to the main filter, allowing the AEC to respond rapidly to environmental changes.

Processing Flow

For each frame, the AEC performs the following high-level steps:

1. Transform microphone and reference signals into the frequency domain.
2. Estimate the echo contribution using the adaptive filters.
3. Subtract the estimated echo from the microphone signal to produce the error signal.
4. Update filter coefficients based on the error signal.
5. Transform the error signal back to the time domain to produce the echo-cancelled output.

Usage

Before starting processing, or whenever the configuration changes, the AEC must be initialised by calling `aec_init()`. This sets up internal state for a given runtime configuration (channels and number of phases).



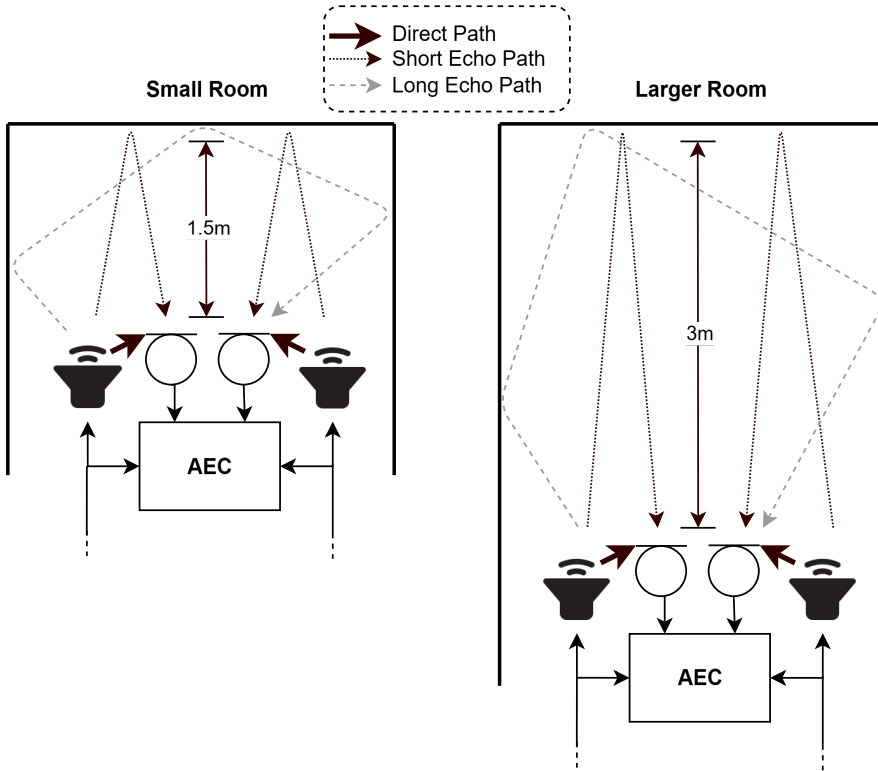


Fig. 2: Echo paths from the speakers to the microphones.

Once initialised, echo cancellation is performed by calling `aec_process_frame()` for each input frame.

Examples of initialising and running the AEC using one or two hardware threads are provided in the [AEC example](#). Alternatively, refer to [Pipeline example](#) to see how to use AEC as part of the [Pipeline Stage 1](#).

For configuration details (compile-time limits, memory pools, schedules), see the [Configuration](#) and [Schedules \(work distribution\)](#) sections below.

Configuration

The AEC is designed to support a range of runtime configurations while avoiding dynamic memory allocation at runtime. There are two layers of configuration:

- ▶ Compile-time capacity (maximums)
 - ▶ [AEC_MAX_Y_CHANNELS](#)
 - ▶ [AEC_MAX_X_CHANNELS](#)
 - ▶ [AEC_MAIN_FILTER_PHASES](#)
 - ▶ [AEC_SHADOW_FILTER_PHASES](#)



These macros determine the size of the memory pools and the upper bounds for run-time settings.

► Runtime configuration

`aec_init()` parameters: `num_y_channels`, `num_x_channels`, `num_main_filter_phases` and `num_shadow_filter_phases`. The run-time configuration must be a *valid subset* of the compile-time limits. Changing any of these parameters requires calling `aec_init()` again to reinitialise the AEC state.

Memory pools AEC binds internal BFP structures to preallocated memory pools:

► `aec_memory_pool_t` (main filter + shared state)

► `aec_shadow_filt_memory_pool_t` (shadow filter)

The pools must be allocated with capacity matching the compile-time macros above. At initialisation, `aec_init()` maps the pools to internal BFP structures sized to the runtime configuration. The pools must remain valid for the lifetime of the AEC instance.

Preconditions To be a valid runtime configuration, `aec_init()` parameters - `num_y_channels`, `num_x_channels`, `num_main_filter_phases` and `num_shadow_filter_phases` must satisfy:

- `num_y_channels` ≤ `AEC_MAX_Y_CHANNELS`
- `num_x_channels` ≤ `AEC_MAX_X_CHANNELS`
- `num_main_filter_phases` ≤ `AEC_MAIN_FILTER_PHASES`
- `num_shadow_filter_phases` ≤ `AEC_SHADOW_FILTER_PHASES`

► Phase-pool demand does not exceed capacity:

- Main filter:
$$\begin{array}{rcl} (\text{num_y_channels} & \times & \text{num_x_channels} \\ \text{num_main_filter_phases}) & + & (\text{num_x_channels} \\ \text{num_main_filter_phases}) & \leq & (\text{AEC_MAX_Y_CHANNELS} \\ \text{AEC_MAX_X_CHANNELS} & \times & \text{AEC_MAIN_FILTER_PHASES}) \\ (\text{AEC_MAX_X_CHANNELS} \times \text{AEC_MAIN_FILTER_PHASES}) & + & \end{array}$$
- Shadow filter:
$$\begin{array}{rcl} (\text{num_y_channels} & \times & \text{num_x_channels} \\ \text{num_shadow_filter_phases}) & \leq & (\text{AEC_MAX_Y_CHANNELS} \\ \text{AEC_MAX_X_CHANNELS} \times \text{AEC_SHADOW_FILTER_PHASES}) & \times & \end{array}$$

Schedules (work distribution)

Distributing `aec_process_frame()` work across hardware threads is controlled by an AEC task distribution schedule (`aec_task_distribution_t`). The schedule is passed to `aec_init()` via the `tdist` argument.

Default schedules The library has pre-compiled schedules for running AEC processing of a pre-defined configuration (`AEC_MAX_Y_CHANNELS` = 2 and `AEC_MAX_X_CHANNELS` = 2) on either 1 or 2 hardware threads: `aec_tdist_chans2_threads1` and `aec_tdist_chans2_threads2`. Either of these can be passed to `aec_init()`.

Custom schedules Alternatively, a custom schedule can be generated by setting `AEC_SCHEDULE_CONFIG` (or `AEC_SCHEDULE_CONFIG_<config>` if specifying multiple build configs in a single CMakeLists.txt) to the desired schedule in the application's CMakeLists.txt. For example:



```
set(AEC_SCHEDULE_CONFIG "1 2 2 10 5")
```

This string encodes:

```
<num_hw_threads>          <max_y_channels>          <max_x_channels>
<max_main_phases> <max_shadow_phases>
```

- ▶ **<num_hw_threads>**: number of hardware threads (supported up to a max of [AEC_LIB_MAX_THREADS](#) hardware threads).
- ▶ **<max_y_channels>**: compile-time maximum microphone channels (overrides [AEC_MAX_Y_CHANNELS](#)).
- ▶ **<max_x_channels>**: compile-time maximum reference channels (overrides [AEC_MAX_X_CHANNELS](#)).
- ▶ **<max_main_phases>**: compile-time maximum main-filter phases (overrides [AEC_MAIN_FILTER_PHASES](#)).
- ▶ **<max_shadow_phases>**: compile-time maximum shadow-filter phases (overrides [AEC_SHADOW_FILTER_PHASES](#)).

When `AEC_SCHEDULE_CONFIG` is set, the compilation process autogenerates:

- ▶ A `aec_task_distribution.c` file containing the task distribution schedule of type [aec_task_distribution_t](#) that targets `<num_hw_threads>` threads.
- ▶ A header file (`aec_conf.h`) that defines the macros above, overriding the library defaults.

The autogenerated files are added to the target sources and includes of the application target and get compiled accordingly. The autogenerated schedule is of the form:

```
aec_task_distribution_t tdist = { ...
```

To use it, in the application, declare the symbol:

```
extern aec_task_distribution_t tdist;
```

and pass `&tdist` as an argument to [aec_init\(\)](#).

Note

A given schedule would work for any runtime subset (fewer y/x channels or phases) as long as [aec_init\(\)](#) preconditions defined in [Preconditions](#) are met.

Parameters

The key AEC parameters are highlighted below:

- ▶ [aec_init\(\)](#) **num_main_filter_phases** - Number of phases for the main filter, typically 10-20. This determines the effective tail length of the main filter, with 15ms (240) samples per phase with the default [AEC_FRAME_ADVANCE](#). More phases allow for better echo cancellation in more reverberant environments, at the cost of increased computation and slower adaptation.
- ▶ [aec_init\(\)](#) **num_shadow_filter_phases** - Number of phases for the shadow filter, typically 5. This determines the effective tail length of the shadow filter, with



15ms (240) samples per phase. The shadow filter is designed to adapt more quickly than the main filter, so it typically has fewer phases than the main filter in order to quickly capture acoustic state changes.

- ▶ [`coherence_mu_config_params_t.mu_scalar`](#) - Scalar controlling the overall rate of the AEC filter adaption, set to 1.0 by default. When `adaption_config` is set to `AEC_ADAPTATION_FORCE_ON`, this value controls the rate of adaption for all frames. When `adaption_config` is set to `AEC_ADAPTATION_AUTO`, this value controls the relative rate of adaption. Values less than 1.0 will slow down the rate of adaption, which can improve stability in some environments, at the cost of slower convergence. Values greater than 1.0 will speed up the rate of adaption, which can improve convergence speed but may reduce stability and attenuation.
- ▶ [`coherence_mu_config_params_t.erle_thresh`](#) - The AEC adaption will be paused when the estimated ERLE (Echo Return Loss Enhancement) drops by this much relative to the long term average. Increasing this value can improve convergence in noisy environments, at the cost of increased deconvergence during near end noise or speech.
- ▶ [`coherence_mu_config_params_t.coh_thresh_abs`](#) - Sets the minimum coherence threshold for AEC adaption. The coherence is measured between the microphone signal and the estimated microphone signal. When the coherence is below this threshold, the AEC filters will not adapt. Decreasing this value can allow for faster convergence in noisy environments, at the cost of increased deconvergence during near end noise or speech.
- ▶ [`coherence_mu_config_params_t.coh_thresh_slow`](#) - Sets the relative coherence threshold for the current frame against the slow moving average. Reducing this value will allow frames with lower coherence than the average to adapt, which can improve convergence in noisy environments, at the cost of increased deconvergence during near end noise or speech.
- ▶ [`coherence_mu_config_params_t.adaption_config`](#) - Configures the adaption behaviour of the AEC. When set to `AEC_ADAPTATION_AUTO`, the AEC will automatically adjust the adaption rate based on the coherence and ERLE thresholds above. When set to `AEC_ADAPTATION_FORCE_ON`, the AEC will adapt on every frame regardless of the coherence or ERLE. When set to `AEC_ADAPTATION_FORCE_OFF`, the AEC will not adapt on any frames.
- ▶ [`REF_ACTIVE_THRESHOLD_DB`](#) - This macro sets the threshold for determining whether the reference signal is active, in decibels relative to full scale. When the maximum value of the reference signal in a frame is below this threshold, the AEC will consider it inactive and will pause adaption. If the reference signal is expected to be far below full scale for a reasonable SPL output, this threshold can be reduced to allow for adaption during low-level playback.

Other AEC parameters are described in the `aec_state.h` header file, and are described in detail in [`aec_config_params_t`](#).

1.2 Interference Canceller

An interference canceller (IC) removes unwanted point noise sources such as cooker hoods, washing machines, or radios for which there is no reference audio signal available. It achieves this by adapting a filter to cancel one microphone with the other. The filter is only updated when speech is not present, which allows the filter to converge to cancel the noise sources in the room. When speech occurs, the filter is held constant, which allows the speech to pass through while maintaining suppression of the noise sources. The IC uses the [*Voice to Noise Ratio Estimator*](#) to detect when speech is present. A delay



line is used to improve the causality of the IC filter by moving the peak towards the middle of the filter taps.

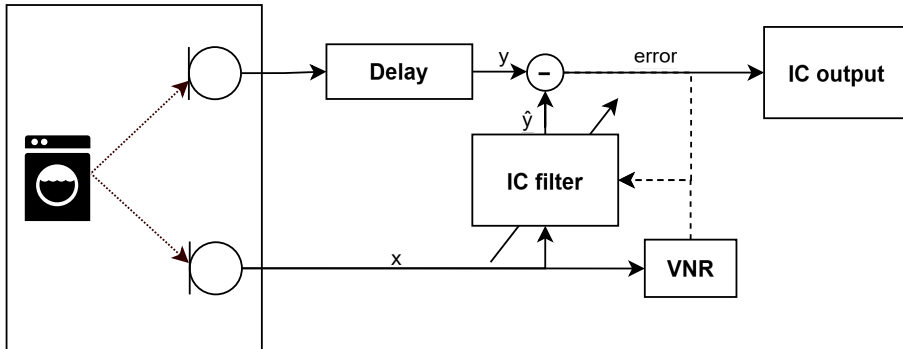


Fig. 3: The IC filter topology.

Overview

The IC component in **lib_voice** processes two microphone channels and attempts to cancel one microphone signal from the other in the absence of voice.

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. It uses the [Voice to Noise Ratio Estimator](#) for detecting presence or absence of voice. When the VNR indicates absence of speech, the IC adapts its filter to remove noise from the environment. When the VNR 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. The IC operates at a fixed 16 kHz sample rate and produces a single output channel.

Signal Representation

Processing is performed on a frame-by-frame basis. Each frame consists of 15 ms of new audio samples (240 samples at 16 kHz) per input channel, with a total of 2 input channels. Input data is expected in fixed-point 32-bit, 1.31 format. The output is the interference-cancelled primary microphone signal, in the same 32-bit, 1.31 format.

Adaptive Filter

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 IC filter has 10 phases, which effectively determines the tail length of the filter.

Processing Flow

For each frame, the IC performs the following steps:

1. Transform microphone signals into the frequency domain.



2. Estimate the interference using the adaptive filter.
3. Subtract the estimated interference from the primary microphone signal to produce the error signal.
4. Update the filter coefficients if the VNR indicates no speech.
5. Transform the error signal back to the time domain to produce the interference-cancelled output.

Usage

Before starting processing, the IC must be initialised by calling `ic_init()`, which sets up internal state of the IC. Once initialised, interference cancellation is performed by calling `ic_process_frame()` for each input frame (see [Pipeline example](#)). `ic_process_frame()` also outputs the VNR estimate for the current frame.

Parameters

The key IC parameters are highlighted below:

- ▶ `ic_adaption_controller_config_t.input_vnr_threshold` - If the VNR estimate for the current frame is above this threshold, the IC will suspend adaptation of its filter coefficients. This allows the IC to maintain suppression of noise sources while allowing speech to pass through. Lowering this threshold will cause the IC to adapt less often, which may reduce the amount of noise suppression but will allow more speech to pass through. This may be desirable in environments with higher levels of diffuse noise, which cannot be cancelled by the IC.
- ▶ `ic_adaption_controller_config_t.input_vnr_threshold_high` - If the input VNR estimate for the current frame is above this threshold, this indicates that the noise in the room is low, and the IC is not needed. The IC will suspend adaptation of its filter coefficients and will also leak the filter coefficients to reduce the amount of noise suppression applied to the output. This allows more speech to pass through at the cost of reduced noise suppression. Lowering this threshold will cause the IC to adapt less often, which may reduce the amount of noise suppression but will allow more speech to pass through.
- ▶ `IC_Y_CHANNEL_DELAY_SAMPS` - The ideal filter for the interference canceller is the deconvolution of the impulse responses between the noise source and each of the two microphones. This filter is non-causal, and the delay between the channels allows the IC to adapt to the ideal filter. However, this adds latency, which can be reduced by reducing `IC_Y_CHANNEL_DELAY_SAMPS`, at the cost of reduced IC performance.

Other IC parameters are described in the `ic_state.h` header file, and are described in detail in `ic_adaption_controller_config_t`.

1.3 Voice to Noise Ratio Estimator

The Voice to Noise Ratio estimator (VNR) estimates 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. A VNR value of 0.5 indicates a voice to noise ratio of -5 dB.

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 in the [Interference Canceller](#), and improved performance of the [Automatic Gain Control](#) blocks that provide a more natural listening experience.



The VNR operates on short frames of audio, transforming the input into the frequency domain and compressing it with a MEL filterbank. Features from the most recent frames are then fed into a pre-trained neural network that outputs the VNR estimate.

The VNR model is a pre-trained TensorFlow Lite model, optimised for the XCORE platform.

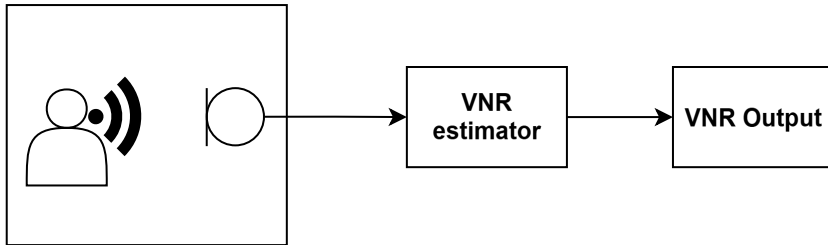


Fig. 4: The VNR topology.

Overview

The VNR component in **lib_voice** processes a single channel of microphone input, estimating the voice to noise ratio in the signal. The VNR operates at a fixed 16 kHz sample rate.

The VNR module processes **VNR_FRAME_ADVANCE** new audio PCM samples every frame. The time domain input is transformed to frequency domain using a **VNR_PROC_FRAME_LENGTH** 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.

Signal Representation

Processing is performed on a frame-by-frame basis. Each frame consists of 15 ms of new audio samples (240 samples at 16 kHz). Input data is expected in fixed-point 32-bit, 1.31 format. The output is a single VNR estimate value in **float_s32_t** format, ranging from 0 to 1, with 1 indicating the strongest speech and 0 indicating the weakest speech compared to noise. A VNR value of 0.5 indicates a voice to noise ratio of -5 dB.

Processing Flow

For each frame, the VNR performs the following steps:

1. Transform the input signal into the frequency domain using a 512-point DFT.
2. Compute the squared magnitude spectrum.
3. Apply a 24-band MEL filterbank to compress the frequency spectrum.
4. Add the new MEL filter output to a rolling buffer of the most recent 4 frames.
5. Normalise the feature patch by subtracting the maximum value.
6. Run inference on the normalised features using the pre-trained TensorFlow Lite model to produce the VNR estimate.



Basic Usage

To use the high-level API, initialise the VNR state by calling `vnr_state_init()`. Then, for each frame, call `vnr_process_frame()` to update VNR's internal state and produce VNR estimate. Refer to the [VNR example](#) to see how to initialise and run the VNR using the basic API.

Advanced Usage

The low-level API separates feature extraction from inference to allow multiple sets of features to share 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.

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.

Refer to the IC source code to see how to initialise and run the VNR using the advanced API.

Parameters

The VNR has no user configurable parameters.

1.4 Noise Suppressor

The Noise Suppressor (NS) reduces stationary noise in an audio signal so that speech is clearer and easier to process. It estimates the noise present in the signal and attenuates it in the frequency domain, producing a cleaner output.

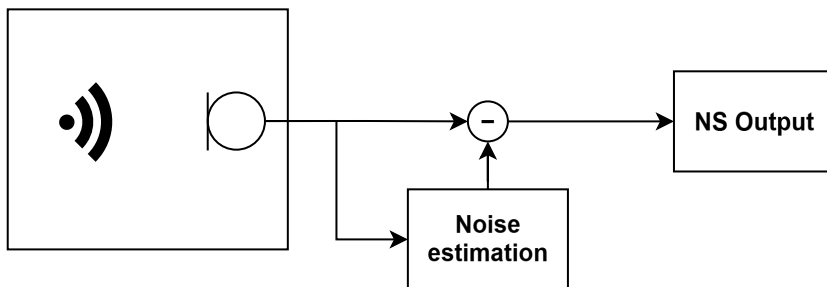


Fig. 5: The Noise Suppressor topology.



Overview

The NS operates in the frequency domain, continuously estimating noise and adapting to changing conditions. It continuously estimates the noise spectrum by tracking the smoothed input spectrum and its local minima, adapting the smoothing rate based on the probability of speech presence. The estimated noise is subtracted from the input spectrum (spectral subtraction), and the resulting signal is transformed back to the time domain for output.

The NS component in **lib_voice** works on a single input channel. 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.

Signal Representation

Noise suppression is performed on a frame-by-frame basis. Each frame consists of 15 ms of audio (240 samples at 16 kHz input sampling rate), with input data expected in fixed-point 32-bit 1.31 format. The output is the noise-suppressed signal in the same format.

Processing Flow

For each frame, the NS performs the following steps:

1. Form the input frame from new samples and previous samples.
2. Apply an analysis window and transform the signal into the frequency domain using a 512-point FFT.
3. Compute the magnitude spectrum.
4. Estimate the noise spectrum and apply spectral subtraction to attenuate noise.
5. Rescale the frequency-domain signal and transform back to the time domain using an inverse FFT.
6. Apply a synthesis window and form the output using overlap-add.

Usage

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. Refer to the [Pipeline example](#) to see how the APIs above are used.

Parameters

The Noise Suppressor has no user configurable parameters.

1.5 Automatic Gain Control

The Automatic Gain Control (AGC) component provides an API to implement Automatic Gain Control within an application. The AGC algorithm can dynamically adapt the audio gain, or apply a fixed gain such that voice content maintains a desired output level. The AGC uses a [Voice to Noise Ratio Estimator](#) to normalise voice content and avoid amplifying noise sources and applies a soft limiter to avoid clipping on the output. The design is based on standard modern AGC techniques as detailed in [Acoustic Echo and Noise Control](#) by Hansler and Schmidt.



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 the VNR, the AGC will adapt only when voice activity is detected, so that speech in the input signal is amplified above other sounds.

The Loss Control process improves the subjective audio quality by attenuating any residual echo of the reference far-end audio. It is designed to be used on the communications channel. In cases where there is both far-end echo and near-end audio then the attenuation is reduced, allowing listeners to interrupt each other. The Loss Control relies on the [Acoustic Echo Canceller](#) to classify and attenuate residual far-end echo.

An optional soft clipping stage is applied at the end of the AGC to avoid hard clipping of the output signal during sudden loud sounds.

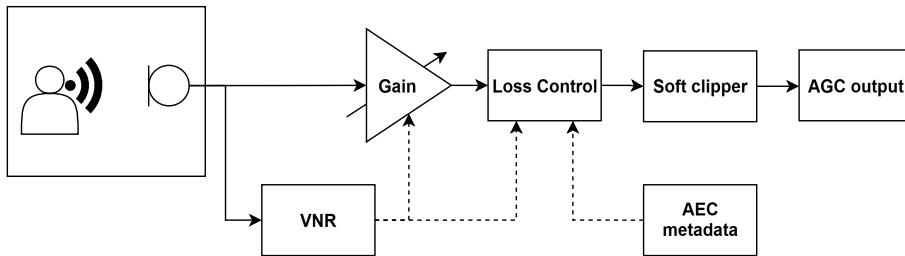


Fig. 6: The AGC topology.

Overview

The AGC component in **lib_voice** works on a single input channel, dynamically adapting the audio gain to maintain voice content at a desired output level while avoiding amplification of noise sources. The AGC operates at a fixed 16 kHz sample rate.

The AGC uses the [Voice to Noise Ratio Estimator](#) to detect voice activity and normalise voice content, ensuring that gain adaption only occurs during speech and not on noise. An optional Loss Control feature can be enabled to attenuate residual far-end echo using metadata from the [Acoustic Echo Canceller](#). A soft limiter can also be applied to prevent clipping on the output.

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.

Signal Representation

Gain control is performed on a frame-by-frame basis. Each frame consists of 15 ms of audio (240 samples at 16 kHz input sampling frequency), with input data expected in fixed-point 32-bit 1.31 format. The output is the gain-adjusted signal in the same format.

Processing Flow

The internal logic of the AGC algorithm is represented in the flow chart shown in [Fig. 7](#). This diagram illustrates the main decision points and processing steps performed for each input frame. It shows how the AGC determines whether to adapt the gain based on voice activity, applies peak and threshold checks, manages loss control, and optionally performs soft clipping.

The logic of the loss control process is shown in the flow chart in [Fig. 8](#). This diagram illustrates how the loss control estimates the state and applies the appropriate attenua-



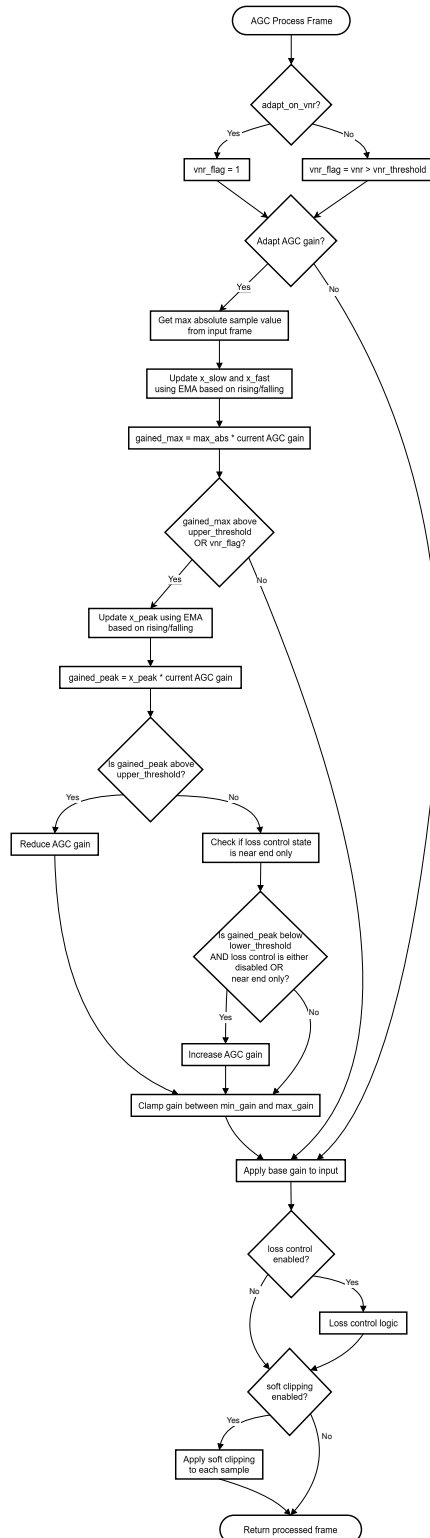


Fig. 7: AGC Logic Flow Chart



tion based on the presence of far-end echo and near-end audio. It is only used when the loss control feature is enabled in the AGC configuration.

A startup delay can be configured to mute output for a specified number of frames after initialisation.

Usage

Before processing any frames, the application must configure and initialise the AGC instance by calling `agc_init()`. Several parameter sets are provided in `agc_profiles.h` which can be used to configure the AGC for different applications. Details on the profiles and key parameters are provided in [AGC Data Structures and Enums](#).

After initialisation, `agc_process_frame()` should be called for each frame. This will update the AGC instance's internal state and produce the output frame by applying the AGC algorithm to the input frame. Refer to the [Pipeline example](#) to see how to use the APIs above.

The AGC gain and Loss Control gain values 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.

Parameters

The key AGC parameters are highlighted below:

- ▶ `agc_config_t.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.
- ▶ `agc_config_t.vnr_threshold` - VNR threshold for voice activity detection. A higher value will only adapt the AGC on clean speech. A lower value will adapt the AGC on noisy speech, but may also adapt to more non-speech signals.
- ▶ `agc_config_t.gain` - The current gain to be applied, not including loss control. When `adapt` is false, this gain will be applied to every frame. When `adapt` is true, the initial value of this gain will be applied to the first frame and then it will be adapted on subsequent frames.
- ▶ `agc_config_t.max_gain` - The maximum gain allowed when adaption is enabled. This can be used to prevent the AGC amplifying very quiet signals.
- ▶ `agc_config_t.upper_threshold` - The target maximum peak level of the AGC output. If the AGC output goes above this level, the gain is reduced.
- ▶ `agc_config_t.lower_threshold` - The target minimum peak level of the AGC output. If the AGC output goes below this level, the gain is increased.
- ▶ `agc_config_t.soft_clipping` - Boolean to enable soft-clipping of the output frame.
- ▶ `agc_config_t.lc_enabled` - Boolean to enable loss control. The loss control applies additional attenuation when there is no near end speech. This must be disabled if the application doesn't have an AEC or VNR.
- ▶ `agc_config_t.lc_near_delta` - Delta multiplier used when only near-end activity is detected. How many times louder the near-end signal must be than the background noise when there is no far-end playback. If the near end speech is not heard during



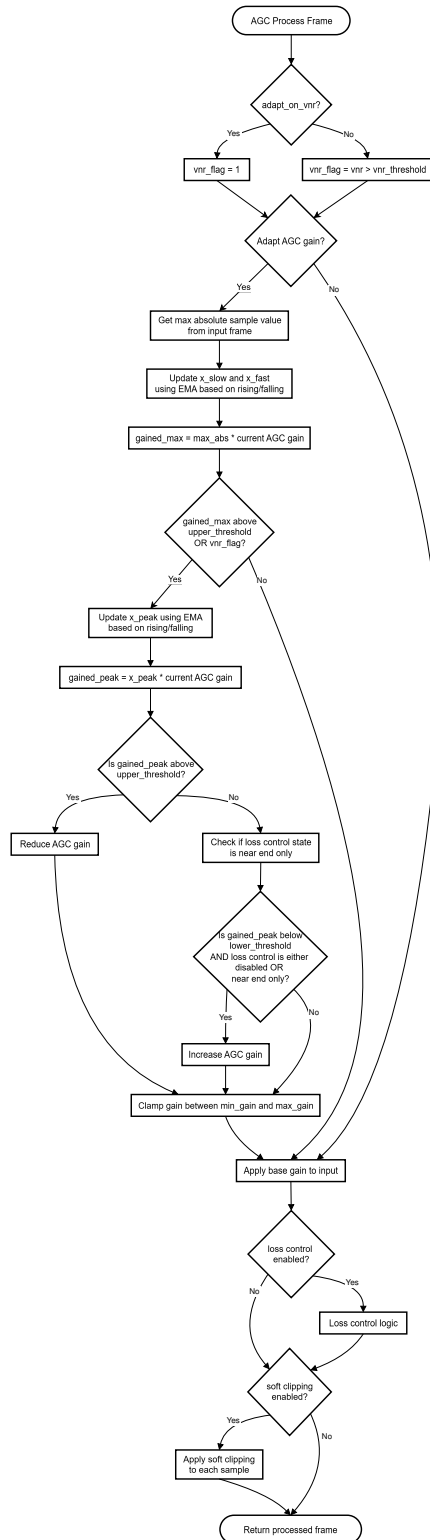


Fig. 8: Loss Control Logic Flow Chart



silence, reduce this value. If too much non-speech background noise is heard, increase this value.

- ▶ [`agc_config_t.lc_near_delta_far_active`](#) - Delta multiplier used when both near-end and far-end activity is detected. How many times louder the near end signal must be above the residual far-end speech (after the AEC) to be detected during double talk. If the near end speech is not heard during double talk, reduce this value. If there is too much breakthrough of residual far-end echo when there is no near-end speech present, increase this value.
- ▶ [`agc_config_t.lc_gain_double_talk`](#) - Loss control gain to apply when double-talk is detected. Reducing this value will reduce the level of the near-end speech during double-talk, but may help to reduce the level of residual far-end echo that is heard.

Other AGC parameters are described in the `agc_profiles.h` header file, and are described in detail in [`agc_config_t`](#).

1.6 Automatic Delay Estimation and Correction

The Automatic Delay Estimation and Correction (ADEC) module provides functions to estimate and automatically correct for delay offsets between the reference and the loudspeakers. ADEC is designed to be used in combination with an [*Acoustic Echo Canceller*](#).

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 basic topology of ADEC is shown in [Fig. 9](#). ADEC requests a delay correction value that the application can use to insert delay into either the reference or microphone signal path. When the loudspeaker signal lags behind the reference signal, the application should delay the reference channel. When the reference signal lags behind the loudspeaker, the application should delay the microphone channel.

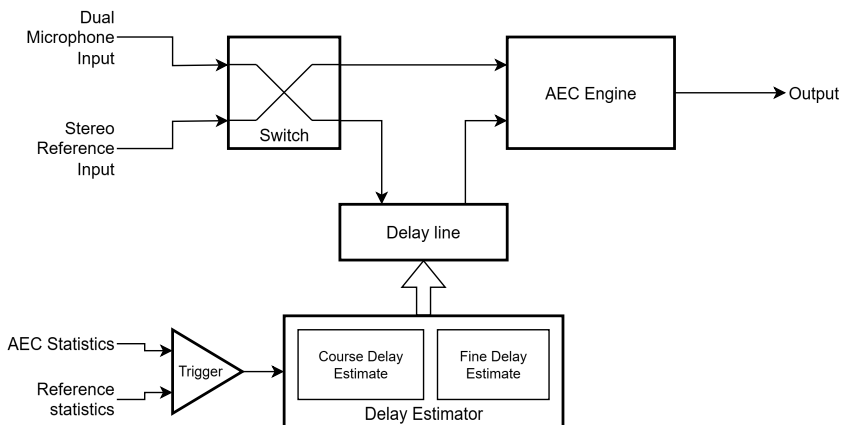


Fig. 9: The basic topology of ADEC. ADEC allows the relative delay between the microphone and reference signals to be estimated and corrected.



AEC Delays and Alignment

The time window modelled by the Acoustic Echo Canceller (AEC) is finite due to the filter tail length. 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. Fig. 10 shows a well aligned AEC filter, where the whole impulse response is captured. Fig. 11 and Fig. 12 show examples of poorly aligned AEC filters where the reference audio arrives either too early or too late with respect to the microphone signal, resulting in a start of the filter filled with zeros or the loss of the tail of the impulse response.

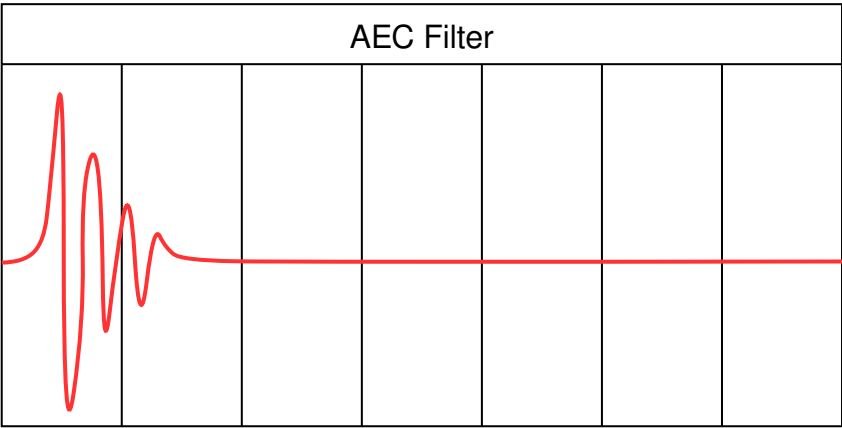


Fig. 10: In this AEC filter, the reference signal and microphone signal are well aligned, and the full impulse response is captured within the filter.

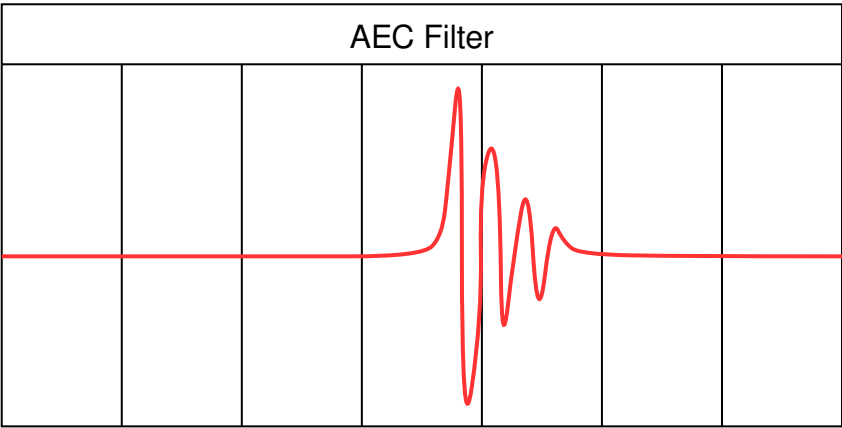


Fig. 11: In this AEC filter, the reference signal arrives before the microphone signal, the start of the filter is filled with zeros and the tail of the impulse response is lost.

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. This can be achieved by either using a fixed delay or ADEC, depending on the system design.



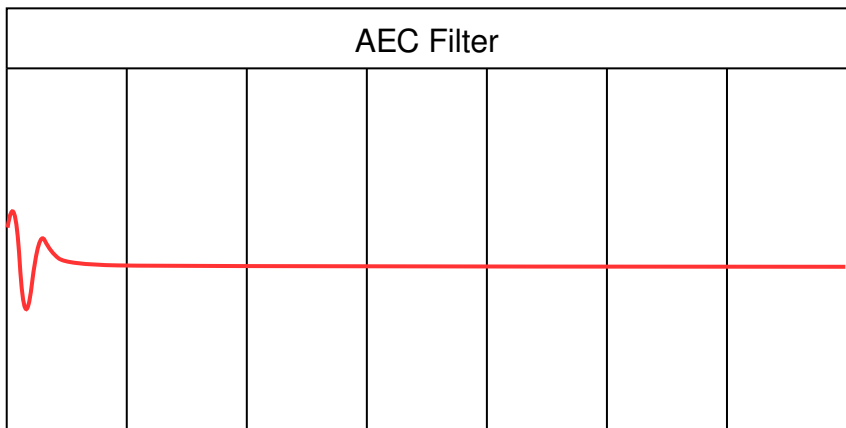


Fig. 12: In this AEC filter, the microphone signal arrives before the reference signal, and the start of the impulse response is lost.

Systems not requiring ADEC

In an ideal system design, the relative delay between the reference audio and microphones is well known and fixed. In these systems ADEC should not be used, and a fixed delay should be applied to either the reference or microphone signals to ensure they are time aligned at the input of the AEC. Examples of systems that do not need ADEC are shown in Fig. 13 and Fig. 14. In both these systems, a fixed delay can be used instead.

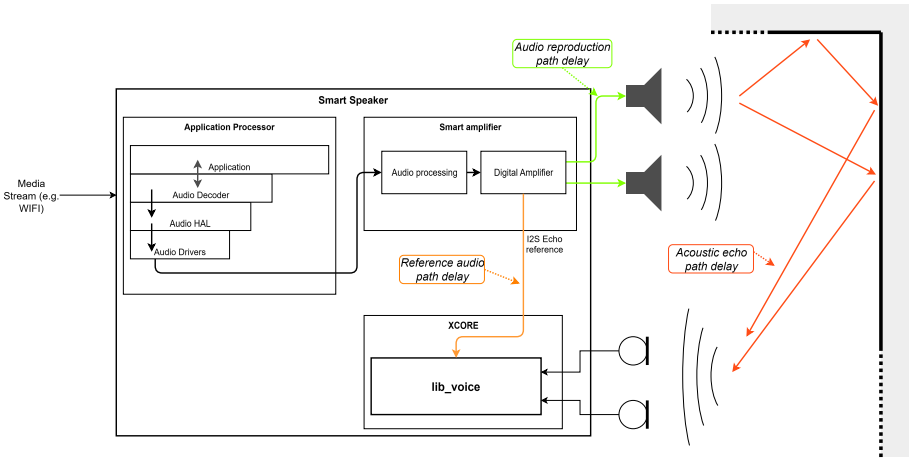


Fig. 13: In this system, the reference audio is sent to the XCORE device over I2S from the smart amplifier chipset. This will result in a well defined and fixed delay between the reference audio and the microphone signals, which can be compensated for by applying a fixed delay to either the reference or microphone signals as required, without the need for ADEC.



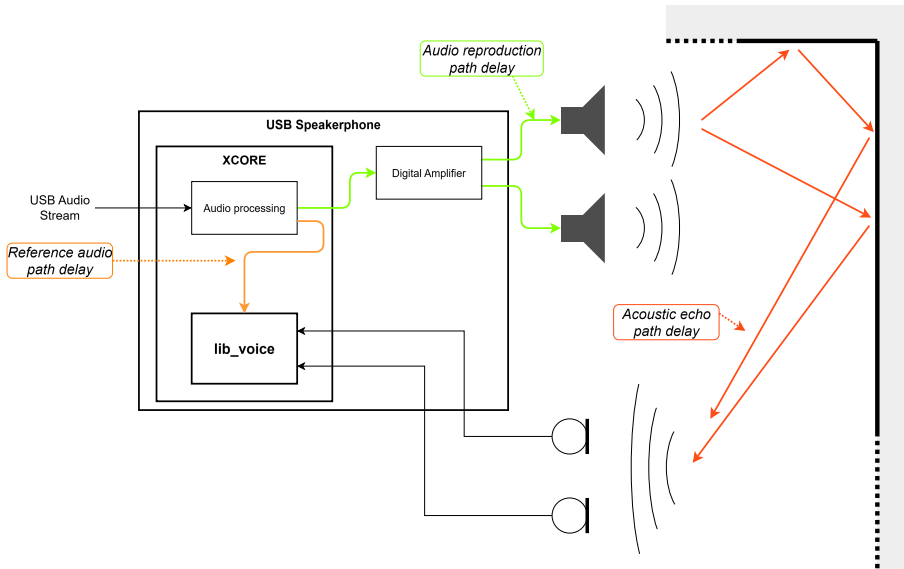


Fig. 14: In this system, the reference audio is sent to the XCORE device over USB from a host device, and the XCORE device sends the signal to the amplifier and loudspeakers. A duplicate of the signal is sent to the AEC. The timing over USB may vary, but the relative delay between the loudspeaker and microphone signals is fixed, so ADEC is not required.

Systems requiring ADEC

In some applications, it is not possible to know the relative delay between the reference audio and microphone signals, or the delay may vary over time. For example, in a Smart TV, the audio may be played out of the TV's internal loudspeakers or an external soundbar, and the delay will be different in these two scenarios. An example of this system is shown in Fig. 15. In these applications, ADEC can be used to ensure the reference and microphone signals remain time aligned at the input of the AEC, even when the delay changes.

Overview

Automatic delay estimation can either be triggered at power-up, manually if the host system configuration changes, or automatically when the AEC performance degrades due to delay changes in the system. For most applications (where a fixed delay cannot be used) it is recommended to enable ADEC on boot, so that the AEC starts with the correct delay and can converge as quickly as possible.

It is advised to only enable the automatic mode of ADEC if it cannot be predicted when delay changes will occur. This is because the automatic mode relies on monitoring the AEC performance and can take some time to react to delay changes, during which the AEC performance may be poor. If the system design allows, it is recommended to trigger delay estimation manually when a delay change is expected, for example by sending a command from the host when the user changes the TV's audio output settings.



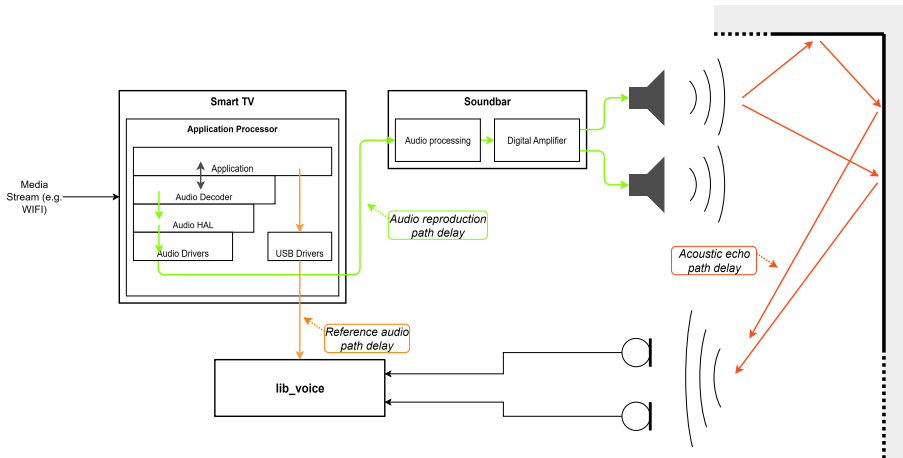


Fig. 15: In this system, the reference audio is sent to the XCORE device over USB from a host device. The audio reproduction path goes through the Smart TV's internal drivers, before being processed in the soundbar. Different models of soundbar would have different latencies, and the latency may also change based on the TV's audio settings, so the relative delay between the reference and microphone signals may be unknown and variable, requiring ADEC to maintain time alignment at the input of the AEC.

Processing flow

The ADEC process will not begin until the reference signal is present and has sufficient energy.

During normal operation, ADEC monitors the AEC performance and can make decisions about when to apply delay corrections based on the estimated delay and the AEC performance related metadata collected from the echo canceller. 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. ADEC uses the metadata to measure the AEC goodness, and will modify the delay if the goodness falls.

Possible causes that may trigger an estimation cycle (where automatic mode is enabled):

- ▶ Host changing applications causing a delay change between loudspeakers and reference.
- ▶ Large volume changes between the reference and the loudspeaker play-back.
- ▶ User equipment changes, such as switching from TV audio output to playing the audio through a soundbar.

If the goodness falls, ADEC will first try to correct the delay by applying a small delay correction to the input of the echo canceller.

If the goodness does not improve after the small delay correction, ADEC will transition to delay estimation mode. The delay estimation process re-purposes the AEC to detect larger delays. During estimation, the AEC does not perform cancellation. On entering delay estimation, ADEC requests:



- ▶ 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 with respect to the reference input.
- ▶ A restart of AEC in a new configuration that has more adaptive filter phases, in order to 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. The AEC restarts and reconverges based on the corrected delay, and ADEC goes back to its normal mode of monitoring AEC performance and correcting for small delay offsets.

Usage

For most applications where a fixed delay cannot be used, it is recommended to enable ADEC on boot so the AEC starts with correct delay alignment and converges quickly. Enabling automatic mode should be reserved for systems where delay changes cannot be predicted, as ADEC relies on monitoring AEC performance and may take time to react to changes. When possible, trigger delay estimation manually (via `force_de_cycle_trigger`) when delay changes are expected, such as when users change audio output settings.

Before processing any frames, the application must configure and initialise the ADEC instance by calling `adec_init()`. This function takes a pointer to the `adec_state_t` structure and a pointer to an `adec_config_t` configuration structure. Example configurations can be seen in the [Configuration](#) section below.

For each frame of audio, the application should:

1. Call `adec_estimate_delay()` with the AEC filter coefficients to estimate the current delay. This function analyzes the AEC filter phases to determine the peak energy location, which indicates the delay.
2. Call `adec_process_frame()` with the delay estimate and AEC statistics. This function uses the delay estimate along with AEC performance metrics (ERLE, peak power, etc.) to make decisions about whether delay correction or AEC reconfiguration is needed.

Refer to the [Pipeline Stage 1](#) source code to see how to use these APIs and to the [Pipeline example](#) to see how to use ADEC within the Stage1 API.

Configuration

ADEC is configured through the `adec_config_t` structure passed to `adec_init()`. This configuration can also be modified at runtime by accessing the `adec_config` member of the `adec_state_t` structure.

Example configurations are provided in the `adec_init()` API documentation. See the doxygen comments in `lib_voice/api/adec/adec.h` for complete code examples showing:

- ▶ ADEC configured for delay estimation only at startup (with `bypass = 1`)
- ▶ ADEC configured for automatic delay estimation and correction (with `bypass = 0`)

Note that when ADEC is enabled on boot, the AEC will initially be used for delay estimation. As a result the AEC convergence will not begin until ADEC has estimated the delay and reconfigured the AEC to normal mode. This can result in poor echo cancellation for the first few seconds after boot.



Delay buffer constraints The maximum delay that can be corrected is limited by the delay buffer size, which is defined by [ADEC_DE_DELAY_SAMPS](#) (corresponding to [ADEC_DE_DELAY_MS](#) milliseconds, default 150 ms). The ADEC can measure and correct delays within the range of ± 150 ms relative to the current delay setting.

When requesting delay corrections, ADEC includes a headroom margin (defined by [ADEC_DE_DELAY_HEADROOM_SAMPS](#), default 240 samples or 15 ms) to ensure that the microphone signal does not arrive earlier than the reference signal, which would prevent the AEC filter from converging due to lack of causality.

Manually triggering ADEC ADEC can be manually triggered to start a delay estimation cycle by setting the [force_de_cycle_trigger](#) member of the ADEC configuration to 1. This can be used to get an initial delay estimate at boot, or to trigger a new estimation cycle when a delay change is expected, for example when the user changes the TV's audio output settings.

Parameters

ADEC has two user configurable parameters, both part of the [adec_config_t](#) structure:

- ▶ [adec_config_t.bypass](#) - When set to 1, ADEC evaluates the current input frame metrics but does not make any delay correction or AEC reset and reconfiguration requests. This is useful for testing or when delay correction needs to be temporarily disabled. Default: 0.
- ▶ [adec_config_t.force_de_cycle_trigger](#) - When set to 1, ADEC bypasses the normal monitoring process and immediately transitions to delay estimation mode for measuring the delay offset. This is useful for triggering delay estimation at startup or when the system configuration is known to have changed. When using ADEC through the [Pipeline Stage 1](#) (recommended), this flag is automatically reset to 0 after triggering the delay estimation transition. If using ADEC directly, the application must manually reset this flag to prevent repeated DE cycles. Default: 0.

Both parameters can be modified at runtime by accessing the [adec_config](#) member of the [adec_state_t](#) structure.

1.7 Pipeline Stage 1

Stage1 is typically the first stage in an audio pipeline. It orchestrates delay alignment, Acoustic Echo Cancellation (AEC), and Adaptive Delay Estimation/Canceler (ADEC), and propagates per-frame metadata downstream.

Overview

The Stage1 component in [lib_voice](#) integrates the [Acoustic Echo Canceller](#), [Automatic Delay Estimation and Correction](#), and delay buffering to provide echo-cancelled audio with automatic delay correction. Stage1 operates at a fixed 16 kHz sample rate.

Stage1 manages the transition between normal AEC operation and delay estimation mode, applies delay corrections to maintain optimal AEC performance, and generates metadata (reference energy, correlation factors, and activity flags) for downstream processing stages.

Two pipeline architectures are supported:

- ▶ **Standard Architecture:** Processes multiple microphone channels through both AEC and IC sequentially



- **Alternating Architecture:** Selectively enables AEC or IC based on reference signal presence, reducing memory requirements and enabling longer AEC filter tails

Signal Representation

Stage1 processes audio on a frame-by-frame basis. Each frame consists of 15 ms of audio (240 samples at 16 kHz), with input and output data in fixed-point 32-bit 1.31 format.

Inputs:

- Microphone (Y) channels: Up to 2 channels of microphone input
- Reference (X) channels: Up to 2 channels of reference (loudspeaker) input

Outputs:

- Echo-cancelled audio: Same number of channels as microphone input
- Metadata: Maximum reference energy, AEC correlation factors, reference activity flag

Standard Architecture

In the Standard Architecture pipeline form, all the modules are enabled and called sequentially. This is shown in Fig. 16.

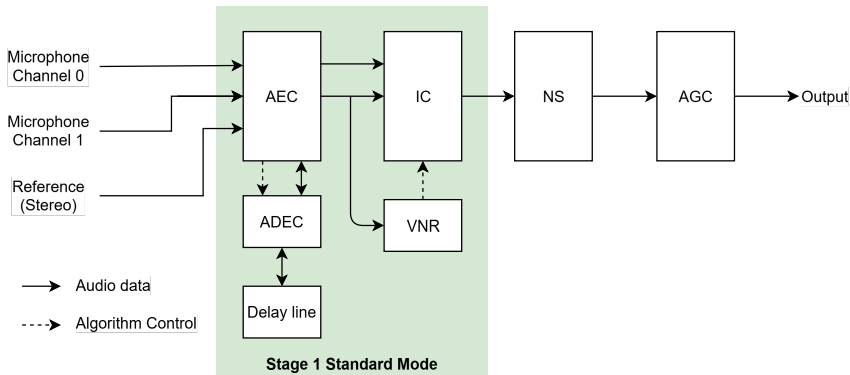


Fig. 16: The Standard Architecture Pipeline.

The AEC is configured for 2 mic input channels, 2 reference input channels, 10 phase main filter and a 5 phase shadow filter. The IC is configured for 2 mic input channels, and 10 phase main filter.

When ADEC goes in delay estimation mode, the AEC gets reconfigured as a 1 mic input channel, 1 reference input channel, 30 main filter phases and no shadow filter, as described in the [Automatic Delay Estimation and Correction](#) documentation. During this, the IC remains active.

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. The AEC stage generates the echo cancelled version of the mic input that is then sent for processing through the IC.



Alternating Architecture

In this pipeline form, the AEC and the IC frame processing are selectively enabled and disabled based on the presence of reference input signal. This is shown in [Fig. 17](#).

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.

This means that only 1 microphone signal requires processing by the AEC, reducing the number of filters required. This saves memory, which can then be used to increase the AEC filter tail length. This can improve AEC performance in more reverberant environments.

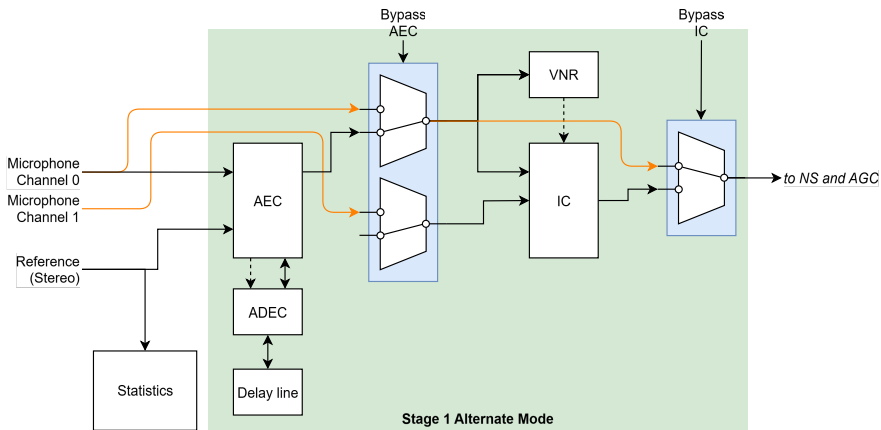


Fig. 17: The Alternating Architecture Stage 1.

When reference audio is detected, the AEC is enabled and the IC is disabled. The AEC processes one microphone input to remove any echo from the reference signal. This is shown in [Fig. 18](#). Note the VNR output from the IC is still generated so that it can be used by the AGC.

When no reference audio is detected, the AEC is disabled and the IC is enabled. The IC processes both microphone inputs to remove any unwanted noise sources in the environment. This is shown in [Fig. 19](#).

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 for highly reverberant environments.

When ADEC goes in delay estimation mode, the AEC gets reconfigured as a 1 mic input channel, 1 reference input channel, 30 main filter phases and no shadow filter, as described in the [Automatic Delay Estimation and Correction](#) documentation. In the absence of activity on the reference channels, when the AEC is disabled, the microphone input is copied directly to the output of the AEC.

Alternating architecture is disabled by default (see [ALT_ARCH_MODE](#)). To enable it, define `ALT_ARCH_MODE` to 1 in the application's CMakeLists.txt.



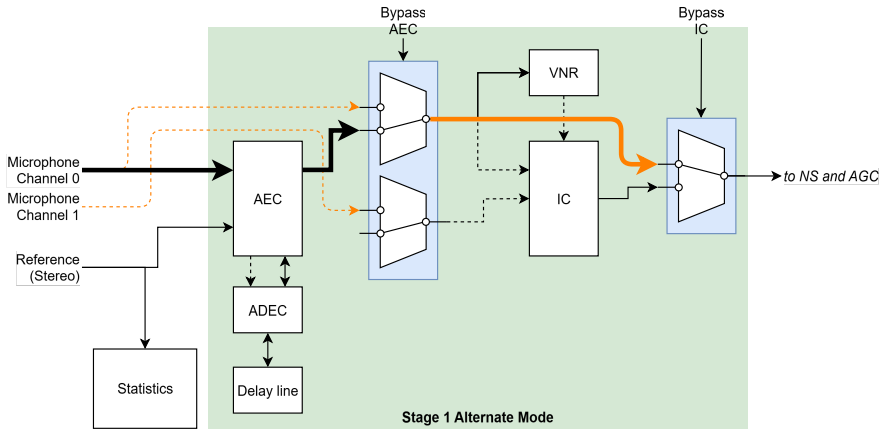


Fig. 18: The Alternating Architecture Stage 1 when the reference signal is present.

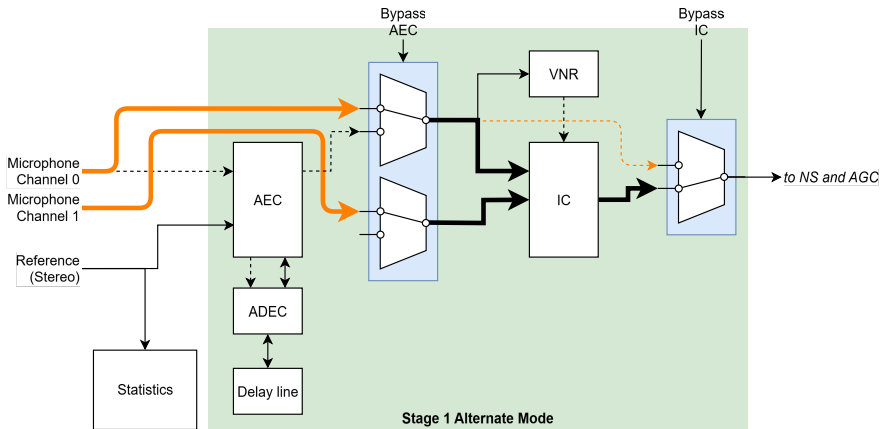


Fig. 19: The Alternating Architecture Stage 1 when no reference signal is present.

Usage

Before starting processing, Stage1 must be initialised by calling `stage1_init()`. This sets up internal state for the provided runtime AEC configurations and ADEC settings.

Once initialised, call `stage1_process_frame()` for each input frame.

Refer to [Pipeline example](#) to see Stage1 integrated into an audio pipeline.

Parameters

The key Stage 1 parameters are highlighted below:

- `REF_ACTIVE_THRESHOLD_DB` - This macro is used in alt arch mode, and sets the threshold for determining whether the reference signal is active, in decibels relative to full scale. When the maximum value of the reference signal in a frame is below this



threshold for **HOLD_AEC_LIMIT_SECONDS**, the AEC will be bypassed and the IC will be enabled. If the reference signal is above this level, the AEC will be enabled and the IC bypassed. Note this parameter is shared with the AEC module.

- **HOLD_AEC_LIMIT_SECONDS** - This macro is used in alt arch mode, and sets the limit in seconds for which AEC is kept enabled after detecting reference as inactive. This is to avoid toggling of AEC and IC when the reference signal is fluctuating around the reference active threshold.



2 Examples

This section goes through the various example applications that are provided alongside the **lib_voice**. The examples are meant to demonstrate the basic usage of the APIs. All examples are located in the **examples** directory.

2.1 AEC example

The AEC example demonstrates the use of API to run a frame of data through the AEC. It also shows how to configure AEC to run on one or two threads. **AEC_THREADS** define toggles the AEC to use 1-threaded or 2-threaded scheduling structs:

```
#if AEC_THREADS == 1
aec_task_distribution_t tdist = aec_tdist_chans2_threads1;
#elif AEC_THREADS == 2
aec_task_distribution_t tdist = aec_tdist_chans2_threads2;
#else
#error "Unsupported number of AEC threads"
#endif
```

The build system will automatically create both configs called **app_aec_1th** and **app_aec_2th**.

After the application has decided which thread distribution scheme to run, it allocates memory for the AEC and initialises it:

```
// Allocate signal data
int32_t DWORD_ALIGNED frame_y[AEC_MAX_Y_CHANNELS][AEC_FRAME_ADVANCE];
int32_t DWORD_ALIGNED frame_x[AEC_MAX_X_CHANNELS][AEC_FRAME_ADVANCE];

// Initialise AEC
aec_state_t DWORD_ALIGNED aec_state;
aec_init(&aec_state,
        AEC_MAX_Y_CHANNELS, AEC_MAX_X_CHANNELS,
        AEC_MAIN_FILTER_PHASES, AEC_SHADOW_FILTER_PHASES, &tdist);
```

After the AEC has been initialised, it is ready to process data. In this example, **frame_y** and **frame_x** memory is reused for AEC output:

```
producer(frame_y, frame_x);
// Reuse mic data memory for the main filter output
// Reuse ref data memory for the shadow filter output
aec_process_frame(&aec_state, frame_y, frame_x, frame_y, frame_x);
consumer(frame_y);
```

Upon execution, the example will print "frame done" when the AEC has processed a frame.

2.2 VNR example

The VNR example demonstrates the use of API to run a frame of data through the VNR and get the estimation of how much voice is present in it.

To run the VNR, the application should first allocate memory for its data and initialise it:

```
// Allocate input and output memory
int32_t input[VNR_FRAME_ADVANCE] = {0};
float_s32_t vnr_out;

// Initialise VNR
vnr_state_t vnr;
vnr_state_init(&vnr);
```

After the VNR has been initialised, it is ready to process data. The VNR output is in **float_s32_t** format, so there's an extra step if the user wants it in **float**.

```
producer(input);
vnr_process_frame(&vnr, &vnr_out, input);
float res = float_s32_to_float(vnr_out);
consumer(res);
```



Upon execution, the example will use the pseudo-random generator to get an input data. This data will be run through the VNR and the score will be printed in the terminal. 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. The pseudo-random data is not representative of a real signal, so the VNR scores in this example tend to be zero.

2.3 Pipeline example

This example demonstrates how to put together a voice processing pipeline. The pipeline will have 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.

There are two pipelines supported in this example: Standard Architecture and Alternating Architecture. Building one or the other config is controlled via the `ALT_ARCH_MODE` define. The build system will automatically create both configs called `app_pipeline_std_arch` and `app_pipeline_alt_arch`.

To create the pipeline, the application must first initialise all the individual components:

```
// Initialise AEC, DE, ADEC stages
aec_conf_t aec_de_mode_conf, aec_non_de_mode_conf;
#ifdef ALT_ARCH_MODE
aec_non_de_mode_conf.num_y_channels = 1;
aec_non_de_mode_conf.num_x_channels = AP_MAX_X_CHANNELS;
aec_non_de_mode_conf.num_main_filt_phases = 15;
aec_non_de_mode_conf.num_shadow_filt_phases = AEC_SHADOW_FILTER_PHASES;
#else
aec_non_de_mode_conf.num_y_channels = AP_MAX_Y_CHANNELS;
aec_non_de_mode_conf.num_x_channels = AP_MAX_X_CHANNELS;
aec_non_de_mode_conf.num_main_filt_phases = AEC_MAIN_FILTER_PHASES;
aec_non_de_mode_conf.num_shadow_filt_phases = AEC_SHADOW_FILTER_PHASES;
#endif
aec_non_de_mode_conf.tdist = &aec_tdist_chans2_threads1;

aec_de_mode_conf.num_y_channels = 1;
aec_de_mode_conf.num_x_channels = 1;
aec_de_mode_conf.num_main_filt_phases = 30;
aec_de_mode_conf.num_shadow_filt_phases = 0;
aec_de_mode_conf.tdist = &aec_tdist_chans2_threads1;

// Disable ADEC's automatic mode. We only want to estimate and correct for the delay at startup
adec_config_t adec_conf;
adec_conf.bypass = 1; // Bypass automatic DE correction
#ifdef DISABLE_INITIAL_DELAY_EST
// Do not force a DE correction cycle on startup
adec_conf.force_de_cycle_trigger = 0;
#else
// Force a delay correction cycle, so that delay correction happens once after initialisation.
// Make sure this is set back to 0 after adec has requested a transition into DE mode once,
// to stop any further delay correction (automatic or forced) by ADEC
adec_conf.force_de_cycle_trigger = 1;
#endif
stage1_init(&state->stage_1_state, &aec_de_mode_conf, &aec_non_de_mode_conf, &adec_conf);

// Initialise IC, VNR
ic_init(&state->ic_state);

// Initialise NS
ns_init(&state->ns_state);

// Initialise AGC
agc_config_t agc_conf_asr = AGC_PROFILE_ASR;
agc_init(&state->agc_state, &agc_conf_asr);
```

After the pipeline has been initialised, the data can be run through it. All the modules in the example can run on separate threads. To exchange information between the pipeline stages the metadata struct will need to be created to be populated and consumed by the different modules.

```
/** Stage1 - AEC, DE, ADEC*/
// stage1 will not process the frame in-place,
// since mic input is needed to overwrite the output in certain cases
```

(continues on next page)



(continued from previous page)

```

int32_t stage_1_out[AEC_MAX_Y_CHANNELS][AP_FRAME_ADVANCE];

stage1_process_frame(&state->stage_1_state, &stage_1_out[0], &md.max_ref_energy,
                    &md.aec_corr_factor, &md.ref_active_flag, input_y_data, input_x_data);

// Bypass IC if the reference is high in the alt arch mode
#if ALT_ARCH_MODE
if(md.ref_active_flag) {
    state->ic_state.config_params.bypass = 1;
}
else {
    state->ic_state.config_params.bypass = 0;
}
#endif
/** IC and VNR*/
int32_t ic_output[AP_FRAME_ADVANCE];
float_s32_t input_vnr_pred;

ic_process_frame(&state->ic_state, input_data[0], input_data[1], ic_output, &input_vnr_pred);
md.vnr_pred_flag = input_vnr_pred;

/** NS*/
int32_t ns_output[AP_FRAME_ADVANCE];

ns_process_frame(&state->ns_state, ns_output, ic_output);

/** AGC*/
agc_meta_data_t agc_md;
agc_md.aec_ref_power = md.max_ref_energy;
agc_md.vnr_flag = md.vnr_pred_flag;
agc_md.ref_active_flag = md.ref_active_flag;
agc_md.aec_corr_factor = md.aec_corr_factor;

agc_process_frame(&state->agc_state, output_data, ns_output, &agc_md);

```

Upon execution, the example will print “frame done” when the AEC has processed a frame.

2.4 Building the example

This section assumes that the [XMOSES XTC Tools](#) have been downloaded and installed. It also assumes that the Python is installed and available. The required versions are specified in the accompanying [README](#).

Installation instructions can be found [here](#).

Special attention should be paid to the section on [Installation of Required Third-Party Tools](#).

The application is built using the [xcommon-cmake](#) build system, which is provided with the XTC tools and is based on [CMake](#).

The **lib_voice** software ZIP package should be downloaded and extracted to a chosen working directory.

To configure the build, the following commands should be run from an XTC command prompt:

```

cd lib_voice
pip install -r requirements.txt
cd examples/app_aec
cmake -G "Unix Makefiles" -B build

```

Note

The `pip install -r requirements.txt` stage has to happen before the `cmake` configuration as it will fetch the [xmos-ai-tools](#).



If any dependencies are missing they will be retrieved automatically during this step.

The application binaries should then be built using **xmake**:

```
xmake -j -C build
```

Binary artifacts (.xe files) will be generated under the appropriate subdirectories of the **app_aec/bin** directory — one for each supported build configuration.

For subsequent builds, the **cmake** step may be omitted. If **CMakeLists.txt** or other build files are modified, **cmake** will be re-run automatically by **xmake** as needed.

2.5 Running the example

From an XTC command prompt, the following command should be run from the **examples/app_aec** directory:

```
xrun --io ./bin/2th/app_aec_2th.xe
```



3 Resource usage

3.1 Memory

Table 1 lists the memory requirements for the **lib_voice** DSP components. The estimates are provided as a guideline to help audio pipeline developers assess the memory cost of including a component in the pipeline.

Note

The IC memory usage includes VNR memory usage as well, since VNR processing is part of IC. Stand-alone VNR memory usage is also computed, as shown in the VNR row of the table.

Table 1: Memory usage (in bytes)

Component	Memory use (bytes)
ADEC	8840
AEC_ALT_ARCH	240400
AEC_STD_ARCH	277824
AGC	9896
IC	136344
NS	35664
VNR	62936

3.2 CPU

Table 2 lists the approximate CPU requirements in MIPS for the **lib_voice** DSP components. The MIPS values are computed as the worst-case processor cycles consumed by the component's **process_frame** function. All CPU estimates are based on the default configuration for each feature. Alternate configurations may require more or less MIPS.

These estimates are provided as a guideline to help audio pipeline developers assess the CPU cost of including a component in the pipeline.

Note

The IC MIPS numbers include VNR processing that is part of IC. Stand-alone VNR processing numbers are also computed, as shown in the VNR row of the table.



Table 2: CPU requirements (600 MHz system frequency, 120 MHz per HW thread)

Component	MIPS use
ADEC	1.53
AEC_ALT_ARCH_1THREAD	42.57
AEC_ALT_ARCH_2THREADS	30.7
AEC_STD_ARCH_1THREAD	62.89
AEC_STD_ARCH_2THREADS	40.85
AGC	21.31
IC	13.21
NS	20.79
VNR	1.84



4 API reference

This section will provide a comprehensive walkthrough of **lib_voice**'s APIs, defines and structs.

4.1 AEC

AEC API Functions

group **AEC API Functions**

Functions

```
void aec_init(
    aec_state_t *aec_state, unsigned num_y_channels, unsigned
    num_x_channels, unsigned num_main_filter_phases, unsigned
    num_shadow_filter_phases, const aec_task_distribution_t *tdist,
)
```

Initialise the AEC for a given configuration.

This initializes the aggregated AEC state for the supplied runtime configuration (number of channels and filter phases). Call once at startup, and again whenever the AEC configuration changes.

During initialisation, **aec_init()** binds the AEC internal BFP structures to memory provided by the AEC memory pools. The amount of memory consumed depends on the runtime configuration.

Preconditions

The runtime configuration must be a subset of compile-time limits. This means:

- ▶ $\text{num_y_channels} \leq \text{AEC_MAX_Y_CHANNELS}$
- ▶ $\text{num_x_channels} \leq \text{AEC_MAX_X_CHANNELS}$
- ▶ Total phase-pool demand should not exceed pool capacity, i.e.:

$$(\text{num_y_channels} * \text{num_x_channels} * \text{num_main_filter_phases}) + (\text{num_x_channels} * \text{num_main_filter_phases}) \leq (\text{AEC_MAX_Y_CHANNELS} * \text{AEC_MAX_X_CHANNELS} * \text{AEC_MAIN_FILTER_PHASES}) + (\text{AEC_MAX_X_CHANNELS} * \text{AEC_MAIN_FILTER_PHASES})$$
- ▶ and $(\text{num_y_channels} * \text{num_x_channels} * \text{num_shadow_filter_phases}) \leq (\text{AEC_MAX_Y_CHANNELS} * \text{AEC_MAX_X_CHANNELS} * \text{AEC_SHADOW_FILTER_PHASES})$

Example

```
aec_state_t aec;
unsigned y_chans = 2, x_chans = 2;
unsigned main_phases = 10, shadow_phases = 5;

// Use a library-provided default schedule (2ch, 1 thread)
aec_init(&aec, y_chans, x_chans, main_phases, shadow_phases, &aec_tdist_chans2_threads1);

// Or, passing a custom schedule generated via setting 'AEC_SCHEDULE_CONFIG' in CMake
// extern aec_task_distribution_t tdist;
// aec_init(&aec, y_chans, x_chans, main_phases, shadow_phases, &tdist);
```

Note

The caller must provide memory pools sized for the maximum compile-time AEC configuration (**AEC_MAX_Y_CHANNELS**,



[AEC_MAX_X_CHANNELS](#), [AEC_MAIN_FILTER_PHASES](#), [AEC_SHADOW_FILTER_PHASES](#)). [aec_init\(\)](#) assigns memory from these pools based on the runtime configuration supplied. The memory pools must remain valid for the lifetime of the AEC instance. Any change to the number of channels or filter phases requires calling [aec_init\(\)](#) again to rebind internal state to the memory pools. See [aec_memory_pool_t](#) and [aec_shadow_filt_memory_pool_t](#) for details on memory pool sizing and usage.

Parameters

- ▶ **aec_state** – [inout] AEC state object
- ▶ **num_y_channels** – [in] Number of microphone 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
- ▶ **tdist** – [in] Work distribution to use for scheduling AEC work across hardware threads in [aec_process_frame\(\)](#). Use a library default for 2ch, 1 or 2 threads, otherwise provide an application-defined schedule.

```
void aec_process_frame(
    aec_state_t *aec_state, int32_t (*output_main)[AEC_FRAME_ADVANCE], int32_t
    (*output_shadow)[AEC_FRAME_ADVANCE], int32_t
    (*y_data)[AEC_FRAME_ADVANCE], int32_t (*x_data)[AEC_FRAME_ADVANCE],
)
```

Process a frame of microphone samples using the AEC.

This function performs acoustic echo cancellation on a frame of input microphone samples. It uses the input reference data frame to model the room echo characteristics and adapt the internal main and shadow filters.

Parameters

- ▶ **aec_state** – [inout] AEC state structure
- ▶ **output_main** – [inout] Output from processing the mic input through the main filter
- ▶ **output_shadow** – [inout] Output from processing the mic input through the shadow filter
- ▶ **y_data** – [in] Input microphone data frame
- ▶ **x_data** – [in] Input reference data frame

```
uint32_t aec_detect_input_activity(
    const int32_t (*input_data)[AEC_FRAME_ADVANCE], float_s32_t ac-
    tive_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

```
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_reset_state(aec_state_t *aec_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

- **aec_state** – [inout] AEC state structure

```
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.

```
float_s32_t aec_calc_corr_factor(aec_filter_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

AEC Defines

group **AEC #define constants**



Defines

AEC_MAIN_FILTER_PHASES

Maximum number of main-filter phases per adaptive filter at compile time.

This sets the compile-time capacity for the main adaptive filter and is used to size internal state and memory pools (see [aec_memory_pool_t](#)). It gets overridden by an autogenerated header ([aec_conf.h](#)) when using AEC custom schedule generation via CMake.

The `num_main_filter_phases` passed to [aec_init\(\)](#) must satisfy [aec_phase_pool_capacity](#).

AEC_SHADOW_FILTER_PHASES

Maximum number of shadow-filter phases per adaptive filter at compile time.

This sets the compile-time capacity for the shadow adaptive filter and is used to size internal state and memory pools (see [aec_shadow_filt_memory_pool_t](#)). It gets overridden by an autogenerated header ([aec_conf.h](#)) when using AEC custom schedule generation via CMake.

The `num_shadow_filter_phases` passed to [aec_init\(\)](#) must satisfy [aec_phase_pool_capacity](#).

AEC_MAX_Y_CHANNELS

Maximum number of microphone input channels at compile time. 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_MAX_Y_CHANNELS`, and satisfy the capacity rule in [aec_phase_pool_capacity](#).

This macro is used to size compile-time data structures (see [aec_state_t](#), [aec_memory_pool_t](#)). It gets overridden by an autogenerated header ([aec_conf.h](#)) when using AEC custom schedule generation via CMake. The implementation operates on the runtime value configured via [aec_init\(\)](#).

AEC_MAX_X_CHANNELS

Maximum number of reference (X) input channels at compile time.

Reference input is a copy of the device's loudspeaker output used by the AEC to model echo paths between loudspeakers and microphones.

AEC uses the convention x (time domain) and X (frequency domain) for reference signals.

The `num_x_channels` passed to [aec_init\(\)](#) must be \leq `AEC_MAX_X_CHANNELS` and satisfy the capacity rule in [aec_phase_pool_capacity](#).

This macro is used to size compile-time data structures (see [aec_state_t](#), [aec_memory_pool_t](#)). It gets overridden by an autogenerated header ([aec_conf.h](#)) when using AEC custom schedule generation via CMake. The implementation operates on the runtime value configured via [aec_init\(\)](#).

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, $H_{\hat{y}_{0x0}}$, $H_{\hat{y}_{0x1}}$, $H_{\hat{y}_{1x0}}$ and $H_{\hat{y}_{1x1}}$, 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_LIB_MAX_THREADS

Maximum number of hardware threads supported by the AEC.

This compile-time limit bounds the number of threads that a task distribution schedule can target (see [aec_task_distribution_t](#)). Custom schedules generated via CMake (AEC_SCHEDULE_CONFIG) must set <num_hw_threads> <= AEC_LIB_MAX_THREADS.

AEC Data Structures and Enums

group AEC Data Structure and Enum Definitions

Defines

REF_ACTIVE_THRESHOLD_DB

Reference input level above which it is considered active

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 **erle_thresh**
ERLE threshold

uq2_30 **erle_alpha_rise**
ERLE alpha when bigger

uq2_30 **erle_alpha_fall**
ERLE alpha when smaller

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

unsigned **mu_coh_time**
Number of frames after low coherence, adaption frozen for.

unsigned **mu_erle_time**
Number of frames after low erle, 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 **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 $2^{\text{gamma_log2}}$

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.

float_s32_t **erle**

Current ERLE.

float_s32_t **mov_erle**

Slow moving average ERLE.

int32_t **mu_coh_timer**

Timer for tracking number of frames adaption is frozen for.

int32_t **mu_shad_count**

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

float_s32_t **coh_mu**[[*AEC_MAX_X_CHANNELS*](#)]

Coherence mu.

struct **shadow_filter_params_t**

Public Members

int32_t **shadow_flag**[[*AEC_MAX_Y_CHANNELS*](#)]

shadow_state_e enum indicating shadow filter status

int **shadow_reset_count**[[*AEC_MAX_Y_CHANNELS*](#)]

counter for tracking shadow filter resets

int **shadow_better_count**[[*AEC_MAX_Y_CHANNELS*](#)]

counter for tracking shadow filter copy to main filter

struct **aec_shared_filter_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_filter_state_t::shared_state*](#) for both main and shadow filter point to the common [*aec_shared_t*](#) structure. [[*aec_shared_filter_state_t*](#)]



Public Members

bfp_complex_s32_t

X_fifo[[AEC_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_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_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_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_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_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_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_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`[\[AEC_MAX_Y_CHANNELS\]](#)

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`[\[AEC_MAX_X_CHANNELS\]](#)

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`[\[AEC_MAX_Y_CHANNELS\]](#)

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 overall_Yhat`[\[AEC_MAX_Y_CHANNELS\]](#)

Energy of the estimated 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`[\[AEC_MAX_X_CHANNELS\]](#)

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.

`int32_t ref_active_flag`

Reference active flag. Indicates if the reference signal is active or not for any x channel.

`coherence_mu_params_t coh_mu_state`[\[AEC_MAX_Y_CHANNELS\]](#)

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.

`unsigned X_energy_recalc_bin`



bin index for which the sum of X energy over all the X FIFO phases is re-calculated in the current frame. The index increments from 0 to AEC_PROC_FRAME_LENGTH/2, then decrements back to 0 over successive frames.

struct **aec_filter_state_t**

`#include <aec_state.h> [aec_shared_filter_state_t]`

AEC filter state structure.

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

Public Members

bfp_complex_s32_t **Y_hat**[AEC_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_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_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_haty0x0`, `H_hat[0][10]` to `H_hat[0][19]` points to 10 phases of `H_haty0x1` and `H_hat[0][20]` to `H_hat[0][29]` points to 10 phases of `H_haty0x2`. 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_1d[10]` to `X_fifo_1d[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_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**[\[AEC_MAX_X_CHANNELS\]](#)

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**[\[AEC_MAX_X_CHANNELS\]](#)

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**[\[AEC_MAX_Y_CHANNELS\]](#)

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**[\[AEC_MAX_Y_CHANNELS\]](#)

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**[\[AEC_MAX_Y_CHANNELS\]](#)

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**[\[AEC_MAX_Y_CHANNELS\]](#)[\[AEC_MAX_X_CHANNELS\]](#)

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

float_s32_t **error_ema_energy**[\[AEC_MAX_Y_CHANNELS\]](#)

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**[\[AEC_MAX_Y_CHANNELS\]](#)

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**[\[AEC_MAX_X_CHANNELS\]](#)

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_filter_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_filter_state_t](#) is instantiated for, passed in [aec_init\(\)](#) call.



struct **aec_state_t**

#include <aec_state.h> [[aec_filter_state_t](#)]

AEC state struct.

Data structure holding AEC module's persistent state.

Public Members

[aec_filter_state_t](#) **main_state**

AEC main filter state

[aec_filter_state_t](#) **shadow_state**

AEC shadow filter state

[aec_shared_filter_state_t](#) **shared_state**

AEC state shared between the main and shadow filter

[aec_memory_pool_t](#) **main_mem_pool**

Memory pool for the AEC main filter

[aec_shadow_filt_memory_pool_t](#) **shadow_mem_pool**

Memory pool for the AEC shadow filter

AEC Scheduling Data Structures

group **AEC Scheduling Types**

Defines

AEC_LIB_MAX_CHANNELS

Maximum channel count used by scheduling tables.

Computed as max([AEC_MAX_Y_CHANNELS](#), [AEC_MAX_X_CHANNELS](#)). This defines the upper bound for the channel dimension in the precomputed task-channel schedules within [aec_task_distribution_t](#).

Variables

const [aec_task_distribution_t](#) **aec_tdist_chans2_threads1**

Default schedule for running up to 2Y, 2X channels AEC on 1 hardware thread.

Usage:

► Pass &aec_tdist_chans2_threads1 to [aec_init\(\)](#) via the **tdist** argument.

const [aec_task_distribution_t](#) **aec_tdist_chans2_threads2**

Default schedule for running up to 2Y, 2X channels AEC on 2 hardware threads.

Usage:

► Pass &aec_tdist_chans2_threads2 to [aec_init\(\)](#) via the **tdist** argument.

struct **aec_par_tasks_t**

#include <aec_schedule.h> Data structures for distributing AEC work across hardware threads.

The task distribution scheme covers two scenarios:



- a. Distribute multiple unique tasks across multiple hardware threads. For example, for 3 tasks and 2 threads, distribute [task0, task1, task2] across [Thread0, Thread1].
- b. Distribute (task, channel) pairs across multiple hardware threads. For example, for 3 tasks, 2 channels, and 2 threads, distribute [(task0, ch0), (task0, ch1), (task1, ch0), (task1, ch1), (task2, ch0), (task2, ch1)] across [Thread0, Thread1].

The number of channels used when defining the (task, channel) pairs is fixed to `AEC_LIB_MAX_CHANNELS` (i.e., `max(AEC_MAX_Y_CHANNELS, AEC_MAX_X_CHANNELS)`).

Entry used when distributing tasks across hardware threads.

Public Members

int **task**

Task index.

int **is_active**

Flag indicating whether the task is active on that thread. The task runs on the thread only when **is_active** is 1.

struct **aec_par_tasks_and_channels_t**

#include <aec_schedule.h> Entry used when distributing (task, channel) pairs across hardware threads.

Public Members

int **task**

Task index.

int **channel**

Channel index.

int **is_active**

Flag indicating whether the (task, channel) pair is active on that thread. The pair runs on the thread only when **is_active** is 1.

struct **aec_task_distribution_t**

#include <aec_schedule.h> Precomputed schedules for mapping AEC work to hardware threads.

Provides lookup tables for several configurations:

- Distributing (task, channel) pairs across threads. For example, distributing 3 tasks, each running 2 channels, over 2 hardware threads
- Distributing unique tasks (no channel dimension) across threads. For example, distributing 3 tasks over 2 hardware threads

For each configuration, the corresponding **passes_*** value gives the number of passes required to execute all work items. The 2D arrays are indexed as [thread][work_item]. The second dimension (e.g., `3 * AEC_LIB_MAX_CHANNELS`) is sized (upper bound) for the worst case where a single thread performs all work items (one pass per work item). When multiple threads are available, the corresponding **passes_*** value indicates how many passes are actually required to cover all work items with the given **thread_count**.



Public Members

unsigned **thread_count**

Number of hardware threads this schedule targets (should be \leq AEC_LIB_MAX_THREADS)

unsigned **passes_for_3_tasks_and_channels**

Passes needed to cover all (3 tasks \times channels) work items.

aec_par_tasks_and_channels_t
par_3_tasks_and_channels[AEC_LIB_MAX_THREADS][3 *
 AEC_LIB_MAX_CHANNELS]

Schedule for 3 tasks, AEC_LIB_MAX_CHANNELS channels, scheduled across AEC_LIB_MAX_THREADS threads

unsigned **passes_for_2_tasks_and_channels**

Passes needed to cover all (3 tasks \times channels) work items.

aec_par_tasks_and_channels_t
par_2_tasks_and_channels[AEC_LIB_MAX_THREADS][2 *
 AEC_LIB_MAX_CHANNELS]

Schedule for 2 tasks, AEC_LIB_MAX_CHANNELS channels, scheduled across AEC_LIB_MAX_THREADS threads

unsigned **passes_for_1_tasks_and_channels**

Passes needed to cover all (1 task \times channels) work items.

aec_par_tasks_and_channels_t
par_1_tasks_and_channels[AEC_LIB_MAX_THREADS][1 *
 AEC_LIB_MAX_CHANNELS]

Schedule for 1 task, AEC_LIB_MAX_CHANNELS channels, scheduled across AEC_LIB_MAX_THREADS threads

unsigned **passes_for_3_tasks**

Passes needed to cover 3 unique tasks (no channel dimension).

aec_par_tasks_t **par_3_tasks**[AEC_LIB_MAX_THREADS][3]

Schedule for 3 unique tasks (no channel dimension), scheduled across AEC_LIB_MAX_THREADS threads

unsigned **passes_for_2_tasks**

Passes needed to cover 2 unique tasks (no channel dimension).

aec_par_tasks_t **par_2_tasks**[AEC_LIB_MAX_THREADS][2]

Schedule for 2 unique tasks (no channel dimension), scheduled across AEC_LIB_MAX_THREADS threads

AEC Memory Pool

group **AEC memory pool**



struct **aec_memory_pool_t**

`#include <aec_memory_pool.h>` [aec_memory_pool_t](#)

Memory pool for AEC main filter and shared state buffers.

This pool provides contiguous storage for all BFP (block floating-point from `lib_xcore_math`) structures used by the main filter ([aec_filter_state_t](#)) and the shared filter state ([aec_shared_filter_state_t](#)). The [aec_init\(\)](#) function initializes the BFP structures in the AEC state structures to point to memory buffers from this pool during AEC initialisation.

The memory pool allocates storage based on AEC compile-time configuration parameters:

- ▶ [AEC_MAX_Y_CHANNELS](#)
- ▶ [AEC_MAX_X_CHANNELS](#)
- ▶ [AEC_MAIN_FILTER_PHASES](#)
- ▶ [AEC_SHADOW_FILTER_PHASES](#)

The same pool can be used to initialize AEC for any runtime configuration (passed as arguments to [aec_init\(\)](#)) that is a subset of the compile-time configuration (See [aec_phase_pool_capacity](#)).

Note

This structure exists to own memory, not to describe layout. The memory pool acts as a linear allocation arena used by [aec_init\(\)](#) to initialise BFP structures in the AEC filter state structures. Memory is assigned sequentially from the pool based on the runtime configuration (number of channels and filter phases), and does not have a fixed or semantic mapping to the individual fields of this struct. The named members of this struct exist only to reserve sufficient contiguous storage at compile time. They must not be interpreted as backing specific components of [aec_filter_state_t](#)/[aec_shared_filter_state_t](#) and should never be accessed directly. After initialisation, all access to this memory occurs exclusively through the BFP structures owned by the AEC state.

Public Members

int32_t

mic_input_frame[\[AEC_MAX_Y_CHANNELS\]\[AEC_PROC_FRAME_LENGTH + AEC_FFT_PADDING\]](#)

Memory pointed to by [aec_shared_filter_state_t::y](#) and [aec_shared_filter_state_t::Y](#)

int32_t

ref_input_frame[\[AEC_MAX_X_CHANNELS\]\[AEC_PROC_FRAME_LENGTH + AEC_FFT_PADDING\]](#)

Memory pointed to by [aec_shared_filter_state_t::x](#) and [aec_shared_filter_state_t::X](#). Also reused for main filter [aec_filter_state_t::T](#)

int32_t

mic_prev_samples[\[AEC_MAX_Y_CHANNELS\]\[AEC_PROC_FRAME_LENGTH - AEC_FRAME_ADVANCE\]](#)

Memory pointed to by [aec_shared_filter_state_t::prev_y](#)



```
int32_t
ref_prev_samples[AEC_MAX_X_CHANNELS][AEC_PROC_FRAME_LENGTH
- AEC_FRAME_ADVANCE]
```

Memory pointed to by *aec_shared_filter_state_t::prev_x*

```
complex_s32_t phase_pool_H_hat_X_fifo[((AEC_MAX_Y_CHANNELS *
AEC_MAX_X_CHANNELS * AEC_MAIN_FILTER_PHASES) +
(AEC_MAX_X_CHANNELS * AEC_MAIN_FILTER_PHASES)) *
AEC_FD_FRAME_LENGTH]
```

Memory pointed to by main filter *aec_filter_state_t::H_hat*,
aec_shared_filter_state_t::X_fifo, main filter *aec_filter_state_t::X_fifo_1d*
and shadow filter *aec_filter_state_t::X_fifo_1d*

```
complex_s32_t Error[AEC_MAX_Y_CHANNELS][AEC_FD_FRAME_LENGTH]
```

Memory pointed to by main filter *aec_filter_state_t::Error* and
aec_filter_state_t::error

```
complex_s32_t Y_hat[AEC_MAX_Y_CHANNELS][AEC_FD_FRAME_LENGTH]
```

Memory pointed to by main filter *aec_filter_state_t::Y_hat* and
aec_filter_state_t::y_hat

```
int32_t X_energy[AEC_MAX_X_CHANNELS][AEC_FD_FRAME_LENGTH]
```

Memory pointed to by main filter *aec_filter_state_t::X_energy*

```
int32_t sigma_XX[AEC_MAX_X_CHANNELS][AEC_FD_FRAME_LENGTH]
```

Memory pointed to by *aec_shared_filter_state_t::sigma_XX*

```
int32_t inv_X_energy[AEC_MAX_X_CHANNELS][AEC_FD_FRAME_LENGTH]
```

Memory pointed to by main filter *aec_filter_state_t::inv_X_energy*

```
int32_t
overlap[AEC_MAX_Y_CHANNELS][AEC_UNUSED_TAPS_PER_PHASE * 2]
```

Memory pointed to by main filter *aec_filter_state_t::overlap*

```
struct aec_shadow_filt_memory_pool_t
```

```
#include <aec_memory_pool.h> aec_shadow_filt_memory_pool_t
```

Memory pool for AEC shadow filter.

This pool provides contiguous storage for all BFP (block floating-point from **lib_xcore_math**) structures used by the AEC shadow filter (*aec_filter_state_t*). The *aec_init()* function initializes the BFP structures in the AEC state structures to point to memory buffers from this pool during AEC initialisation.

The memory pool allocates storage based on AEC compile-time configuration parameters:

- ▶ *AEC_MAX_Y_CHANNELS*
- ▶ *AEC_MAX_X_CHANNELS*
- ▶ *AEC_MAIN_FILTER_PHASES*
- ▶ *AEC_SHADOW_FILTER_PHASES*

The same pool can be used to initialize AEC for any runtime configuration (passed as arguments to *aec_init()*) that is a subset of the compile-time configuration (See *aec_phase_pool_capacity*).



Note

This structure exists to own memory, not to describe layout. The memory pool acts as a linear allocation arena used by `aec_init()` to initialise BFP structures in the AEC filter state structures. Memory is assigned sequentially from the pool based on the runtime configuration (number of channels and filter phases), and does not have a fixed or semantic mapping to the individual fields of this struct. The named members of this struct exist only to reserve sufficient contiguous storage at compile time. They must not be interpreted as backing specific components of `aec_filter_state_t/aec_shared_filter_state_t` and should never be accessed directly. After initialisation, all access to this memory occurs exclusively through the BFP structures owned by the AEC state.

Public Members

complex_s32_t **phase_pool_H_hat**[`AEC_MAX_Y_CHANNELS` * `AEC_MAX_X_CHANNELS` * `AEC_SHADOW_FILTER_PHASES` * `AEC_FD_FRAME_LENGTH`]

Memory pointed to by shadow filter `aec_filter_state_t::H_hat`

complex_s32_t **Error**[`AEC_MAX_Y_CHANNELS`][`AEC_FD_FRAME_LENGTH`]

Memory pointed to by shadow filter `aec_filter_state_t::Error` and `aec_filter_state_t::error`

complex_s32_t **Y_hat**[`AEC_MAX_Y_CHANNELS`][`AEC_FD_FRAME_LENGTH`]

Memory pointed to by shadow filter `aec_filter_state_t::Y_hat` and `aec_filter_state_t::y_hat`

complex_s32_t **T**[`AEC_MAX_X_CHANNELS`][`AEC_FD_FRAME_LENGTH`]

Memory pointed to by shadow filter `aec_filter_state_t::T`

int32_t **X_energy**[`AEC_MAX_X_CHANNELS`][`AEC_FD_FRAME_LENGTH`]

Memory pointed to by shadow filter `aec_filter_state_t::X_energy`

int32_t **inv_X_energy**[`AEC_MAX_X_CHANNELS`][`AEC_FD_FRAME_LENGTH`]

Memory pointed to by shadow filter `aec_filter_state_t::inv_X_energy`

int32_t **overlap**[`AEC_MAX_Y_CHANNELS`][`AEC_UNUSED_TAPS_PER_PHASE` * 2]

Memory pointed to by shadow filter `aec_filter_state_t::overlap`

4.2 IC**IC API Functions***group* **High Level API Functions****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 be 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_process_frame(
    ic_state_t *state, int32_t y_data[IC_FRAME_ADVANCE], int32_t
    x_data[IC_FRAME_ADVANCE], int32_t output[IC_FRAME_ADVANCE], float_s32_t
    *input_vnr_pred,
)
```

Filter one frame of audio data inside the IC, calculate VNR prediction, adapt the IC.

Runs *ic_filter*, *ic_calc_vnr_pred* and *ic_adapt*.

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
- **input_vnr_pred** – [out] voice to noise estimate of the IC input

```
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)
```

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** – **[out]** voice to noise estimate of the IC input

void **ic_adapt**(*ic_state_t* *state)

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

IC Defines

group **IC #define constants**

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_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.

IC Data Structures and Enums

group IC Data Structures

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 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

4.3 VNR

VNR API Functions

group VNR API functions

Functions

void **vnr_state_init**(*vnr_state_t* *vnr)

Initialise the VNR state.

This function should be called once at device startup.

Parameters

- ▶ **vnr** – [inout] pointer to the VNR state structure

void **vnr_process_frame**(
vnr_state_t *vnr, float_s32_t *output, int32_t input[*VNR_FRAME_ADVANCE*],
)

Calculate the Voice to Noise Ratio estimation from a frame of input data.

This function takes a frame of new samples, converts them to features and passes those to the inference engine. 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** – [inout] VNR state structure
- ▶ **output** – [out] Pointer to return the resulting ratio
- ▶ **input** – [in] Array of frame data on which to perform the VNR

group VNR feature extraction API functions



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.

group VNR inference API functions

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.

VNR Defines

group

VNR #define constants common to both feature extraction and inference



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.

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.

VNR Data structures and Enums

group VNR data structure definitions

struct **vnr_input_state_t**

#include <vnr_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_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_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.

struct **vnr_state_t**

#include <vnr_state.h> State structure used for the VNR.

Public Members

vnr_input_state_t **input_state**

VNR Input state

vnr_feature_state_t **feature_state**

VNR Feature state

4.4 NS

NS API Functions

group **NS API functions**

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 Defines and Structure Definitions

group NS API structure definitions

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_overlap**[[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_alpha_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, specified in samples.

unsigned **reset_counter**

Filter reset counter.

4.5 AGC

AGC API Functions

group **AGC API functions**

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



`agc_meta_data_t agc_meta_data_init()`

Initialise AGC meta data structure.

Set the AGC meta data values to indicate no VNR, no AEC and no reference signal.

`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_t md = agc_meta_data_init();
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 Defines

Three pre-defined profiles are provided in *agc_profiles.h* to configure the AGC for different applications:

group **Pre-defined AGC configuration 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_PROFILE_XCV_COMMS

AGC profile tuned for human to human communications.

These profiles can be used to configure the AGC instance by passing them to the *agc_init()* function.

AGC Data Structures and Enums*group* **AGC API structure definitions**

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.

AGC_META_DATA_NO_REF

If the application has no reference signal, this pre-processor definition can be assigned to the **ref_active_flag** in [agc_meta_data_t](#) to make it clear in the code that there is no reference signal.

struct [agc_config_t](#)

#include <agc.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.

float_s32_t [vnr_threshold](#)

VNR threshold for voice activity detection. A higher value will only adapt the AGC on clean speech. A lower value will adapt the AGC on noisy speech, but may also adapt to more non-speech signals.

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. When **adapt** is false, this gain will be applied to every frame. When **adapt** is true, the initial value of this gain will be applied to the first frame and then it will be adapted on subsequent frames.



float_s32_t **max_gain**

The maximum gain allowed when adaption is enabled. This can be used to prevent the AGC amplifying very quiet signals.

float_s32_t **min_gain**

The minimum gain allowed when adaption is enabled.

float_s32_t **upper_threshold**

The target maximum peak level of the AGC output. If the AGC output goes above this level, the gain is reduced.

float_s32_t **lower_threshold**

The target minimum peak level of the AGC output. If the AGC output goes below this level, the gain is increased.

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.

uint32_t **startup_delay**

Number of frames to mute the output at startup.

int **lc_enabled**

Boolean to enable loss control. The loss control applies additional attenuation when there is no near end speech. This must be disabled if the application doesn't have an AEC or VNR.

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. How many times louder the near-end signal must be than the background noise when there is no far-end playback. If the near end speech is not heard during silence, reduce this value. If too much non-speech background noise is heard, increase this value.

float_s32_t `lc_near_delta_far_active`

Delta multiplier used when both near-end and far-end activity is detected. How many times louder the near end signal must be above the residual far-end speech (after the AEC) to be detected during double talk. If the near end speech is not heard during double talk, reduce this value. If there is too much breakthrough of residual far-end echo when there is no near-end speech present, increase this value.

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. Reducing this value will reduce the level of the near-end speech during double-talk, but may help to reduce the level of residual far-end echo that is heard.

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.

float_s32_t `lc_vnr_low`

Low VNR threshold for background estimation.

int `lc_vnr_low_count_limit`

Frame count limit for low VNR detection.

struct `agc_state_t`

#include <agc.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.

int **lc_vnr_low_count**

Counter of how many frames the VNR has been low for.

uint32_t **frame_count**

Frame counter since initialisation, used for startup delay

struct **agc_meta_data_t**

#include <agc.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

float_s32_t **vnr_flag**

Estimated voice-to-noise ratio 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.

int32_t **ref_active_flag**

Flag to indicate if the reference signal is currently active

4.6 ADEC

ADEC API Functions

group ADEC API Functions

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 Defines

group **ADEC #define constants**

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 USER MODIFIABLE.

ADEC_DE_DELAY_MS

Initial delay of microphone in the DE mode in milliseconds. This allows measuring up to **ADEC_DE_DELAY_MS** ms of delay in cases when the mic is earlier than the reference.

ADEC_DE_DELAY_SAMPS

Same as **ADEC_DE_DELAY_MS** but in samples.

ADEC Data Structures and Enums

group **ADEC Data Structure and Enum Definitions**

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. Initialising ADEC with this flag set to 1 can be used to get an initial delay estimate at boot.

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.

4.7 Stage1

Stage1 API Functions

group Stage1 API

Functions

```
void stage1_init(
    stage1_t *state, adec_config_t *de_conf, adec_config_t
    *non_de_conf, adec_config_t *adec_config,
)
```

Initialise Stage1 processing.

Sets up persistent state for Stage1, initialises ADEC, AEC (in non delay estimation mode) and the delay buffer. Also resets internal counters used to control AEC enable/disable behaviour in alt-arch mode.

All pointers must be non-NULL. The `stage1_t` memory must persist for the lifetime of the stage 1 processing.

Parameters

- ▶ **state** – [inout] Stage1 state to initialise.
- ▶ **de_conf** – [in] AEC runtime configuration used when the delay estimator path is enabled.



- ▶ **non_de_conf** – [in] AEC runtime configuration running in non delay estimation mode
- ▶ **adec_config** – [in] ADEC configuration

```
void stage1_process_frame(
    stage1_t *state, int32_t (*output_frame)[AEC_FRAME_ADVANCE], float_s32_t
    *max_ref_energy, float_s32_t *aec_corr_factor, int32_t
    *ref_active_flag, int32_t (*input_y)[AEC_FRAME_ADVANCE], int32_t (*input_x)[AEC_FRAME_ADVANCE],
)
```

Performs stage1 processing on a frame of input data.

This function delays the input by the estimated delay, performs AEC processing on the input frame, updates metadata propagated to downstream pipeline stages, runs ADEC and applies ADEC result (e.g. switch AEC config, change applied delay).

Supports standard or alternating-architecture mode controlled by the compile-time flag **ALT_ARCH_MODE**.

Parameters

- ▶ **state** – [inout] Persistent Stage1 state.
- ▶ **output_frame** – [out] Output frame buffer [Y channels][AEC_FRAME_ADVANCE] in Q31 format.
- ▶ **max_ref_energy** – [out] Maximum reference-channel energy (float_s32_t) for this frame.
- ▶ **aec_corr_factor** – [out] AEC correction factor (float_s32_t) computed for this frame.
- ▶ **ref_active_flag** – [out] Set non-zero if reference is detected active this frame.
- ▶ **input_y** – [in] Microphone (Y) input frame [Y channels][AEC_FRAME_ADVANCE] in Q31 format.
- ▶ **input_x** – [in] Reference (X) input frame [X channels][AEC_FRAME_ADVANCE] in Q31 format.

Stage1 defines and data structures

group Stage1 types

Defines

ALT_ARCH_MODE

Enable stage1 alternative arch mode

HOLD_AEC_LIMIT_SECONDS

Limit in seconds for which AEC is kept enabled after detecting reference as inactive. Used only in alt arch configuration.

struct **aec_conf_t**

#include <stage1.h> AEC runtime configuration structure.

Public Members

uint8_t **num_x_channels**

< Number of reference (X) input channels at runtime Number of microphone (Y) input channels at runtime



uint8_t **num_main_filt_phases**

Runtime main-filter phase count

uint8_t **num_shadow_filt_phases**

Runtime shadow-filter phase count

const [aec_task_distribution_t](#) ***tdist**

Pointer to the work distribution schedule to use (e.g., [aec_tdist_chans2_threads1](#))

struct **stage1_t**

#include <stage1.h> Persistent state for stage1 processing.

It aggregates AEC, ADEC and delay buffer state, AEC runtime configurations for delay and non-delay estimation mode and control flags used in stage1 processing.

Public Members

[aec_state_t](#) **aec_state**

AEC state

[adec_state_t](#) **adec_state**

ADEC state

[delay_buf_state_t](#) **delay_state**

Delay buffer state

[aec_conf_t](#) **aec_de_mode_conf**

AEC config in delay estimation mode

[aec_conf_t](#) **aec_non_de_mode_conf**

AEC config in non delay estimation (regular) mode

int32_t **delay_estimator_enabled**

Flag indicating if AEC is running in delay estimation mode

float_s32_t **ref_active_threshold**

Threshold used for detecting activity on the reference audio channel

int32_t **hold_aec_count**

Number of consecutive frames reference has been inactive for. Used only in alt-arch mode

int32_t **hold_aec_limit**

Number of frames the reference must be inactive before AEC is disabled. Used only in alt-arch mode





Copyright © 2026, All Rights Reserved.

XMOS Ltd. is the owner or licensee of this design, code, or Information (collectively, the "Information") and is providing it to you "AS IS" with no warranty of any kind, express or implied and shall have no liability in relation to its use. XMOS Ltd makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.

XMOS, XCORE, VocalFusion and the XMOS logo are registered trademarks of XMOS Ltd. in the United Kingdom and other countries and may not be used without written permission. Company and product names mentioned in this document are the trademarks or registered trademarks of their respective owners.

