



lib_mic_array: PDM microphone array library

Publication Date: 2026/2/5

Document Number: XM-010267-UG v6.0.0

IN THIS DOCUMENT

1	Introduction	2
2	Overview	3
2.1	Capabilities	3
2.2	High level process view	4
3	Using lib_mic_array	6
3.1	Including in an application	6
3.2	Default usage model	6
4	Examples	11
4.1	Running the examples	11
5	Resource usage	13
5.1	Discrete Resources	14
5.2	Compute	15
5.3	Memory	16
6	Software structure	17
6.1	High level view	17
6.2	Sub-Components	19
7	Decimation filters	20
7.1	Filters provided as part of lib_mic_array	20
7.2	Decimator stage 1	23
7.3	Decimator stage 2	25
8	Custom decimation filters	25
8.1	Designing a custom filter	25
8.2	Using custom filters	26
9	Advanced usage	28
9.1	Declare resources	28
9.2	Construct MicArray object	28
9.3	Define app-callable functions	29
9.4	Application main function	29
10	Sample filters	30
10.1	DC Offset elimination	30
11	API Reference	31
11.1	C++ API Reference	31
11.2	C API Reference	47

1 Introduction

lib_mic_array provides microphone array processing support on X MOS devices. It interfaces with one or more PDM (Pulse Density Modulation) microphones, capturing and converting their output into high-quality PCM audio suitable for downstream voice and audio processing pipelines.

PDM microphones produce a high-rate, 1-bit digital bitstream. The library captures these PDM streams on the device and performs the required filtering and decimation to produce 32-bit PCM audio samples.

For high level description of mic array processing and the library capabilities, refer to [Overview](#). To get started with using the library, see [Using lib_mic_array](#) and [Examples](#).

Note

From Version 5.0 onwards, the library does not support XS2 or XS1 devices. Please use version 4.5.0 if you need support for these devices: https://github.com/xmos/lib_mic_array/releases/tag/v4.5.0



2 Overview

lib_mic_array is a library for capturing and processing PDM microphone data on xcore.ai devices.

PDM microphones are a kind of ‘digital microphone’ which captures audio data as a stream of 1-bit samples at a very high sample rate. The high sample rate PDM stream is captured by the device, filtered and decimated to a 32-bit PCM audio stream.

2.1 Capabilities

- ▶ Both SDR (1 mic per pin) and DDR (2 mics per pin) microphone configurations are supported
- ▶ Configurable clock divider allows user-selectable PDM sample clock frequency (3.072 MHz typical)
- ▶ Configurable *two-stage decimating FIR filter*
 - ▶ First stage has fixed tap count of 256 and decimation factor of 32
 - ▶ Second stage has fully configurable tap count and decimation factor
 - ▶ Custom filter coefficients can be used for either stage
 - ▶ Pre-designed reference filters with total decimation factor of 192, 96 and 64 are provided (16 kHz, 32 kHz and 48 kHz output sample rates with 3.072 MHz input PDM clock).
 - ▶ Filter generation scripts and examples are included to support custom filter design.
- ▶ Supports 1-, 4- and 8-bit ports.
- ▶ Supports 1 to 16 microphones
 - ▶ Includes ability to capture samples on a subset of a port’s pins (e.g. 3 PDM microphones may be used with a 4- or 8-bit port)
 - ▶ Also supports microphone channel index remapping
 - ▶ See *Compute* for the mic array MIPS requirement.
- ▶ Optional *DC offset elimination filter*
- ▶ Sample framing with user selectable frame size (down to single samples)



2.2 High level process view

This section gives a brief overview of the steps to process a PDM audio stream into a PCM audio stream. This section is concerned with the steady state behaviour and does not describe any necessary initialization steps. The high level process view is depicted in the figure [Mic Array High Level Process](#).

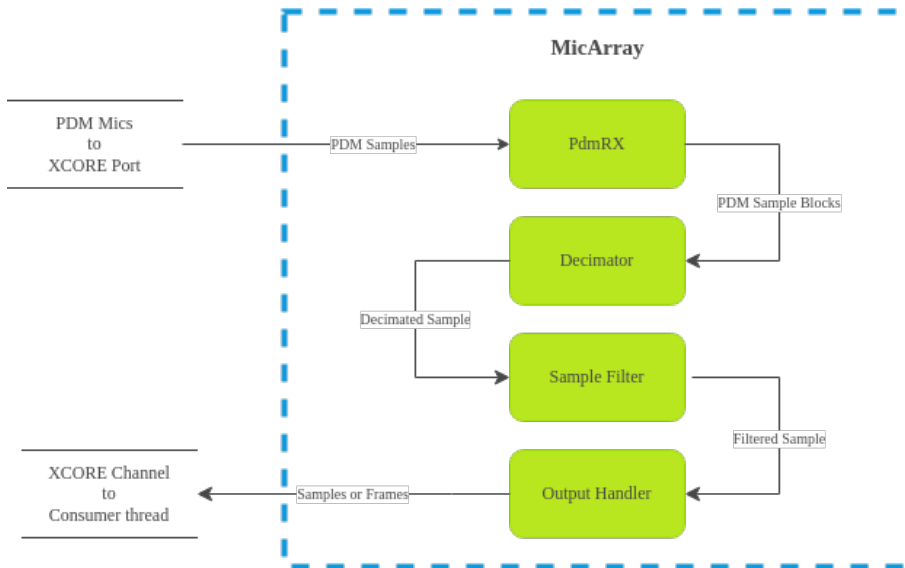


Fig. 1: Mic Array High Level Process

Execution contexts

The mic array unit uses two different execution contexts. The first is the PDM RX service ("PDM RX"), which is responsible for reading PDM samples from the physical port, and has relatively little work to do, but also has a strict real-time constraint on reading port data in a timely manner. The second is the decimation thread, which is where all processing other than PDM capture is performed.

This two-context model relaxes the need for tight coupling and synchronization between PDM RX and the decimation thread, allowing significant flexibility in how samples are processed in the decimation thread.

PDM RX is typically run within an interrupt on the same hardware core as the decimation thread, but it can also be run as a separate thread in cases where many channels result in a high processing load.

Likewise, the decimators may be split into multiple parallel hardware threads in the case where the processing load exceeds the MIPS available in a single hardware thread.

Step 1: PDM capture

The PDM data signal is captured by the xcore.ai device's port hardware. The port receiving the PDM signals buffers the received samples. Each time the port buffer is filled, PDM RX reads the received samples.



Samples are collected word-by-word and assembled into blocks. Each time a block has been filled, the block is transferred to the decimation thread where all remaining mic array processing takes place.

The size of PDM data blocks varies depending upon the configured number of microphone channels and the configured second stage decimator's decimation factor. Each PDM data block will contain exactly enough PDM samples to produce one new mic array (multi-channel) output sample.

Step 2: First stage decimation

The conversion from the high-sample-rate PDM stream to lower-sample-rate PCM stream involves two stages of decimating filters. After the decimation thread receives a block of PDM samples, the samples are filtered by the first stage decimator.

The first stage decimator has a fixed decimation factor of **32** and a fixed tap count of **256**. An application is free to supply its own filter coefficients for the first stage decimator (using the fixed decimation factor and tap count), however this library also provides a reference filter for the first stage decimator that is recommended for most applications.

The first stage decimating filter is an FIR filter with 16-bit coefficients, and where each input sample corresponds to a **+1** or a **-1** (typical for PDM signals). The output of the first stage decimator is a block of 32-bit PCM samples with a sample time **32** times longer than the PDM sample time.

See [Decimation filters](#) for further details.

Step 3: Second stage decimation

The second stage decimator is a decimating FIR filter with a configurable decimation factor and tap count. Like the first stage decimator, this library provides a reference filter suitable for the second stage decimator. The supplied filter has a tap count of **64** and a decimation factor of **6**.

The output of the first stage decimator is a block of **N*K** PCM values, where **N** is the number of microphones and **K** is the second stage decimation factor. This is just enough samples to produce one output sample from the second stage decimator.

The resulting sample is vector-valued (one element per channel) and has a sample time corresponding to **32*K** PDM clock periods. Using the reference filters and a 3.072 MHz PDM clock, this corresponds to an output sampling rate of 16 kHz, 32 kHz, or 48 kHz.

See [Decimation filters](#) for further details.

Step 4: post-processing

After second stage decimation, the resulting sample goes to post-processing where two (optional) post-processing steps are available.

The first is a simple IIR filter, called DC Offset Elimination, which seeks to ensure each output channel tends to approach zero mean. DC Offset Elimination can be disabled if not desired. See [Sample filters](#) for further details.

The second post-processing step is framing, where instead of signalling each sample of audio to subsequent processing stages one at a time, samples can be aggregated and transferred to subsequent processing stages as non-overlapping blocks. The size of each frame is configurable (down to 1 sample per frame, where framing is functionally disabled).



Finally, the sample or frame is transmitted over a XCORE channel from the mic array module to the next stage of the processing pipeline.

Extending/Modifying mic array behaviour

Most applications are expected to use the [default](#) usage model, which provides a complete, ready-to-integrate mic-array pipeline and should be sufficient for the majority of use cases. Only applications with specialised requirements such as non-standard data-transfer mechanisms, custom decimation behaviour etc. are likely to require customisation beyond the default model.

At the core of **lib_mic_array** are several C++ class templates which are loosely coupled and intended to be easily overridden for modified behaviour. The mic array unit itself is an object made by the composition of several smaller components which perform well-defined roles (See [Software structure](#)).

For example, modifying the mic array unit to use some mechanism other than a channel to move the audio frames out of the mic array is a matter of defining a small new class encapsulating just the modified transfer behaviour, and then instantiating the mic array class template with the new class as the appropriate template parameter.

3 Using lib_mic_array

3.1 Including in an application

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

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

```
set(APP_DEPENDENT_MODULES "lib_mic_array")
```

Applications should then include the `mic_array.h` header file.

3.2 Default usage model

The default model described in this section is the recommended approach for integrating the mic array into an application and is suitable for most use cases. Its constraints should be reviewed to determine whether it is compatible with the requirements of the target application (see [Limitations of the default model](#)).

Typical sequence of steps required to create and use a mic-array instance in an application:

- ▶ [Identify hardware resources](#)
- ▶ [Declare a struct describing the required hardware resources](#)
- ▶ [Initialise and start the mic array task](#)
- ▶ [Receive audio frames in the receiver thread](#)
- ▶ [If required, shutdown the mic array task and optionally re-init and restart.](#)

The following sections describe the above steps in detail, followed by a [code example demonstrating the mic array default usage model](#).



Identify hardware resources

The key hardware resources to be identified are the *ports* and *clock blocks* that will be used by the mic array unit. The ports correspond to the physical pins on which clocks and sample data will be signaled. Clock blocks are a type of hardware resource which can be attached to ports to coordinate the presentation and capture of signals on physical pins.

Clock blocks While clock blocks may be more abstract than ports, their implications for this library are actually simpler. First, the mic array unit will need a way of taking the audio master clock and dividing it to produce a PDM sample clock. This can be accomplished with a clock block. This will be the clock block which the API documentation refers to as “Clock A”.

Second, if (and only if) the PDM microphones are being used in a Dual Data Rate (DDR) configuration a second clock block will be required. In a DDR configuration 2 microphones share a physical pin for output sample data, where one signals on the rising edge of the PDM clock and the other signals on the falling edge. The second clock block required in a DDR configuration is referred to as “Clock B” in the API documentation.

Each tile on an xcore.ai device has 5 clock blocks available. In code, a clock block is identified by its resource ID, which are given as the preprocessor macros `XS1_CLKBLK_1` through `XS1_CLKBLK_5`.

Unlike ports, which are tied to specific physical pins, clock blocks are fungible. The application is free to use any clock block that has not already been allocated for another purpose.

Ports Three ports are needed for the mic array component. As mentioned above, ports are physically tied to specific device pins, and so the correct ports must be identified for correct behaviour.

Note that while ports are physically tied to specific pins, this is *not* a 1-to-1 mapping. Each port has a port width (measured in bits) which is the number of pins which comprise the port. Further, the pin mappings for different ports *overlap*, with a single pin potentially belonging to multiple ports. When identifying the needed ports, take care that both the pin map (see the documentation for your xcore.ai package) and port width are correct.

The first port needed is a 1-bit port on which the audio master clock is received. In the documentation, this is usually referred to as `p_mclk`.

The second port needed is a 1-bit port on which the PDM clock will be signaled to the PDM mics. This port is referred to as `p_pdm_clk`.

The third port is that on which the PDM data is received. In an SDR configuration, the width of this port must be greater than or equal to the number of microphones. In a DDR configuration, twice this port width must be greater than or equal to the number of microphones. This port is referred to as `p_pdm_mics`.

XCORE applications are typically compiled with an “XN” file (with a “.xn” file extension). An XN file is an XML document which describes some information about the device package as well as some other helpful board-related information. The identification of ports may have already been done in the XN file. Following is a snippet from an XN file with mappings for the three ports described above:

```
...
<Tile Number="1" Reference="tile[1]">
  <!-- MIC related ports -->
  <Port Location="XS1_PORT_1G" Name="PORT_PDM_CLK"/>
  <Port Location="XS1_PORT_1F" Name="PORT_PDM_DATA"/>
  <!-- Audio ports -->
  <Port Location="XS1_PORT_1D" Name="PORT_MCLK_IN_OUT"/>
</Tile>
```

(continues on next page)



(continued from previous page)

```

<Port Location="XS1_PORT_1C" Name="PORT_I2S_BCLK"/>
<Port Location="XS1_PORT_1B" Name="PORT_I2S_LRCLK"/>
<!-- Used for looping back clocks -->
<Port Location="XS1_PORT_1N" Name="PORT_NOT_IN_PACKAGE_1"/>
</File>
...

```

The first 3 ports listed, `PORT_PDM_CLK`, `PORT_PDM_DATA` and `PORT_MCLK_IN_OUT` are respectively `p_pdm_clk`, `p_pdm_mics` and `p_mclk`. The value in the `Location` attribute (e.g. `XS1_PORT_1G`) is the port name as you will find it in the package documentation.

In this case, either `PORT_PDM_CLK` or `XS1_PORT_1G` can be used in code to identify this port.

Other Resources In addition to ports and clock blocks, there are also several other hardware resource types used by `lib_mic_array` which are worth considering. Running out of any of these will preclude the mic array from running correctly (if at all)

- ▶ **Threads** - At least one hardware thread is required to run the mic array component. If PDM RX service is not running in the interrupt context (`MIC_ARRAY_CONFIG_USE_PDM_ISR = 0`), a separate thread is required to run it.
- ▶ **Compute** - The mic array unit will require a fixed number of MIPS (millions of instructions per second) to perform the required processing. The exact requirement will depend on the configuration used.
- ▶ **Memory** - The mic array requires a modest amount of memory for code and data. (see [Resource usage](#)).
- ▶ **Chanends** - At least 4 chanends must be available for signalling between threads/sub-components.

Declare hardware resources

Once the ports and clock blocks to be used have been identified, these resources can be represented in code using a `pdm_rx_resources_t` struct. The following is an example of declaring resources in a DDR configuration. See [pdm_rx_resources_t](#), [PDM_RX_RESOURCES_SDR\(\)](#) and [PDM_RX_RESOURCES_DDR\(\)](#) for more details.

```

pdm_rx_resources_t pdm_res = PDM_RX_RESOURCES_DDR(
    PORT_MCLK_IN,
    PORT_PDM_CLK,
    PORT_PDM_DATA,
    24576000,
    3072000,
    XS1_CLKBLK_1,
    XS1_CLKBLK_2);

```

Initialise and start the mic array

Once the resources are identified and the `pdm_rx_resources_t` struct is populated, the application needs to call the mic array functions to initialise and start the mic array task.

Call [mic_array_init\(\)](#) to initialise the mic array component. `mic_array_init()` accepts arguments for configuring the mic array component at run-time.

Additional configuration parameters defined in `mic_array_conf_default.h` specify the default compile-time configuration of the mic array component. These can be overridden by the application in a `mic_array_conf.h` file included by the ap-



plication or through the application's CMakeLists.txt. See the [Configuration defines \(mic_array_conf_default.h\)](#) section for details.

This is followed by a call to [mic_array_start\(\)](#) which starts the mic array thread(s). [mic_array_start\(\)](#) takes a single argument - the XCORE chanend used to transmit audio frames to subsequent stages of the processing pipeline. Typically, the call to [mic_array_start\(\)](#) will be placed directly in a `par { ... }` block along with other threads to be started on the same tile. [mic_array_start\(\)](#) is a blocking call that runs the mic array thread(s) until the application signals shutdown using [ma_shutdown\(\)](#).

Receive PCM frames

The application receives output PCM frames from the mic array task over an XCORE chanend. This is the other end of the XCORE chanend passed to [mic_array_start\(\)](#). A call to [ma_frame_rx\(\)](#) with the chanend as an argument is typically called from another thread to receive PCM blocks from the mic array for further processing.

Shutdown

The application can shut down the mic array task by calling [ma_shutdown\(\)](#). The shutdown request is sent over the same channel end that is used by [ma_frame_rx\(\)](#). Therefore, the application must ensure that [ma_frame_rx\(\)](#) is not being called concurrently when invoking [ma_shutdown\(\)](#).

Calling [ma_shutdown\(\)](#) causes the mic-array threads to terminate ([mic_array_start\(\)](#) returns). [ma_shutdown\(\)](#) itself returns only after all mic-array threads have fully terminated. Once shut down, [mic_array_init\(\)](#) and [mic_array_start\(\)](#) can be called again to restart the mic array.

Note

Shutting down and restarting the mic array is the supported method for changing the output sample rate. The sample rate cannot be modified while the mic array is running; instead, call [ma_shutdown\(\)](#), reconfigure the desired rate, and then restart the mic array.

Example code

The code block below shows the application `main()` function containing calls to the mic array functions:

```
#include <platform.h>
#include <xs1.h>
#include "mic_array_task.h"

on tile[PORT_PDM_CLK_TILE_NUM] : port p_mclk = PORT_MCLK_IN_OUT;
on tile[PORT_PDM_CLK_TILE_NUM] : port p_pdm_clk = PORT_PDM_CLK;
on tile[PORT_PDM_CLK_TILE_NUM] : port p_pdm_data = PORT_PDM_DATA;
on tile[PORT_PDM_CLK_TILE_NUM] : clock clk_a = XS1_CLKBLK_1;
on tile[PORT_PDM_CLK_TILE_NUM] : clock clk_b = XS1_CLKBLK_2;

unsafe{
void receive_from_mics(chanend c_pcm)
{
    int32_t audio_frame[MIC_ARRAY_CONFIG_MIC_COUNT * 1]; // frame = 1 sample for each of the MIC_ARRAY_CONFIG_MIC_
    ←COUNT channels
    while(1)
    {
        ma_frame_rx(audio_frame, (chanend_t)c_pcm, MIC_ARRAY_CONFIG_MIC_COUNT, 1);
        // further processing of the pcm data...
    }
}

int main() {
    chan c_audio_frames;
```

(continues on next page)



(continued from previous page)

```

par {
  on tile[1]: {
    pdm_rx_resources_t pdm_res = PDM_RX_RESOURCES_DDR(p_mclclk, p_pdm_clk, p_pdm_data, 24576000, 3072000, clk_a,
    ← clk_b);

    mic_array_init(&pdm_res, null, 48000);

    par {
      mic_array_start((chanend_t) c_audio_frames);
      receive_from_mics(c_audio_frames);
    }
  }
}
return 0;
}

```

Note

Code above does not demonstrate mic array shutdown. See the [mic array shutdown example](#) for usage of `ma_shutdown()`.

Limitations of the default model

The default model for using the mic array, while easiest to integrate in an application, has the following constraints:

- Fixed supported output sampling rates:

Only **16 kHz**, **32 kHz**, and **48 kHz** output sampling rates are supported. This is because the default decimation filters provided as part of the library (see [Filters provided as part of lib_mic_array](#)) are designed for a small set of decimation factors and they assume a fixed input PDM frequency of **3.072 MHz**. See [Custom decimation filters](#) and [app_custom_filter](#) for using custom filters for the mic array (provided they are compatible with the [TwoStageDecimator](#) implementation)

- Only one mic array instance:

The default model does not support multiple independent, mic array instances

- Single decimator thread:

In the default model, the entire decimation process runs within a single hardware thread. For higher microphone counts, this may exceed the available compute capacity of one thread (see [mic array MIPS requirement](#)). In general, up to four microphones can be accommodated within a single thread, depending on the configuration. Higher channel counts therefore require splitting the decimation process across multiple hardware threads and using lib_mic_array in a [customised](#) configuration. The [app_par_decimator](#) example demonstrates a two-threaded decimator implementation.

- Memory usage:

The default API incurs an additional memory overhead compared to a custom instantiation of [MicArray](#) object. This overhead is primarily from wrapper code and inclusion of all provided filter coefficient sets, even when only a subset is used (see [Memory usage \(in bytes\)](#)). For memory-constrained systems a custom configuration might be preferable.

For custom usage involving creating a [MicArray](#) object with potentially custom implementations of the component classes, refer to the advanced use methods described in [Advanced usage](#)



4 Examples

Example applications are included with **lib_mic_array** to illustrate both the default usage model and advanced/custom integration approaches.

These examples are located in the **examples** directory of the library. All examples run on the [XK-VOICE-L71](#) hardware platform.

Examples overview:

- ▶ **app_mic_array** – demonstrates the *default* way of using the mic array in an application (recommended starting point).
- ▶ **app_custom_filter** - demonstrates using a *custom decimation filter* in the mic array processing
- ▶ **app_shutdown** – demonstrates *shutting down and restarting* the mic array.
- ▶ **app_par_decimator** – demonstrates an *advanced* integration approach. Replaces the standard decimator component with a custom two-threaded implementation.

4.1 Running the examples

This section describes how to build and run the example applications provided with the **lib_mic_array** library. The application used here is **app_mic_array** which demonstrates use of the default mic array API. For other examples, the process is similar, except for the application/folder name.

Required hardware

- ▶ 1x **XK-VOICE-L71** board
- ▶ 2x Micro USB cable (For USB power and XTAG)
- ▶ 3.5mm audio cable to connect to the line out connector on the **XK-VOICE-L71** board

Setup procedure

1. Connect the XTAG to the **XK-VOICE-L71**
2. Connect one Micro USB cable between the host computer and the **XK-VOICE-L71 USB** port.
3. Connect the other Micro USB cable between the host computer and the XTAG.

[XK-VOICE-L71 setup](#) shows the **XK-VOICE-L71** board with all the cables connected.

Building

The following instructions assumes that the [XMOSES XTC tools](#) has been downloaded and installed (see *README* for required version).

Installation instructions can be found [here](#). Particular attention should be paid to the section [Installation of required third-party tools](#).

The application uses the *XMOSES* build and dependency system, [xcommon-cmake](#). *xcommon-cmake* is bundled with the *XMOSES* XTC tools.

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



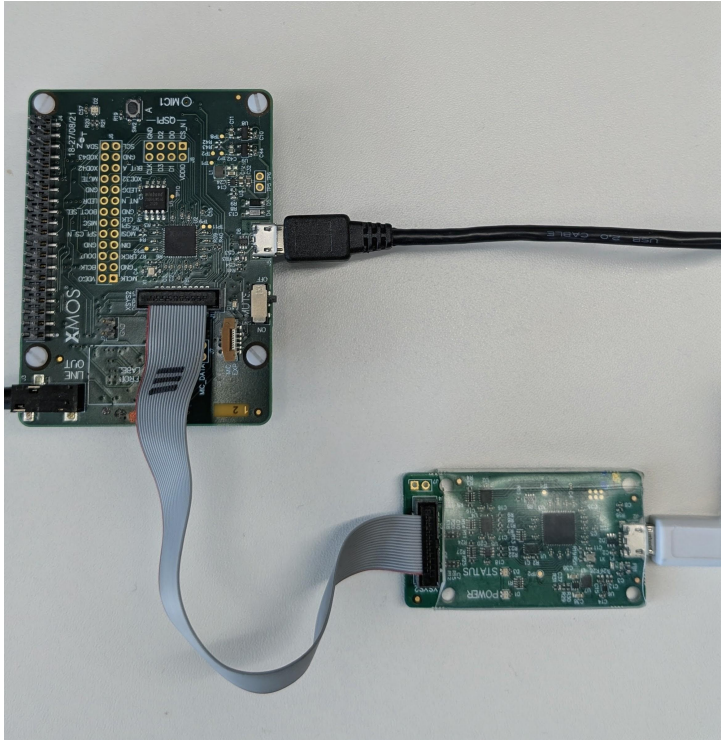


Fig. 2: XK-VOICE-L71 setup

```
cd examples
cd app_mic_array
cmake -G "Unix Makefiles" -B build
```

Any missing dependencies will be downloaded by the build system at this configure step.

Finally, the application binaries can be built using **xmake**:

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

The example build four configurations - **1mic_isr**, **1mic_thread**, **2mic_isr** and **2mic_thread**, which indicate the number of microphones used and whether the PDM RX service runs in an ISR or a hardware thread. The executable binaries (.xe files) for each configuration are placed in **bin/<config>** directory (e.g. **bin/2mic_isr/app_mic_array_2mic_isr.xe**).

Running the application

To run the application from the **/examples/app_mic_array** directory, execute:

```
xrun --xscope bin/2mic_isr/app_mic_array_2mic_isr.xe
```

The command above runs the **2mic_isr** configuration. To run a different configuration, replace **2mic_isr** in the path with the desired build variant.



When running, the application captures the microphones audio and routes it to the DAC on the **XK-VOICE-L71** after linearly scaling the PCM samples received from **lib_mic_array** by a factor of 64. Connect headphones to the 3.5mm LINE OUT jack on the board to listen to the captured microphone signal.

Other examples

app_custom_filter The **app_custom_filter** example behaves like **app_mic_array**, but instead of using the default decimation filters provided by the library, it uses a [custom decimation filter](#).

It demonstrates the use of [mic_array_init_custom_filter\(\)](#), the mic array initialization API for custom filters. The filter used is **good_2_stage_filter_int.pkl**, which is generated by running the filter design script **design_filter.py**. This filter is then quantized and interleaved into the format required by the mic array component by running the **combined.py** script, as described in [Designing a custom filter](#). The script produces a header file (**good_2_stage_filter.h**) containing the first and second stage filter coefficients and related defines. This header is included in the example and is located in the **app_custom_filter/src** directory. For details, see [Using custom filters](#).

app_par_decimator The **app_par_decimator** example behaves like **app_mic_array**, but internally uses a [custom mic array](#) decimator implementation. The custom decimator runs in two threads, with each thread processing one channel of microphone output. This example serves as a reference for using **lib_mic_array** in applications with many microphone channels, where decimation must be split across multiple hardware threads (e.g., two decimator threads, each handling four channels).

app_shutdown The **app_shutdown** example demonstrates shutting down and restarting the mic array each time **Button A** on the **XK-VOICE-L71** is pressed. Each press shuts down the mic array component and restarts it at a different output sampling frequency, cycling between 16 kHz and 48 kHz.

The terminal stdout displays the current sampling rate on each button press:

```
Starting at sample rate 16000
Starting at sample rate 48000
Starting at sample rate 16000
Starting at sample rate 48000
Starting at sample rate 16000
Starting at sample rate 48000
Starting at sample rate 16000
```

5 Resource usage

The mic array unit requires several kinds of hardware resources, including ports, clock blocks, chanends, hardware threads, compute time (MIPS) and memory.

This page attempts to capture the requirements for each hardware type with relevant configurations.

Warning

The usage information below applies when the default usage model is used. Resource usage in an application which uses custom mic array sub-components will depend crucially on the specifics of the customization.



5.1 Discrete Resources

Resource	Count
port	3
clock block	1 (SDR) 2 (DDR)
chanend	4
thread	1 or 2

Ports

In all configurations, the mic array unit requires 3 of the xcore.ai device's hardware ports. Two of these ports (for the master audio clock and PDM clock) must be 1-bit ports. The third (PDM capture port) can be 1-, 4- or 8-bit, depending on the microphone count and SDR/DDR configuration.

Clock Blocks

In applications which use an SDR microphone configuration, the mic array unit requires 1 of the xcore.ai device's 5 clock blocks. This clock block is used both to generate the PDM clock from the master audio clock and as the PDM capture clock.

In applications which use a DDR microphone configuration, the mic array unit requires 2 of the xcore.ai device's 5 clock blocks. One clock is used to generate the PDM clock from the master audio clock, and the other is used as the PDM capture clock (which must operate at different rates in a DDR configuration).

Chanends

Chanends are a hardware resource which allow threads (possibly running on different tiles) to communicate over channels. The mic array unit requires 4 chanends. Two are used for communication between the PDM rx service and the decimation thread. Two more are needed for transferring completed frames from the mic array unit to other application components.

Threads

The PDM rx service can run either as a stand-alone thread or as an interrupt within the decimator thread. Accordingly, the mic array requires one thread when the PDM rx service runs in interrupt mode, and two threads when it runs as a stand-alone thread.

Running PDM rx as a stand-alone thread modestly reduces the mic array unit's MIPS consumption by eliminating the context switch overhead of an interrupt. The cost of that is one hardware thread.

Note

When configured as an interrupt, PDM rx ISR is typically configured on the decimation thread, but this is not a strict requirement. The PDM rx interrupt can be configured for any thread **on the same tile** as the decimation thread. They must be on the same tile because shared memory is used between the two contexts.



5.2 Compute

The compute requirement of the mic array unit depends strongly on the actual configuration being used. The compute requirement is expressed in millions of instructions per second (MIPS) and is approximately linearly related to many of the configuration parameters.

Each tile of an xcore.ai device has 8 hardware threads and a 5 stage pipeline. The exact calculation of how many MIPS are available to a thread is complicated, and is, in general, affected by both the number of threads being used, as well as the work being done by each thread.

As a rule of thumb, however, the core scheduler will offer each thread a minimum of $CORE_CLOCK_MHZ/8$ millions of instruction issue slots per second (\sim MIPS), and no more than $CORE_CLOCK_MHZ/5$ millions of issue slots per second, where $CORE_CLOCK_MHZ$ is the core CPU clock rate (specified as **SystemFrequency** in the XN file). With a core clock rate of 600 MHz, that means that each core should expect at least 75 MIPS.

Table [Estimated MIPS \(per configuration\)](#) shows the mic array MIPS by profiling an application that includes the mic array. The application used to generate the MIPS numbers runs the [default](#) mic array API (so the decimator running in a single hardware thread) with all defines set to their default values as listed in [Configuration defines \(mic_array_conf_default.h\)](#) except for **MIC_ARRAY_CONFIG_MIC_COUNT** and **MIC_ARRAY_CONFIG_USE_PDM_ISR**. These two (along with the output sampling rate) are varied to build the different configurations that are profiled.

Table 1: Estimated MIPS (per configuration)

mic count	PDM RX	output samp freq	MIPS
1	ISR	16000	14.146
1	ISR	32000	17.234
1	ISR	48000	21.305
1	THREAD	16000	12.930
1	THREAD	32000	15.954
1	THREAD	48000	19.961
2	ISR	16000	29.310
2	ISR	32000	34.621
2	ISR	48000	41.934
2	THREAD	16000	27.006
2	THREAD	32000	32.285
2	THREAD	48000	39.533

Note

The MIPS numbers scale approximately linearly with the number of microphones. Although the table lists values only for the 1- and 2-mic configurations, these results can be extrapolated to estimate MIPS for configurations with a higher microphone count. If a given configuration cannot be accommodated within a single hardware thread, the decimator can be split across multiple threads to distribute the compute load.



This approach is not supported by the [default](#) mic array API. An example of a custom multi-threaded decimator implementation can be found in [app_par_decimator](#).

5.3 Memory

The memory cost of the mic array unit has three parts: code, stack and data. Code is the memory needed to store compiled instructions in RAM. Stack is the memory required to store intermediate results during function calls, and data is the memory used to store persistent objects, variables and constants.

Table [Memory usage \(in bytes\)](#) reports the memory usage of two minimal applications that include the mic array: one using the [default](#) mic array API and another using a [custom](#) configuration created by instantiating a [MicArray](#) object.

Both applications are built for **1 or 2 microphones** with a **16 kHz output sample rate**, with the other compile-time parameters set to their default values as defined in [Configuration defines \(mic_array_conf_default.h\)](#).

Memory for higher microphone counts can be extrapolated from the 1- and 2-mic numbers.

Across different sampling rates, memory usage with the default API remains unchanged since the default API compiles all the decimation filters included in the library. For custom usage, the data memory across different sampling rates varies depending on the [decimation filters](#) included.

Table 2: Memory usage (in bytes)

Config	Available on tile	Used	Stack	Code	Data
1mic_custom	524288	12572	572	8070	3930
1mic_default	524288	16580	636	9058	6886
2mic_custom	524288	13964	580	8278	5106
2mic_default	524288	18084	636	9378	8070

Note

The default API requires more memory than the custom configuration. The additional code memory comes from the wrapper and abstraction code included in the default API. The increased data memory results from the inclusion of filter coefficients for all filters provided by the library, whereas the custom build includes only the coefficients required by the specific [MicArray](#) instance.

Note

A minimal empty application (no mic array) occupies ~4.6 KiB (Stack: 356 B, Code: 3754 B, Data: 542 B). Subtract this baseline from the reported totals to isolate mic array overhead.



6 Software structure

The core of **lib_mic_array** are a set of C++ class templates representing the mic array unit and its sub-components.

The template parameters of these class templates are (mainly) used for two different purposes. Non-type template parameters are used to specify certain quantitative configuration values, such as the number of microphone channels or the second stage decimator tap count. Type template parameters, on the other hand, are used for configuring the behaviour of sub-components.

6.1 High level view

At the heart of the mic array API is the **MicArray** class template.

Note

All classes and class templates mentioned are in the **mic_array** C++ namespace unless otherwise specified. Additionally, this documentation may refer to class templates (e.g. **MicArray**) with unbound template parameters as “classes” when doing so is unlikely to lead to confusion.

The **MicArray** class template looks like the following:

```
template <unsigned MIC_COUNT,
         class TDecimator,
         class TPdmRx,
         class TSampleFilter,
         class TOutputHandler>
class MicArray;
```

Here the non-type template parameter **MIC_COUNT** indicates the number of microphone channels to be captured and processed by the mic array unit. Most of the class templates have this as a parameter.

A **MicArray** object comprises 4 sub-components:

Table 3: MicArray sub-components

Member Field	Component Class	Responsibility
PdmRx	TPdmRx	Capturing PDM data from a port
Decimator	TDecimator	2-stage decimation on blocks of PDM data
SampleFilter	TSampleFilter	Post-processing of decimated samples
OutputHandler	TOutputHandler	Transferring audio data to subsequent pipeline stages

Each of the **MicArray** sub-components has a type that is specified as a template parameter when the class template is instantiated. **MicArray** requires the class of each of its sub-components to implement a certain minimal interface. The **MicArray** object interacts with its sub-components using this interface.



Note

Abstract classes are **not** used to enforce this interface contract. Instead, the contract is enforced (at compile time) solely in how the **MicArray** object makes use of the sub-component.

The following diagram [Mic Array High Level Process](#) conceptually captures the flow of information through the **MicArray** sub-components.

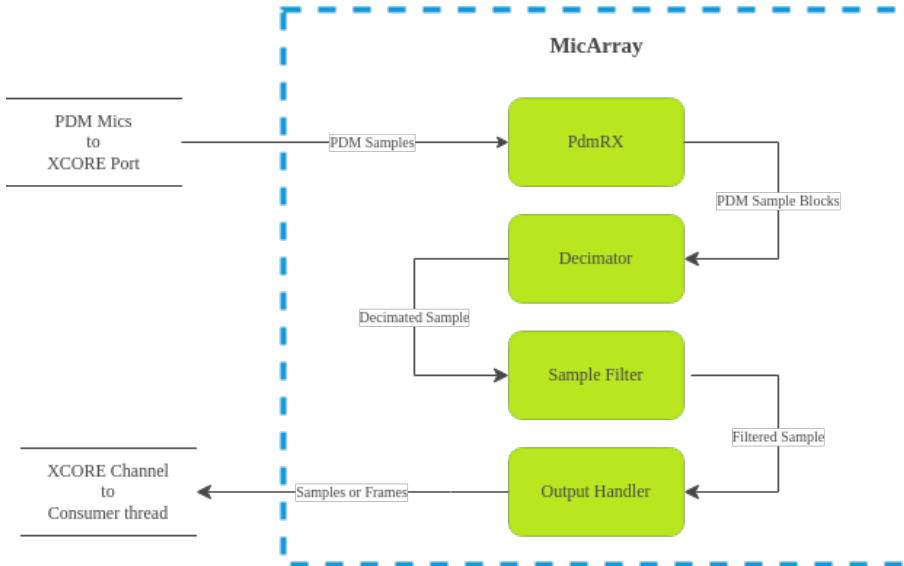


Fig. 3: Mic Array High Level Process

Note

MicArray does not enforce the use of an XCore port for collecting PDM samples or an XCore channel for transferring processed data. This is just the typical usage.

Mic Array / Decimator thread

Aside from aggregating its sub-components into a single logical entity, the **MicArray** class template also holds the high-level logic for capturing, processing and coordinating movement of the audio stream data.

The following code snippet is the implementation for the main mic array thread (or “decimation thread”; not to be confused with (optional) PDM capture thread).

```

int32_t sample_out[MIC_COUNT] = {0};
volatile bool shutdown = false;

while(!shutdown){
    uint32_t *pdm_samples = PdmRx.GetPdmBlock();
    Decimator.ProcessBlock(sample_out, pdm_samples);
    SampleFilter.Filter(sample_out);
    shutdown = OutputHandler.OutputSample(sample_out);
}
  
```

(continues on next page)



(continued from previous page)

```

}
PdmRx.Shutdown();
OutputHandler.CompleteShutdown();
}

```

The thread loops till `OutputHandler.OutputSample()` indicates a shutdown request and on each iteration,

- ▶ Requests a block of PDM sample data from the PDM rx service. This is a blocking call which only returns once a complete block becomes available.
- ▶ Passes the block of PDM sample data to the decimator to produce a single output sample.
- ▶ Applies a post-processing filter to the sample data.
- ▶ Passes the processed sample to the output handler to be transferred to the next stage of the processing pipeline. This may also be a blocking call, only returning once the data has been transferred.

Note that the **MicArray** object doesn't care how these steps are actually implemented. For example, one output handler implementation may send samples one at a time over a channel. Another output handler implementation may collect samples into frames, and use a FreeRTOS queue to transfer the data to another thread.

Sub-Component initialization

Each of **MicArray**'s sub-components may have implementation-specific configuration or initialization requirements. Each sub-component is a **public** member of **MicArray** (see [MicArray sub-components](#)). An application can access a sub-component directly to perform any type-specific initialization or other manipulation.

For example, the [ChannelFrameTransmitter](#) output handler class needs to know the **chanend** to be used for sending samples. This can be initialized on a **MicArray** object **mics** with `mics.OutputHandler.SetChannel(c_sample_out)`.

6.2 Sub-Components

PdmRx

[PdmRx](#), or the PDM RX service is the **MicArray** sub-component responsible for capturing PDM sample data, assembling it into blocks, and passing it along so that it can be decimated.

The **MicArray** class requires only that **PdmRx** implement `GetPdmBlock()`, a blocking call that returns a pointer to a block of PDM data which is ready for further processing and `Shutdown()`, which is called in the even of **MicArray** shutdown.

Generally speaking, **PdmRx** will derive from the [StandardPdmRxService](#) class template. **StandardPdmRxService** encapsulates the logic of using an xCore **port** for capturing PDM samples one word (32 bits) at a time, and managing two buffers where blocks of samples are collected. It also simplifies the logic of running PDM RX as either an interrupt or as a stand-alone thread.

It uses a streaming channel to transfer PDM blocks to the decimator. It also provides methods for installing an optimized ISR for PDM capture.



Decimator

The *Decimator* sub-component encapsulates the logic of converting blocks of PDM samples into PCM samples. The *TwoStageDecimator* class is a decimator implementation that uses a pair of decimating FIR filters to accomplish this.

The first stage has a fixed tap count of **256** and a fixed decimation factor of **32**. The second stage has a configurable tap count and decimation factor.

For more details, see *Decimation filters*.

SampleFilter

The *SampleFilter* sub-component is used for post-processing samples emitted by the decimator. Two implementations for the sample filter sub-component are provided by this library.

The *NopSampleFilter* class can be used to effectively disable per-sample filtering on the output of the decimator. It does nothing to the samples presented to it, and so calls to it can be optimized out during compilation.

The *DcoSampleFilter* class is used for applying the DC offset elimination filter to the decimator's output. The DC offset elimination filter is meant to ensure the sample mean for each channel tends toward zero.

For more details, see *Sample filters*.

OutputHandler

The *OutputHandler* sub-component is responsible for transferring processed sample data to subsequent processing stages.

There are two main considerations for output handlers. The first is whether audio data should be transferred *sample-by-sample* or as *frames* containing many samples. The second is the method of actually transferring the audio data.

The class **ChannelSampleTransmitter** sends samples one at a time to subsequent processing stages using an xCore channel.

The *FrameOutputHandler* class collects samples into frames, and uses a frame transmitter to send the frames once they're ready.

7 Decimation filters

The mic array unit provided by this library uses a two-stage decimation process, implemented in *TwoStageDecimator*, to convert a high sample rate stream of (1-bit) PDM samples into a lower sample rate stream of (32-bit) PCM samples. This is shown in *Simplified Decimator Model*.

The first stage filter is a decimating FIR filter with a fixed tap count (**S1_TAP_COUNT**) of **256** and a fixed decimation factor (**S1_DEC_FACTOR**) of **32**.

The second stage decimator is a fully configurable FIR filter with tap count **S2_TAP_COUNT** and a decimation factor of **S2_DEC_FACTOR** (this can be 1).

7.1 Filters provided as part of lib_mic_array

lib_mic_array provides first and second stage decimation filter coefficients for filters targeting output sampling rates of 16 kHz, 32 kHz and 48 kHz from a starting input PDM



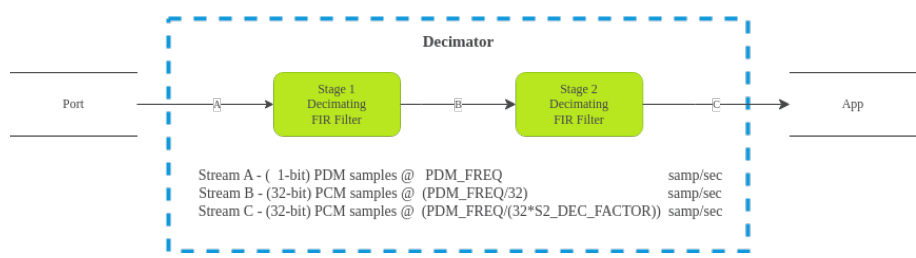



Fig. 4: Simplified Decimator Model

frequency of 3.072 MHz. The first stage decimation filters have a fixed decimation factor of 32 and a fixed tap count of 256.

The second stage filters decimation factors vary based on the output sampling rate.

These filters are available to include in the application, through the header `mic_array/etc/filters_default.h`

 **Note**

Although the first stage has a fixed decimation factor of 32, its coefficients must differ for 16 kHz, 32 kHz, and 48 kHz output sampling rate paths. The passband must be extended appropriately so sufficient bandwidth is preserved before the second stage decimates by 6 (output sampling rate 16 kHz), 3 (output sampling rate 32 kHz), or 2 (output sampling rate 48 kHz).

The increased sample rate will place a higher MIPS burden on the processor. The typical MIPS usage (see section [Resource usage](#)) is in the order of 11 MIPS per channel using a 16 kHz output decimator.

Increasing the output sample rate to 32 kHz using the same length filters will increase processor usage per channel to approximately 13 MIPS rising to 15.6 MIPS for 48 kHz.

Increasing the filter lengths to 148 and 96 for stages 1 and 2 respectively at 48 kHz will increase processor usage per channel to around 20 MIPS.

Stage 1 filters

Table 4: Provided first stage decimation filters

Output PCM sampling rate	Decima- tion factor	Tap count	Coeffs
16 kHz	32	256	stage1_coef
32 kHz	32	256	stage1_32k_coefs
48 kHz	32	256	stage1_48k_coefs



Stage 2 filters

Table 5: Provided second stage decimation filters

Output PCM sam- pling Rate	Deci- mation factor	Tap count	Coeffs	Right shift
16 kHz	6	256	stage2_coef	stage2_shr
32 kHz	3	96	stage2_32k_coefs	stage2_32k_shift
48 kHz	2	96	stage2_48k_coefs	stage2_48k_shift

Filter characteristics

This sections provides filters characteristics of the filters provided as part of `lib_mic_array`.

16 kHz output PCM sampling rate filter

[16 kHz output sampling rate filter freq response](#) shows the frequency response of the first and second stages of the provided 16 kHz sampling rate filters as well as the cascaded overall response.

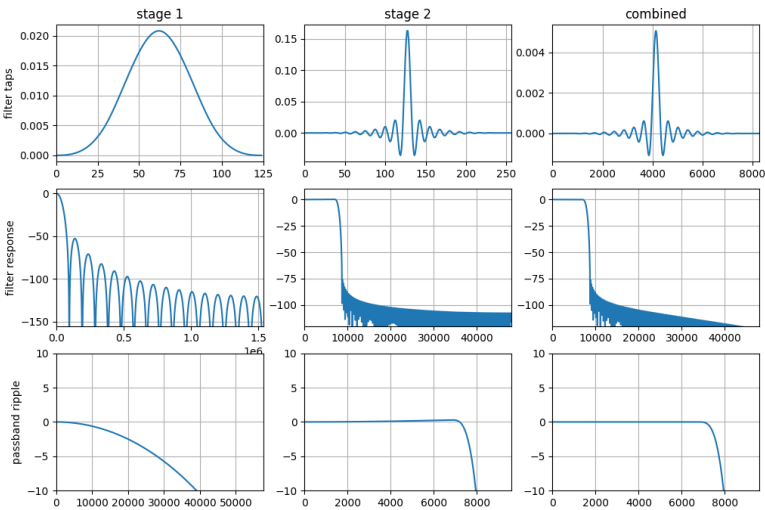


Fig. 5: 16 kHz output sampling rate filter freq response

32 kHz output PCM sampling rate filter

[32 kHz output sampling rate filter freq response](#) shows the frequency response of the first and second stages of the provided 32 kHz sampling rate filters as well as the cascaded overall response. Note that the overall combined response provides a nice flat passband.



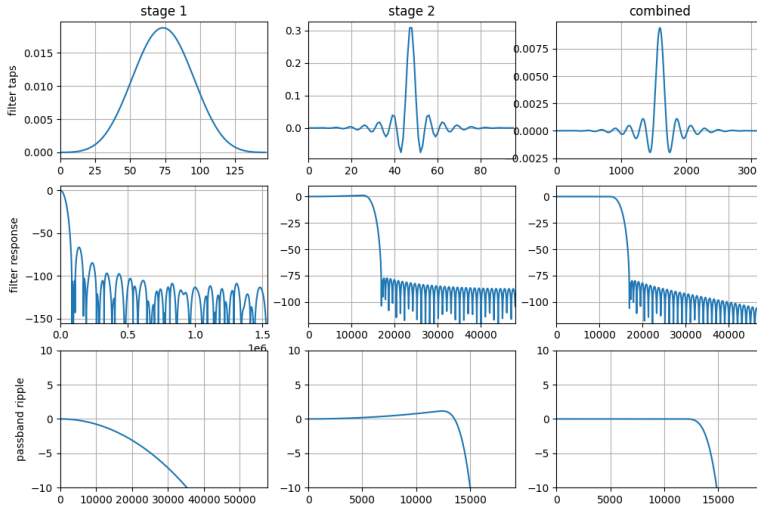


Fig. 6: 32 kHz output sampling rate filter freq response

48 kHz output PCM sampling rate filter

[48 kHz output sampling rate filter freq response](#) shows the frequency response of the first and second stages of the provided 48 kHz sampling rate filters as well as the cascaded overall response. Note that the overall combined response provides a nice flat passband.

The following sections provide more details about the first and second stage decimation filters, implemented in [TwoStageDecimator](#).

7.2 Decimator stage 1

For the first stage decimating FIR filter, the actual filter coefficients used are configurable, so an application is free to use a custom first stage filter, as long as the tap count is **256** and the decimation factor is **32**.

Filter implementation (Stage 1)

The input to the first stage decimator (here called “Stream A”) is a stream of 1-bit PDM samples with a sample rate of `PDM_FREQ`. Rather than each PDM sample representing a value of **0** or **1**, each PDM sample represents a value of either **+1** or **-1**. Specifically, on-chip and in-memory, a bit value of **0** represents **+1** and a bit value of **1** represents **-1**.

The output from the first stage decimator, Stream B, is a stream of 32-bit PCM samples with a sample rate of $\text{PDM_FREQ}/\text{S1_DEC_FACTOR} = \text{PDM_FREQ}/32$. For example, if `PDM_FREQ` is 3.072 MHz, then Stream B’s sample rate is 96.0 kHz.

The first stage filter is structured to make optimal use of the XCore XS3 vector processing unit (VPU), which can compute the dot product of a pair of 256-element 1-bit vectors in a single cycle. The first stage uses 256 16-bit coefficients for its filter taps.



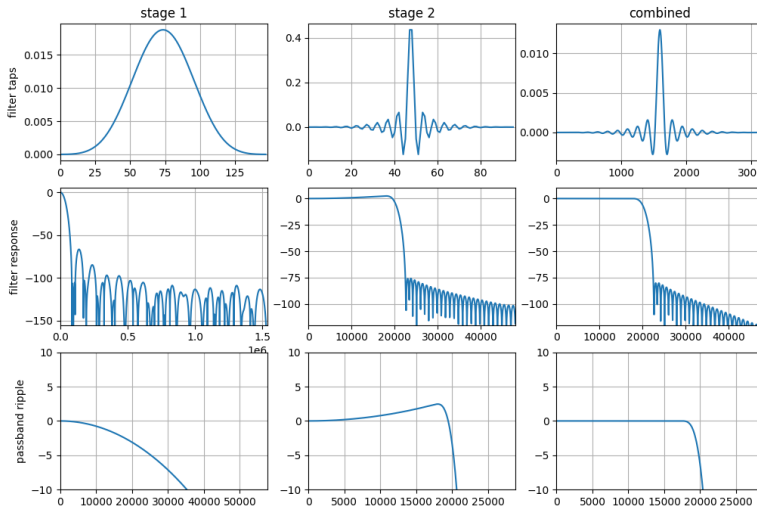


Fig. 7: 48 kHz output sampling rate filter freq response

The signature of the filter function is

```
int32_t fir_1x16_bit(uint32_t signal[8], uint32_t coeff_1[]);
```

Each time 32 PDM samples (1 word) become available for an audio channel, those samples are shifted into the 8-word (256-bit) filter state, and a call to `fir_1x16_bit` results in 1 Stream B sample element for that channel.

The actual implementation for the first stage filter can be found in `src/fir_1x16_bit.S`. Additional usage details can be found in `api/etc/fir_1x16_bit.h`.

Note that the 256 16-bit filter coefficients are **not** stored in memory as a standard coefficient array (i.e. `int16_t filter[256] = {b[0], b[1], ... };`). Rather, in order to take advantage of the VPU, the coefficients must be rearranged bit-by-bit into a block form suitable for VPU processing.

The filter state (delay line) consists of 256 one-bit PDM samples (equal to the number of filter taps) and requires a buffer of 8 unsigned 32-bit words for storage.

Filter Conversion Script

Taking a set of floating-point coefficients, quantizing them into 16-bit coefficients and 'boggling' them into the correct memory layout can be a tricky business. To simplify this process, this library provides a Python3 script that does this process automatically.

The script can be found in this repository at `python/stage1.py`.



7.3 Decimator stage 2

An application is free to supply its own second stage filter.

Filter implementation (Stage 2)

The input to the second stage decimator (here called "Stream B") is the stream of 32-bit PCM samples emitted from the first stage decimator with a sample rate of `PDM_FREQ/32`.

The output from the second stage decimator, Stream C, is a stream of 32-bit PCM samples with a sample rate of `PDM_FREQ/(32*S2_DEC_FACTOR)`. For example, if `PDM_FREQ` is 3.072 MHz, and `S2_DEC_FACTOR` is 6, then Stream C's sample rate (the sample rate received by the main application code) is

$$3.072 \text{ MHz} / (32*6) = 16 \text{ kHz}$$

The second stage filter uses the 32-bit FIR filter implementation from [lib_xcore_math](#). See `xs3_filter_fir_s32()` in that library for more implementation details.

The filter state (delay line) consists of as many 32-bit samples as there are taps in the stage-2 filter, and requires those many 32-bit words for storage.

8 Custom decimation filters

In the [TwoStageDecimator](#), the tap count and decimation factor for the first stage decimator are fixed to 256 and 32 respectively, as described in [Decimator stage 1](#).

These parameters cannot be changed without implementing a custom decimator, which is outside the scope of this document.

However, both the first-stage and second-stage filter coefficients may be replaced, and the second-stage decimation factor and tap count may be freely modified by running the mic array component with custom filters. This is described in the following sections.

8.1 Designing a custom filter

A filter design script is provided in `python/filter_design/design_filter.py`. The script contains functions to generate the filters currently provided as part of `lib_mic_array` and save them as `.pk1` files. Using these functions as a guide, the script can be extended to generate custom filters tailored to the application's needs.

Note that in [TwoStageDecimator](#), both the first and second stage filters are implemented using fixed-point arithmetic, which requires the coefficients to be presented in a specific format. The helper scripts `python/stage1.py` and `python/stage2.py` generate the correctly quantized and interleaved coefficient arrays required by the library.

After generating a `.pk1` file, the helper scripts `stage1.py` and `stage2.py` can be used to format the first and second stage filters, respectively, into the fixed-point C arrays required by the library.

Alternatively, the `combined.py` script can process both the first and second stage filters in one step. When executed, `stage1.py`, `stage2.py`, and `combined.py` print the filter coefficients as C-style arrays, along with filter-related defines such as tap count, decimation factor, etc., as `#define` macros on stdout.



The scripts can also be run with the `--file-prefix <prefix>` (or `-fp <prefix>`) option. In this mode, all arrays and defines are written into a header file (`<prefix>.h`), which can be included in the application.

Running `python combined.py --help` from the `python` directory shows full usage instructions.

From the `python` directory, the workflow is typically:

```
python filter_design/design_filter.py          # generates the
python combined.py <custom_filter.pkl> -fp <prefix> # .pkl files
                                                    # converts the
                                                    # .pkl file to C
                                                    # arrays for stage
                                                    # 1 and 2, and
                                                    # writes to
                                                    # <prefix>.h
```

8.2 Using custom filters

When using the `TwoStageDecimator` provided by the library, the `mic_array_init_custom_filter()` function is used to initialize a mic array instance with a custom 2-stage decimation filter.

Note

The custom filter provided to `mic_array_init_custom_filter()` must be compatible with the `TwoStageDecimator` requirements. Specifically, it must be a 2-stage filter. The tap count and decimation factor for the first-stage decimator are fixed at 256 and 32, respectively, and the filter must be compatible with the [Filter implementation \(Stage 1\)](#).

The second-stage decimation filter tap count and decimation ratio are flexible, provided it is a standard FIR filter compatible with [Filter implementation \(Stage 2\)](#). Using custom filters that are incompatible with the implementation in `TwoStageDecimator` is outside the scope of this documentation.

The `mic_array_conf_t` structure is populated with the decimator and PDM RX configurations before calling `mic_array_init_custom_filter()`. In particular, the application must:

- ▶ Allocate all filter coefficient buffers, filter state buffers, and PDM RX buffers referenced by the decimator and PDM RX configurations, and ensure they persist for the lifetime of the mic array instance (until `mic_array_start()` returns).
- ▶ Populate the decimator pipeline configuration (`mic_array_decimator_conf_t`) and the PDM RX configuration (`pdm_rx_config_t`) structures.

Note

When designing custom filters as described in [Designing a custom filter](#), the filter coefficient arrays and associated configuration parameters can be written to a header file by running `combined.py`. This file is included in the application, and the decimation filter coefficients and other parameters in `mic_array_filter_conf_t` are populated from it.



The application must still allocate persistent runtime buffers for the filter delay line ([mic_array_filter_conf_t](#)) and for the PDM RX input and output buffers ([pdm_rx_conf_t](#)).

Below is a snippet from the [app_custom_filter](#) source code showing how the [mic_array_conf_t](#) structure is populated:

```
#include "good_2_stage_filter.h" // Autogenerated by running 'python combined.py filter_design/good_2_stage_filter_
↪ int.pk1 -fp good_2_stage_filter'

void init_mic_conf(mic_array_conf_t &mic_array_conf, mic_array_filter_conf_t (&filter_conf)[2], unsigned
↪ *channel_map)
{
    static int32_t stg1_filter_state[APP_MIC_COUNT][8];
    static int32_t stg2_filter_state[APP_MIC_COUNT][GOOD_2_STAGE_FILTER_STG2_TAP_COUNT];
    memset(&mic_array_conf, 0, sizeof(mic_array_conf_t));

    //decimator
    mic_array_conf.decimator_conf.filter_conf = &filter_conf[0];
    mic_array_conf.decimator_conf.num_filter_stages = 2;
    // filter stage 1
    filter_conf[0].coef = (int32_t*)good_2_stage_filter_stg1_coef;
    filter_conf[0].num_taps = GOOD_2_STAGE_FILTER_STG1_TAP_COUNT;
    filter_conf[0].decimation_factor = GOOD_2_STAGE_FILTER_STG1_DECIMATION_FACTOR;
    filter_conf[0].state = (int32_t*)stg1_filter_state;
    filter_conf[0].shr = GOOD_2_STAGE_FILTER_STG1_SHR;
    filter_conf[0].state_words_per_channel = filter_conf[0].num_taps/32; // works on 1-bit samples
    // filter stage 2
    filter_conf[1].coef = (int32_t*)good_2_stage_filter_stg2_coef;
    filter_conf[1].num_taps = GOOD_2_STAGE_FILTER_STG2_TAP_COUNT;
    filter_conf[1].decimation_factor = GOOD_2_STAGE_FILTER_STG2_DECIMATION_FACTOR;
    filter_conf[1].state = (int32_t*)stg2_filter_state;
    filter_conf[1].shr = GOOD_2_STAGE_FILTER_STG2_SHR;
    filter_conf[1].state_words_per_channel = GOOD_2_STAGE_FILTER_STG2_TAP_COUNT;

    // pdm rx
    static uint32_t pdmr_x_out_block[APP_MIC_COUNT][GOOD_2_STAGE_FILTER_STG2_DECIMATION_FACTOR];
    static uint32_t __attribute__((aligned(8))) pdmr_x_out_block_double_buf[2][APP_MIC_COUNT * GOOD_2_STAGE_FILTER_
↪ STG2_DECIMATION_FACTOR];
    mic_array_conf.pdmrx_conf.pdm_out_words_per_channel = GOOD_2_STAGE_FILTER_STG2_DECIMATION_FACTOR;
    mic_array_conf.pdmrx_conf.pdm_out_block = (uint32_t*)pdmrx_out_block;
    mic_array_conf.pdmrx_conf.pdm_in_double_buf = (uint32_t*)pdmrx_out_block_double_buf;
    mic_array_conf.pdmrx_conf.channel_map = channel_map;
}
```

Once [mic_array_conf_t](#) is populated, mic array can be initialised by calling [mic_array_init_custom_filter\(\)](#) and started by calling [mic_array_start\(\)](#):

```
unsigned channel_map[2] = {0,1};
mic_array_conf_t mic_array_conf;
mic_array_filter_conf_t filter_conf[2];
init_mic_conf(mic_array_conf, filter_conf, channel_map);
mic_array_init_custom_filter(&pdm_res, &mic_array_conf);

par {
    mic_array_start((chanend_t) c_audio_frames);

    <... other tasks ...>
}
```

Note

Apart from calling [mic_array_init_custom_filter\(\)](#) instead of [mic_array_init\(\)](#), all other aspects of using the custom filter API, such as including the mic array in an application, declaring resources, and overriding build-time default configuration, are exactly the same as in the default usage model described in [Using lib_mic_array](#).



9 Advanced usage

Note

Even when using a custom integration, including the library in an application follows the same procedure as described in [Including in an application](#).

The advanced use method requires familiarity with the mic array [software structure](#) and a basic understanding of C++.

A custom integration involves creating a custom [MicArray](#) object. This allows overriding default behaviour such as decimation, PDM reception etc.

Before creating a custom object, the required hardware resources must be identified. This process is the same as described in [Identify hardware resources](#).

Configuring the mic array using a custom [MicArray](#) object requires a C++ source file to be added to the application.

The following sections describe the general structure of this file.

9.1 Declare resources

The first step is to declare a `pdm_rx_resources_t` struct listing the hardware resources.

```
pdm_rx_resources_t pdm_res = PDM_RX_RESOURCES_DDR(
    PORT_MCLK_IN,
    PORT_PDM_CLK,
    PORT_PDM_DATA,
    MCLK_FREQ,
    PDM_FREQ,
    XS1_CLKBLK_1,
    XS1_CLKBLK_2);
```

9.2 Construct MicArray object

Next, instantiate a [MicArray](#) object. The example below creates the object with the standard component classes that are provided by the library. One or more of these can be overridden by a custom class, provided it implements the [interface](#) required by the [MicArray](#):

```
using TMicArray = mic_array::MicArray<APP_N_MICS,
    mic_array::TwoStageDecimator<APP_N_MICS,
        STAGE2_DEC_FACTOR_48KHZ,
        MIC_ARRAY_48K_STAGE_2_TAP_COUNT>,
    mic_array::StandardPdmRxService<APP_N_MICS_IN,
        APP_N_MICS,
        STAGE2_DEC_FACTOR_48KHZ>,
    typename std::conditional<APP_USE_DC_ELIMINATION,
        mic_array::DcoeSampleFilter<APP_N_MICS>,
        mic_array::NopSampleFilter<APP_N_MICS>>::type,
    mic_array::FrameOutputHandler<APP_N_MICS,
        MIC_ARRAY_CONFIG_SAMPLES_PER_FRAME,
        mic_array::ChannelFrameTransmitter>>;

TMicArray mics;
```

- ▶ `TwoStageDecimator`, `StandardPdmRxService`, `DcoeSampleFilter`, and `FrameOutputHandler` can all be replaced with custom classes if needed.
- ▶ Any custom class must implement the same interface expected by [MicArray](#).



Note

The `examples/app_par_decimator` example demonstrates replacing the standard decimator class with a custom multi-threaded decimator.

Note

If the application requires custom decimation filters but they're compatible with the [TwoStageDecimator](#) implementation, refer to [Custom decimation filters](#) to see how to do so.

9.3 Define app-callable functions

Define the functions that the application will call to initialise and run the mic array - `app_mic_array_init()` and `app_mic_array_start()`:

```
MA_C_API
void app_mic_array_init()
{
    mics.Decimator.Init((uint32_t*) stage1_48k_coefs, stage2_48k_coefs, stage2_48k_shift); // Replace stage1_48k_
    ← coefs with custom filter coefs if needed
    mics.PdmRx.Init(pdm_res.p_pdm_mics);
    mic_array_resources_configure(&pdm_res, MCLK_DIVIDER);
    mic_array_pdm_clock_start(&pdm_res);
}

DECLARE_JOB(ma_task_start_pdm, (TMicArray&));
void ma_task_start_pdm(TMicArray& m){
    m.PdmRx.ThreadEntry();
}

DECLARE_JOB(ma_task_start_decimator, (TMicArray&, chanend_t));
void ma_task_start_decimator(TMicArray& m, chanend_t c_audio_frames){
    m.ThreadEntry();
}

MA_C_API
void app_mic_array_task(chanend_t c_frames_out)
{
    #if APP_USE_PDMRX_ISR
        CLEAR_KEDI()

        mics.OutputHandler.FrameTx.SetChannel(c_frames_out);
        // Setup the ISR and enable. Then start decimator.
        mics.PdmRx.AssertOnDroppedBlock(false);
        mics.PdmRx.InstallISR();
        mics.PdmRx.UnmaskISR();
        mics.ThreadEntry();
    #else
        mics.OutputHandler.FrameTx.SetChannel(c_frames_out);
        PAR_JOBS(
            PJOB(ma_task_start_pdm, (mics)),
            PJOB(ma_task_start_decimator, (mics, c_frames_out))
        );
    #endif
}
```

- `app_mic_array_init()` initialises the component classes and configures the hardware resources. Note that things like initialising the decimator with custom filters would be done here by passing said filters to the `mics.Decimator.Init()` call.
- `app_mic_array_task()` starts the PDM and decimator threads or ISR-based processing.

9.4 Application main function

The application main function is modified to include calls to `app_mic_array_init()` and `app_mic_array_task()`



```
int main() {
    par {
        on tile[1]: {
            app_mic_array_init();
            par {
                app_mic_array_task((chanend_t) c_audio_frames);
                <... other application threads ...>
            }
        }
    }
    return 0;
}
```

10 Sample filters

Following the two-stage decimation procedure is an optional post-processing stage called the sample filter. This stage operates on each sample emitted by the second stage decimator, one at a time, before the samples are handed off for framing or transfer to the rest of the application's audio pipeline.

Note

This is represented by the **SampleFilter** sub-component of the **MicArray** class template.

An application may implement its own sample filter in the form of a C++ class which implements the **Filter()** function as required by **MicArray**. See the implementation of **DcoeSampleFilter** for a simple example.

10.1 DC Offset elimination

The current version of this library provides a simple IIR filter called DC Offset Elimination (DCOE) that can be used as the sample filter. This is a high-pass filter meant to ensure that each audio channel will tend towards a mean sample value of zero.

Enabling/Disabling DCOE

Whether the DCOE filter is enabled by default and how to enable or disable it depends on which approach your project uses to include the mic array component in the application.

Default model DCOE is **enabled** by default in **mic_array_conf_default.h**. To disable, override **MIC_ARRAY_CONFIG_USE_DC_ELIMINATION** through **mic_array_conf.h** file in the application or in the application's CMakeLists.txt.

Advanced usage model If the project does not use the default models to include the mic array unit in the application, then precisely how the DCOE filter is included may depend on the specifics of the application. In general, however, the DCOE filter will be enabled by using **DcoeSampleFilter** as the **TSampleFilter** template parameter for the **MicArray** class template.

For example, sub-classing **mic_array::MicArray** as follows will enable DCOE for any **MicArray** implementation deriving from that sub-class.

```
#include "mic_array/cpp/MicArray.hpp"
using namespace mic_array;
...
template <unsigned MIC_COUNT, class TDecimator,
         class TPdMRx, class TOutputHandler>
class DcoeEnabledMicArray : public MicArray<MIC_COUNT, TDecimator, TPdMRx,
         DcoeSampleFilter, TOutputHandler>
{
    ...
};
```



DCOE filter equation

As mentioned above, the DCOE filter is a simple IIR filter given by the following equation, where $x[t]$ and $x[t-1]$ are the current and previous input sample values respectively, and $y[t]$ and $y[t-1]$ are the current and previous output sample values respectively.

```
R = 252.0 / 256.0
y[t] = R * y[t-1] + x[t] - x[t-1]
```

DCOE filter frequency response

The plot below indicates the frequency response of DCOE filter [DCOE filter frequency response](#).

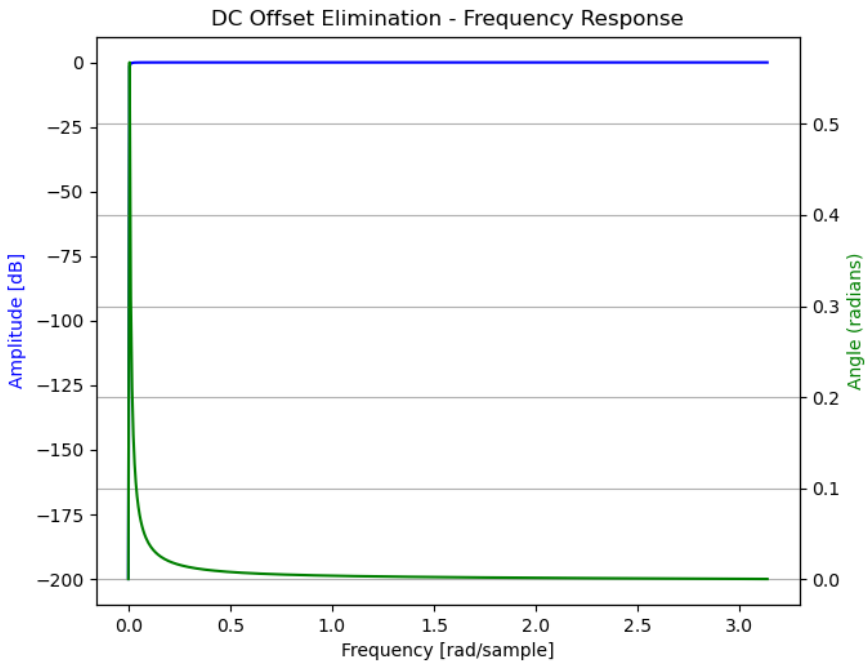


Fig. 8: DCOE filter frequency response

11 API Reference

11.1 C++ API Reference

MicArray

template<unsigned **MIC_COUNT**, class **TDecimator**, class **TPdmRx**, class **TSampleFilter**, class **TOutputHandler**>

class **MicArray**

Represents the microphone array component of an application.

Template Parameters



- ▶ **MIC_COUNT** – Number of microphone output channels from the mic array component
- ▶ **TDecimator** – Type for the decimator. See [Decimator](#).
- ▶ **TPdmRx** – Type for the PDM rx service used. See [PdmRx](#).
- ▶ **TSampleFilter** – Type for the output filter used. See [SampleFilter](#).
- ▶ **TOutputHandler** – Type for the output handler used. See [OutputHandler](#).

Public Functions

inline **MicArray**()

Construct a [MicArray](#).

This constructor uses the default constructor for each of its components, [PdmRx](#), [Decimator](#), [SampleFilter](#), and [OutputHandler](#).

void **ThreadEntry**()

Entry point for the decimation thread.

This function does not return. It loops indefinitely, collecting blocks of PDM data from [PdmRx](#) (which must have already been started), uses [Decimator](#) to filter and decimate the sample stream to the output sample rate, applies any post-processing with [SampleFilter](#), and then delivers the stream of output samples through [OutputHandler](#).

Public Members

[TPdmRx](#) **PdmRx**

The PDM rx service.

The template parameter **TPdmRx** is the concrete class implementing the microphone array's PDM rx service, which is responsible for collecting PDM samples from a port and delivering them to the decimation thread.

TPdmRx is only required to implement one function, **GetPdmBlock()**:

```
uint32_t* GetPdmBlock();
```

GetPdmBlock() returns a pointer to a block of PDM data, formatted as expected by the decimator. **GetPdmBlock()** is called *from the decimator thread* and is expected to block until a new full block of PDM data is available to be decimated.

For example, [StandardPdmRxService::GetPdmBlock\(\)](#) waits to receive a pointer to a block of PDM data from a streaming channel. The pointer is sent from the PdmRx interrupt (or thread) when the block has been completed. This is used for capturing PDM data from a port.

[TDecimator](#) **Decimator**

The Decimator.

The template parameter **TDecimator** is the concrete class implementing the microphone array's decimation procedure. **TDecimator** is only required to implement one function, **ProcessBlock()**:

```
void ProcessBlock(
    int32_t sample_out[MIC_COUNT],
    uint32_t *pdm_block);
```

ProcessBlock() takes a block of PDM samples via its **pdm_block** parameter, applies the appropriate decimation logic, and outputs a single (multi-channel) sample via its **sample_out** parameter. The size and formatting of the PDM block expected by the decimator depends on its particular implementation.



TSampleFilter **SampleFilter**

The output filter.

The template parameter **TSampleFilter** is the concrete class implementing the microphone array's sample filter component. This component can be used to apply additional non-decimating, non-interpolating filtering of samples. **TSampleFilter()** is only required to implement one function, **Filter()**:

```
void Filter(int32_t sample[MIC_COUNT]);
```

Filter() takes a single (multi-channel) sample from the decimator component's output and may update the sample in-place.

For example a sample filter based on the *DcoeSampleFilter* class template applies a simple first-order IIR filter to the output of the decimator, in order to eliminate the DC component of the audio signals.

If no additional filtering is required, the *NopSampleFilter* class template can be used for **TSampleFilter**, which leaves the sample unmodified. In this case, it is expected that the call to *NopSampleFilter::Filter()* will ultimately get completely eliminated at build time. That way no addition run-time compute or memory costs need be introduced for the additional flexibility.

Even though **TDecimator** and **TSampleFilter** both (possibly) apply filtering, they are separate components of the *MicArray* because they are conceptually independent.

TOutputHandler **OutputHandler**

The output handler.

The template parameter **TOutputHandler** is the concrete class implementing the microphone array's output handler component. After the PDM input stream has been decimated to the appropriate output sample rate, and after any post-processing of that output stream by the sample filter, the output samples must be delivered to another thread for any additional processing. It is the responsibility of this component to package and deliver audio samples to subsequent processing stages.

TOutputHandler is only required to implement one function, **OutputSample()**:

```
void OutputSample(int32_t sample[MIC_COUNT]);
```

OutputSample() is called exactly once for each mic array output sample. **OutputSample()** may block if necessary until the subsequent processing stage ready to receive new data. However, the decimator thread (in which **OutputSample()** is called) as a whole has a real-time constraint - it must be ready to pull the next block of PDM data while it is available.



StandardPdmRxService

struct **pdm_rx_isr_context_t**

PDM rx interrupt configuration and context.

Public Members

port_t **p_pdm_mics**

Port on which PDM samples are received.

uint32_t ***pdm_buffer**[2]

Pointers to a pair of buffers used for storing captured PDM samples.

The buffers themselves are allocated by the application and passed to [mic_array::StandardPdmRxService::Init](#). The idea is that while the PDM rx ISR is filling one buffer, the decimation thread is busy processing the contents of the other buffer. If the real-time constraint is maintained, the decimation thread will be finished with the contents of its buffer before the PDM rx ISR fills the other buffer. Once full, the PDM rx ISR does a double buffer pointer swap and hands the newly-filled buffer to the decimation thread.

unsigned **phase**

Tracks the completeness of the buffer currently being filled.

Each read of samples from **p_pdm_mics** gives one word of data. This variable tracks how many more port reads are required before the current buffer has been filled.

unsigned **phase_reset**

The number of words to read from **p_pdn_mics** to fill a buffer.

chanend_t **c_pdm_data**

Streaming chanend the PDM rx ISR uses to signal the decimation thread that another buffer is full and ready to be processed.

The streaming channel itself is allocated by [mic_array::StandardPdmRxService](#), which owns the other end of the channel.

unsigned **credit**

Used for detecting when the real-time constraint is violated by the decimation thread.

Each time the decimation thread is given a block of PDM data to process, **credit** is reset to 2. Each time the PDM rx ISR hands a block of PDM data to the decimation thread, this is decremented.

Deadlock Condition

[mic_array::StandardPdmRxService](#) uses a streaming channel to facilitate communication between the two execution contexts used by the mic array, the decimation thread and the PDM rx ISR. A streaming channel is used because it allows the contexts to operate asynchronously.

A channel has a 2 word buffer, and as long as there is room in the buffer, an **OUT** instruction putting a word (in this case, a pointer) into the channel is guaranteed not to block. This is important because the PDM rx ISR is typically configured on the same hardware thread as the decimation thread.



If a thread is blocked on an **OUT** instruction to a channel, in order to unblock the thread, an **IN** must be issued on the other end of that channel. But because the PDM rx ISR is blocked, it cannot hand control back to the decimation thread, which means the decimation thread can never issue an **IN** instruction to unblock the ISR. The result is a deadlock.

Unfortunately, there is no way for a thread to query a chanend to determine whether it will block if an **OUT** instruction is issued. That is why **credit** is used. Before issuing an **OUT** to **c_pdm_data**, the PDM rx ISR checks whether **credit** is non-zero. If so, the ISR issues the **OUT** instruction as normal and decrements **credit**.

If **credit** is zero, the default behavior of PDM rx ISR is to raise an exception (**ET_ECALL**). This reflects the idea that it is generally better if system-breaking errors loudly announce themselves (at least by default). If using [mic_array::StandardPdmRxService](#), this behavior can be changed by passing **false** in a call to [mic_array::StandardPdmRxService::AssertOnDroppedBlock\(\)](#), which will allow blocks of PDM data to be silently dropped (while still avoiding a permanent deadlock).

unsigned **missed_blocks**

Controls and records anti-deadlock behavior.

If the PDM rx ISR finds that **credit** is 0 when it's time to send a filled buffer to the decimation thread, it uses **missed_blocks** to control whether the PDM rx ISR should raise an exception or silently drop the block of PDM data.

If **missed_blocks** is -1 (its default value) an exception is raised. Otherwise **missed_blocks** is used to record the number of blocks that have been quietly dropped.

[pdm_rx_isr_context_t](#) **pdm_rx_isr_context**

Configuration and context of the PDM rx ISR when [mic_array::StandardPdmRxService](#) is used in interrupt mode.

pdm_rx_isr (**pdm_rx_isr.S**) directly allocates this object as configuration and state parameters required by that interrupt routine.

static inline void **enable_pdm_rx_isr** (const port_t p_pdm_mics)

Configure port to use **pdm_rx_isr** as an interrupt routine.

This function configures **p_pdm_mics** to use **pdm_rx_isr** as its interrupt vector and enables the interrupt on the current hardware thread.

This function does NOT unmask interrupts.

Parameters

p_pdm_mics – Port resource to enable ISR on.

template<unsigned **CHANNELS_IN**, unsigned **CHANNELS_OUT**>

class **StandardPdmRxService**

PDM rx service which collects PDM sample data from a port and uses a streaming channel to send a block of data by pointer further down the mic array pipeline.

This class template is intended to be used for the **TPdmRx** template parameter of [MicArray](#), where it represents the [MicArray::PdmRx](#) component of the mic array.

This class can run the PDM rx service either as a stand-alone thread or through an interrupt.



Inter-context Transfer

A streaming channel is used to transfer control of the PDM data block between execution contexts (i.e. thread->thread or ISR->thread).

The mic array unit receives blocks of PDM data from an instance of this class by calling `GetPdmBlock()`, which blocks until a new PDM block is available.

`StandardPdmRxService` collects blocks of PDM samples from a port and makes them available to the decimation thread as the blocks are completed.

This class provides the logic for aggregating PDM data taken from a port into blocks, and provides methods `ReadPort()`, `SendBlock()` and `GetPdmBlock()`.

`ReadPort()` is responsible for reading 1 word of data from `p_pdm_mics`.

`SendBlock()` is provided a block of PDM data as a pointer and is responsible for signaling that to the subsequent processing stage.

`ReadPort()` and `SendBlock()` are used by `StandardPdmRxService` itself (when running as a thread, rather than ISR).

`GetPdmBlock()` is responsible for receiving a block of PDM data from `SendBlock()` as a pointer, deinterleaving the buffer contents, and returning a pointer to the PDM data in the format expected by the mic array unit's decimator component. See

`GetPdmBlock()` is called by the decimation thread. The pair of functions, `SendBlock()` and `GetPdmBlock()` facilitate inter-thread communication, `SendBlock()` being called by the transmitting end of the communication channel, and `GetPdmBlock()` being called by the receiving end.

Layouts

The buffer transferred by `SendBlock()` contains `CHANNELS_IN * this->pdm_out_words_per_channel` words of PDM data for `CHANNELS_IN` microphone channels. The words are stored in reverse order of arrival. See `mic_array::deinterleave_pdm_samples()` for additional details on this format.

Within `GetPdmBlock()` (i.e. mic array thread) the PDM data block is deinterleaved and copied to another buffer in the format required by the decimator component, which is returned by `GetPdmBlock()`. This buffer contains `CHANNELS_OUT * this->pdm_out_words_per_channel` words for `CHANNELS_OUT` microphone channels.



Channel Filtering

In some cases an application may be required to capture more microphone channels than should actually be processed by subsequent processing stages (including the decimator component). For example, this may be the case if 4 microphone channels are desired but only an 8 bit wide port is physically available to capture the samples.

This class template has a parameter both for the number of channels to be captured by the port (`CHANNELS_IN`), as well as for the number of channels that are to be output for consumption by the `MicArray`'s decimator component (`CHANNELS_OUT`).

When the PDM microphones are in an SDR configuration, `CHANNELS_IN` must be the width (in bits) of the XCore port to which the microphones are physically connected. When in a DDR configuration, `CHANNELS_IN` must



be twice the width (in bits) of the XCore port to which the microphones are physically connected.

CHANNELS_OUT is the number of microphone channels to be consumed by the mic array's decimator component (i.e. must be the same as the **MIC_COUNT** template parameter of the decimator component). If all port pins are connected to microphones, this parameter will generally be the same as **CHANNELS_IN**.

Channel Index (Re-)Mapping

The input channel index of a microphone depends on the pin to which it is connected. Each pin connected to a port has a bit index for that port, given in the 'Signal Description and GPIO' section of your package's datasheet.

Suppose an **N**-bit port is used to capture microphone data, and a microphone is connected to bit **B** of that port. In an SDR microphone configuration, the input channel index of that microphone is **B**, the same as the port bit index.

In a DDR configuration, that microphone will be on either input channel index **B** or **B+N**, depending on whether that microphone is configured for in-phase capture or out-of-phase capture.

Sometimes it may be desirable to re-order the microphone channel indices. This is likely the case, for example, when **CHANNELS_IN** > **CHANNELS_OUT**.

By default output channels are mapped from the input channels with the same index. If **CHANNELS_IN** > **CHANNELS_OUT**, this means that the input channels with the highest **CHANNELS_IN-CHANNELS_OUT** indices are dropped by default.

The [*MapChannel\(\)*](#) and [*MapChannels\(\)*](#) methods can be used to specify a non-default mapping from input channel indices to output channel indices. It takes a pointer to a **CHANNELS_OUT**-element array specifying the input channel index for each output channel.

Template Parameters

- ▶ **CHANNELS_IN** – The number of microphone channels to be captured by the port. For example, if using a 4-bit port to capture 6 microphone channels in a DDR configuration (because there are no 3 or 6 pin ports) **CHANNELS_IN** should be 8, because that's how many must be captured, even if two of them are stripped out before passing audio frames to subsequent application stages.
- ▶ **CHANNELS_OUT** – The number of output microphone channels to be delivered by this [*StandardPdmRxService*](#) instance.

Public Functions

uint32_t **ReadPort**()

Read a word of PDM data from the port.

Returns

A **uint32_t** containing 32 PDM samples. If **MIC_COUNT** >= 2 the samples from each port will be interleaved together.

void **SendBlock**(uint32_t *block)

Send a block of PDM data to a listener.

Parameters

block – PDM data to send.



void **Init**(port_t p_pdm_mics, pdm_rx_conf_t &pdm_rx_config)

Initialize the PDM RX service.

Sets the input port and binds application-provided buffers from `pdm_rx_conf_t` `pdm_rx_config`.

Requirements:

- ▶ `pdm_rx_config.pdm_in_double_buf` must be sized $2 * \text{CHANNELS_IN} * \text{pdm_rx_config.pdm_out_words_per_channel}$ words and remain valid for the lifetime of the service.
- ▶ `pdm_rx_config.pdm_out_block` must be sized $\text{CHANNELS_OUT} * \text{pdm_rx_config.pdm_out_words_per_channel}$ words and remain valid for the lifetime of the service.

Parameters

- ▶ `p_pdm_mics` – Port from which PDM samples are captured.
- ▶ `pdm_rx_config` – PDM RX configuration

void **MapChannels**(const unsigned map[`CHANNELS_OUT`])

Set the input-output mapping for all output channels.

By default, input channel index `k` maps to output channel index `k`.

This method overrides that behavior for all channels, re-mapping each output channel such that output channel `k` is derived from input channel `map[k]`.

Note

Changing the channel mapping while the mic array unit is running is not recommended.

Parameters

- `map` – Array containing new channel map.

void **MapChannel**(unsigned out_channel, unsigned in_channel)

Set the input-output mapping for a single output channel.

By default, input channel index `k` maps to output channel index `k`.

This method overrides that behavior for a single output channel, configuring output channel `out_channel` to be derived from input channel `in_channel`.

Note

Changing the channel mapping while the mic array unit is running is not recommended.

Parameters

- ▶ `out_channel` – Output channel index to be re-mapped.
- ▶ `in_channel` – New source channel index for `out_channel`.

void **InstallISR**()

Install ISR for PDM reception on the current core.

Note

This does not unmask interrupts.



void **UnmaskISR**()

Unmask interrupts on the current core.

uint32_t ***GetPdmBlock**()

Get a block of PDM data.

Because blocks of PDM samples are delivered by pointer, the caller must either copy the samples or finish processing them before the next block of samples is ready, or the data will be clobbered.

Note

This is a blocking call.

Returns

Pointer to block of PDM data.

void **AssertOnDroppedBlock**(bool doAssert)

Set whether dropped PDM samples should cause an assertion.

If **doAssert** is set to **true** (default), the PDM rx ISR will raise an exception (**ET_CALL**) if it is ready to deliver a PDM block to the mic array thread when the mic array thread is not ready to receive it. If **false**, dropped blocks can be tracked through **pdm_rx_isr_context.missed_blocks**.

void **SetPort**(port_t p_pdm_mics)

Set the port from which to collect PDM samples.

void **ThreadEntry**()

Entry point for PDM processing thread.

This function loops forever, performing a port read and if a new block has completed, signal a block send, every iteration.



TwoStageDecimator

template<unsigned **MIC_COUNT**>

class **TwoStageDecimator**

First and Second Stage Decimator.

This class template represents a two stage decimator which converts a stream of PDM samples to a lower sample rate stream of PCM samples.

Concrete implementations of this class template are meant to be used as the **TDecimator** template parameter in the [MicArray](#) class template.

Template Parameters

MIC_COUNT – Number of microphone channels.

Public Functions

void **Init**(mic_array_decimator_conf_t &decimator_conf)

Initialize the two-stage decimator from a configuration struct [mic_array_decimator_conf_t](#) **decimator_conf**.

Reads stage-1 and stage-2 filter parameters from **decimator_conf** and prepares internal state: The caller must ensure all pointers inside **decimator_conf.filter_conf[0]** and **decimator_conf.filter_conf[0]** are valid and persist for the lifetime of the decimator.

Parameters

decimator_conf – Decimator pipeline configuration.

void **ProcessBlock**(int32_t sample_out[**MIC_COUNT**], uint32_t *pdm_block)

Process one block of PDM data.

Processes a block of PDM data to produce an output sample from the second stage decimator.

pdm_block contains exactly enough PDM samples to produce a single output sample from the second stage decimator. The layout of **pdm_block** should (effectively) be:

```
struct {
    struct {
        // lower word indices are older samples.
        // less significant bits in a word are older samples.
        uint32_t samples[S2_DEC_FACTOR];
    } microphone[MIC_COUNT]; // mic channels are in ascending order
} pdm_block;
```

A single output sample from the second stage decimator is computed and written to **sample_out[]**.

Parameters

- ▶ **sample_out** – Output sample vector.
- ▶ **pdm_block** – PDM data to be processed.

Public Members

const uint32_t ***filter_coef**

Pointer to filter coefficients for Stage 1

uint32_t ***pdm_history_ptr**

Pointer to filter state (PDM history) for stage-1 filter.



unsigned **pdm_history_sz**

Per-mic channel filter state (PDM history) size in 32-bit words for stage-1 filter.

filter_fir_s32_t **filters**[*MIC_COUNT*]

Stage 2 FIR filters

unsigned **decimation_factor**

Stage 2 filter decimation factor.



SampleFilter

NopSampleFilter

template<unsigned **MIC_COUNT**>

class **NopSampleFilter**

SampleFilter which does nothing.

To be used as the **TSampleFilter** template parameter of *MicArray* when no post-decimation filtering is desired.

Calls to **NopSampleFilter::Filter()** are intended to be optimized out at compile time.

Template Parameters

MIC_COUNT – Number of microphone channels.

Public Functions

inline void **Filter**(int32_t sample[**MIC_COUNT**])

Do nothing.

DcoeSampleFilter

template<unsigned **MIC_COUNT**>

class **DcoeSampleFilter**

Filter which applies DC Offset Elimination (DCOE).

To be used as the **TSampleFilter** template parameter of *MicArray* when DCOE is desired as post-processing after the decimation filter.

The filter is a simple first-order IIR filter which applies the following filter equation:

```
R = 252.0 / 256.0
y[t] = R * y[t-1] + x[t] - x[t-1]
```

Template Parameters

MIC_COUNT – Number of microphone channels.

Public Functions

void **Init**()

Initialize the filter states.

The filter states must be initialized prior to calls to **Filter()**.

void **Filter**(int32_t sample[**MIC_COUNT**])

Apply DCOE filter on samples.

sample is an array of samples to be filtered, and is updated in-place.

The filter states must have been initialized with a call to **Init()** prior to calling this function.

Parameters

sample – Samples to be filtered. Updated in-place.



OutputHandler

An `OutputHandler` is a class which meets the requirements to be used as the `TOutputHandler` template parameter of the `MicArray` class template. The basic requirement is that it have a method:

This method is how the mic array communicates its output with the rest of the application's audio processing pipeline. `MicArray` calls this method once for each mic array output sample.

See `MicArray::OutputHandler` for more details.

FrameOutputHandler

template<unsigned `MIC_COUNT`, unsigned `SAMPLE_COUNT`, template<unsigned, unsigned> class `FrameTransmitter`, unsigned `FRAME_COUNT` = 1>

class `FrameOutputHandler`

`OutputHandler` implementation which groups samples into non-overlapping multi-sample audio frames and sends entire frames to subsequent processing stages.

This class template can be used as an `OutputHandler` with the `MicArray` class template. See `MicArray::OutputHandler`.

Classes derived from this template collect samples into frames. A frame is a 2 dimensional array with one index corresponding to the audio channel and the other index corresponding to time step, e.g.:

```
int32_t frame[MIC_COUNT][SAMPLE_COUNT];
```

Each call to `OutputSample()` adds the sample to the current frame, and then iff the frame is full, uses its `FrameTx` component to transfer the frame of audio to subsequent processing stages. Only one of every `SAMPLE_COUNT` calls to `OutputSample()` results in an actual transmission to subsequent stages.

With `FrameOutputHandler`, the thread receiving the audio will generally need to know how many microphone channels and how many samples to expect per frame (although, strictly speaking, that depends upon the chosen `FrameTransmitter` implementation).

Template Parameters

- ▶ **MIC_COUNT** – The number of audio channels in each sample and each frame.
- ▶ **SAMPLE_COUNT** – Number of samples per frame. The `SAMPLE_COUNT` template parameter is the number of samples assembled into each audio frame. Only completed frames are transmitted to subsequent processing stages. A `SAMPLE_COUNT` value of 1 effectively disables framing, transmitting one sample for each call made to `OutputSample`.
- ▶ **FrameTransmitter** – The concrete type of the `FrameTx` component of this class.
- ▶ **FRAME_COUNT** – The number of frame buffers an instance of `FrameOutputHandler` should cycle through. Unless audio frames are communicated with subsequent processing stages through shared memory, the default value of 1 is usually ideal.

Public Functions

inline `FrameOutputHandler()`

Construct new `FrameOutputHandler`.



The default no-argument constructor for **FrameTransmitter** is used to create **FrameTx**.

```
inline FrameOutputHandler(
    FrameTransmitter<MIC_COUNT, SAMPLE_COUNT> frame_tx,
)
```

Construct new *FrameOutputHandler*.

Uses the provided *FrameTransmitter* to send frames.

Parameters

frame_tx – Frame transmitter for sending frames.

```
bool OutputSample(int32_t sample[MIC_COUNT])
```

Add new sample to current frame and output frame if filled.

Parameters

sample – Sample to be added to current frame.

```
void CompleteShutdown()
```

Complete mic array shutdown process.

Public Members

FrameTransmitter<**MIC_COUNT**, **SAMPLE_COUNT**> **FrameTx**

FrameTransmitter used to transmit frames to the next stage for processing.

FrameTransmitter is the template, template parameter used in this class to control how frames of audio data are communicated with subsequent pipeline stages.

The type supplied for **FrameTransmitter** must be a class template with two integer template parameters, corresponding to this class's **MIC_COUNT** and **SAMPLE_COUNT** template parameters respectively, indicating the shape of the frame object to be transmitted.

The **FrameTransmitter** type is required to implement a single method:

```
void OutputFrame(int32_t frame[MIC_COUNT][SAMPLE_COUNT]);
```

OutputFrame() is called once for each completed audio frame and is responsible for the details of how the frame's data gets communicated to subsequent stages. For example, the *ChannelFrameTransmitter* class template uses an XCore channel to send samples to another thread (by value).

Alternative implementations might use shared memory or an RTOS queue to transmit the frame data, or might even use a port to signal the samples directly to an external DAC.

ChannelFrameTransmitter

template<unsigned **MIC_COUNT**, unsigned **SAMPLE_COUNT**>

class **ChannelFrameTransmitter**

Frame transmitter which transmits frame over a channel.

This class template is meant for use as the **FrameTransmitter** template parameter of *FrameOutputHandler*.

When using this frame transmitter, frames are transmitted over a channel using the frame transfer API in **mic_array/frame_transfer.h**. Usually, a call to *ma_frame_rx()* (with the other end of *c_frame_out* as argument) should be used to receive the frame on another thread.

If the receiving thread is not waiting to receive the frame when *OutputFrame()* is called, that method will block until the frame has been transmitted. In order to



ensure there are no violations of the mic array's real-time constraints, the receiver should be ready to receive a frame as soon as it becomes available.

Frames can be transmitted between tiles using this class.

Note

While `OutputFrame()` is blocking, it will not prevent the PDM rx interrupt from firing.

Template Parameters

- ▶ **MIC_COUNT** – Number of audio channels in each frame.
- ▶ **SAMPLE_COUNT** – Number of samples per frame.

Public Functions

inline **ChannelFrameTransmitter**()

Construct a `ChannelFrameTransmitter`.

If this constructor is used, `SetChannel()` must be called to configure the channel over which frames are transmitted prior to any calls to `OutputFrame()`.

inline **ChannelFrameTransmitter**(chanend_t c_frame_out)

Construct a `ChannelFrameTransmitter`.

The supplied value of `c_frame_out` must be a valid chanend.

Parameters

c_frame_out – Chanend over which frames will be transmitted.

void **SetChannel**(chanend_t c_frame_out)

Set channel used for frame transfers.

The supplied value of `c_frame_out` must be a valid chanend.

Parameters

c_frame_out – Chanend over which frames will be transmitted.

chanend_t **GetChannel**()

Get the chanend used for frame transfers.

Returns

Channel to be used for frame transfers.

bool **OutputFrame**(int32_t frame[MIC_COUNT][SAMPLE_COUNT])

Transmit the specified frame.

See `ChannelFrameTransmitter` for additional details.

Parameters

frame – Frame to be transmitted.

void **CompleteShutdown**()

Complete mic array shutdown process by exchanging end tokens with the app. This causes `ma_shutdown()` to return indicating mic array shutdown completion.



Misc

```
template<unsigned MIC_COUNT>
void mic_array::deinterleave_pdm_samples(
    uint32_t *samples, unsigned s2_dec_factor,
)
```

Deinterleave the channels of a block of PDM data.

PDM samples received on a port are shifted into a 32-bit buffer in such a way that the samples for each microphone channel are all interleaved with one another. The first stage decimator, however, requires these to be separated.

samples must point to a buffer containing (**MIC_COUNT*s2_dec_factor**) words of PDM data. Because the decimation factor for the first stage decimator is a fixed value of 32, 32 PDM samples from each microphone is enough to produce one output sample (a **MIC_COUNT**-element vector) from the first stage decimator. **32*s2_dec_factor** PDM samples for each of the **MIC_COUNT** microphone channels is then exactly what is required to produce a single output sample from the second stage decimator.

The PDM data will be deinterleaved in-place.

On input, the format of the buffer to which **samples** points is assumed to be such that the following function will extract (only) the **k**th sample for microphone channel **n** (where **k** is a time index, not a memory index):

Input Format

```
unsigned get_sample(uint32_t* samples,
                   unsigned MIC_COUNT, unsigned s2_dec_factor,
                   unsigned n, unsigned k)
{
    const end_word = MIC_COUNT * s2_dec_factor - 1; // chronologically first
    const unsigned samp_per_word = 32 / MIC_COUNT;
    const words_from_end = k / samp_per_word;
    const uint32_t word_val = samples[end_word-words_from_end];
    const unsigned bit_offset = (k % end_word) + n;
    return (word_val >> bit_offset) & 1;
}
```

Here, the words of **samples** are stored in reverse order (older samples are at higher word indices), and within a word the oldest samples are the least significant bits. The LSb of a word is always microphone channel 0, and the MSb of a word is always microphone channel **MIC_COUNT-1**.

Upon return, the format of the buffer to which **samples** points will be such that the following function will extract (only) the **k**th sample for microphone channel **n**:

Output Format

```
unsigned get_sample(uint32_t* samples,
                   unsigned MIC_COUNT, unsigned s2_dec_factor,
                   unsigned n, unsigned k)
{
    const unsigned subblock = (s2_dec_factor-1)-(k/32);
    const unsigned word_val = samples[subblock * MIC_COUNT + n];
    return (word_val >> (k%32)) & 1;
}
```

Here, each word contains samples from only a single channel, with words at higher addresses containing older samples. **samples[0]** contains the newest samples for microphone channel 0, and **samples[MIC_COUNT-1]** contains the newest samples for microphone channel **MIC_COUNT-1**. **samples[MIC_COUNT]** contains the next-oldest set of samples for channel 0, and so on.



Template Parameters

MIC_COUNT – Number of channels represented in PDM data. One of {1, 2, 4, 8}

Parameters

- ▶ **samples** – Pointer to block of PDM samples.
- ▶ **s2_dec_factor** – Stage2 decimator decimation factor.

11.2 C API Reference**mic_array_conf_struct.h**

struct **mic_array_filter_conf_t**

Configuration for a single decimator stage (e.g., stage 1 or stage 2).

All memory referenced by this structure is owned by the caller. The caller must allocate and initialize all buffers (coefficients and state) and ensure that all pointers remain valid for the entire lifetime of the decimator.

Public Members

int32_t ***coef**

Pointer to the filter coefficient array.
This buffer must be aligned to a 32-bit word boundary.

int32_t ***state**

Pointer to filter state (delay line) buffer for all microphones in int32_t words.
For multi-mic use this typically points to a buffer of size **output_mic_count** * @ref [mic_array_filter_conf_t::num_taps](#) 32-bit words.
This buffer must be aligned to a 32-bit word boundary.

right_shift_t **shr**

Final right-shift applied to the filter's accumulator prior to output.
See lib_xcore_math xmath/filter.h (filter_fir_s32_t::shift) for details.

unsigned **num_taps**

Number of FIR taps in this filter stage.

unsigned **decimation_factor**

Decimation ratio (downsampling factor) applied by this filter stage.

unsigned **state_words_per_channel**

Per-microphone state buffer size in int32_t words.
Used to index state for each mic: state[mic * state_words_per_channel ...].

struct **mic_array_decimator_conf_t**

Configuration for the full decimator pipeline.

Contains an array of stage configurations in execution order.

- ▶ filter_conf[0] is the first stage (PDM front-end).
- ▶ filter_conf[1] is the second stage (PCM FIR decimator).



Public Members

mic_array_filter_conf_t ***filter_conf**

Array of filter stage configurations (ordered in filter execution order)

unsigned **num_filter_stages**

Number of filter stages in the decimation pipeline.

Specifies the length of **filter_conf**. For a typical two-stage decimator, set this to 2 with **filter_conf**[0] as stage 1 and **filter_conf**[1] as stage 2. Must be ≥ 1 .

struct **pdm_rx_conf_t**

Configuration for the PDM RX service.

Public Members

uint32_t ***pdm_out_block**

PDM RX output block (input to the decimator) for all microphones.

Packed PDM samples as 32-bit words. The layout is a contiguous buffer sized **output_mic_count** * **pdm_out_words_per_channel** 32-bit words, arranged as [output_mic_count][pdm_out_words_per_channel]. In the **pdm_out_words_per_channel** 32-bit words, lower indexed words represent older samples. Within a 32-bit word the less significant bits are older samples. Typically contains enough PDM words to produce one PCM sample per microphone after decimation.

This buffer must be aligned to a 32-bit word boundary.

uint32_t ***pdm_in_double_buf**

PDM input block (double buffered) for all microphones.

Packed PDM input samples as 32-bit words. The layout is a contiguous buffer of size $2 * \text{input_mic_count} * \text{pdm_out_words_per_channel}$ 32-bit words, arranged as [2][input_mic_count][pdm_out_words_per_channel]. The buffer is double buffered such that one buffer is processed by the decimator while the other is filled by the PDM RX service.

This buffer must be aligned to an 8-byte boundary, as required by the deinterleave functions.

const unsigned ***channel_map**

Array mapping **output_mic_count** outputs to input microphone indices.

Array dimension must be **output_mic_count**. The i^{th} entry gives the input mic index mapped to output mic index i .

unsigned **pdm_out_words_per_channel**

Number of 32-pdm_sample subblocks to be captured for each microphone channel by the PDM RX service.

This is the number of words required to produce one PCM sample at the output of the decimator and is sized depending on the decimator configuration. It is typically equal to the 2^{nd} stage decimation filter's decimation factor (in case of a 2 stage decimator).



struct `mic_array_conf_t`

Top-level configuration passed to `mic_array_init_custom_filter()`.

Bundles the decimator pipeline configuration and the PDM RX configuration. All buffers referenced by pointers inside these sub-configs must be allocated and owned by the caller and must remain valid for the lifetime of the mic array task (until `mic_array_start()` returns).

Note

- When using `mic_array_init_custom_filter()`, both members must be populated.

Public Members

`mic_array_decimator_conf_t` **decimator_conf**

decimator configuration

`pdm_rx_conf_t` **pdmrx_conf**

PDM RX service configuration

Using the `mic_array` - default model or with custom filters

Configuration defines (`mic_array_conf_default.h`) An application using the mic array needs to have defines set for compile-time configuration of the mic array instance. Defaults for these defines are defined in the header file `mic_array_conf_default.h`. These defines should be overridden in an optional header file `mic_array_conf.h` file or in the application's `CMakeLists.txt`.

This section fully documents all of the settable defines and their default values.

`MIC_ARRAY_CONFIG_MIC_COUNT`

Number of PDM microphone channels. Default: 2.

`MIC_ARRAY_CONFIG_MIC_IN_COUNT`

Number of input mic channels - This is the width of the pdm port. Default: `MIC_ARRAY_CONFIG_MIC_COUNT`.

`MIC_ARRAY_CONFIG_USE_PDM_ISR`

Use interrupt-driven PDM capture (1 = ISR, 0 = polling/task). Default: 1 -> Use ISR.

`MIC_ARRAY_CONFIG_SAMPLES_PER_FRAME`

PCM samples per frame emitted by the driver. Must be ≥ 1 . Default: 1.

`MIC_ARRAY_CONFIG_USE_DC_ELIMINATION`

Enable DC elimination on the PCM output (1 = enabled). Default: 1.



Function definitions (mic_array_task.h) The header file `mic_array_task.h` contains the function API declarations that the application needs to call to initialise and start a mic array instance when using the default model (`mic_array_init()` and `mic_array_start()`) or when using custom filter (`mic_array_init_custom_filter()` and `mic_array_start()`).

```
void mic_array_init(
    pdm_rx_resources_t *pdm_res, const unsigned *channel_map, unsigned output_samp_freq,
)
```

Initializes the mic array task.

Initializes the contexts for the decimator thread and configures the clocks and ports for PDM reception. Note that this does not start any threads or PDM capture.

After calling this, the PDM clock is active and signaling, but the PDM rx service has not been activated (when running in the interrupt context)/PDM rx thread is not created (when running in thread context) so the received PDM samples are ignored.

Parameters

- ▶ **pdm_res** – Pointer to the `pdm_rx_resources_t` struct containing hardware resources required by the mic array module.
- ▶ **channel_map** – Optional mapping from PDM input pin to mic-array output channel index.
 - ▶ Array size: `MIC_ARRAY_CONFIG_MIC_COUNT`
 - ▶ `channel_map[i]` is the PDM pin index that should feed mic output channel `i`.
 - ▶ If `channel_map` is NULL, a default 1:1 mapping is used: PDM pin `i` -> mic output channel `i`.
 - ▶ Valid values for `channel_map[i]` are in `[0, MIC_ARRAY_CONFIG_MIC_IN_COUNT-1]`.
- ▶ **output_samp_freq** – Target sampling rate (in Hz) for the decimated PCM output stream. Supported values: 16000, 32000, 48000 (Hz).

```
void mic_array_start(chanend_t c_frames_out)
```

Start the mic array task.

This function sets up and activates the PDM rx service in ISR mode (when `MIC_ARRAY_CONFIG_USE_PDM_ISR=1`), and then immediately begins executing the decimator. When `MIC_ARRAY_CONFIG_USE_PDM_ISR=0`, it starts the PDM rx service and decimator as 2 parallel threads.

After calling this the real-time condition is active, meaning there must be another thread waiting to pull frames from the other end of `c_frames_out` as they become available.

Parameters

- ▶ **c_frames_out** – (Non-streaming) Channel over which to send processed frames of audio.

```
void mic_array_init_custom_filter(
    pdm_rx_resources_t *pdm_res, mic_array_conf_t *mic_array_conf,
)
```

Initialize the mic array using application-supplied decimation filter and PDM RX buffers.

All memory referenced by `mic_array_conf` must be allocated and owned by the caller and must remain valid for the lifetime of the mic array task, i.e., until



mic_array_start returns. This excludes the [mic_array_conf_t](#) structure itself and the [mic_array_decimator_conf_t::filter_conf](#) array, which may reside on the caller's stack.

After successful initialization, the PDM clock is configured and started, but no threads/ISRs are running until mic_array_start() is called.

Note

The caller owns the buffers referenced by **mic_array_conf** and must ensure correct sizing and required alignment of these buffers.

Parameters

- ▶ **pdm_res** – Pointer to hardware resources used by the mic array module (ports, clocks, and chanends).
- ▶ **mic_array_conf** – Pointer to top-level configuration that bundles the decimator pipeline configuration ([mic_array_decimator_conf_t](#)) and the PDM RX configuration ([pdm_rx_conf_t](#)).

filters_default.h

The filters described below are the first and second stage filters provided by this library which are used with the [TwoStageDecimator](#) class template by default.

Stage 1 - PDM-to-PCM Decimating FIR Filter

Decimation Factor: 32
Tap Count: 256

The first stage decimation FIR filter converts 1-bit PDM samples into 32-bit PCM samples and simultaneously decimates by a factor of 32.

A typical input PDM sample rate will be 3.072M samples/sec, thus the corresponding output sample rate will be 96k samples/sec.

The first stage filter uses 16-bit coefficients for its taps. Because this is a highly optimized filter targeting the VPU hardware, the first stage filter is presently restricted to using exactly 256 filter taps.

For more information about the example first stage filter supplied with the library, including frequency response and steps for using a custom first stage filter, see [Decimation filters](#).

STAGE1_DEC_FACTOR

Macro indicating Stage 1 Decimation Factor.

This is the ratio of input sample rate to output sample rate for the first filter stage. This value is fixed, even when using custom filter coefficients.

STAGE1_TAP_COUNT

Macro indicating Stage 1 Filter Tap Count.

This is the number of filter taps in the first stage filter. This value is fixed, even when using custom filter coefficients.



STAGE1_WORDS

Macro indicating Stage 1 Filter Word Count.

This is a helper macro to indicate the number of 32-bit words required to store the filter coefficients.

Note

Even though the coefficients are 16-bit, the related lib_mic_array structs and functions expect them to be contained in an array of `uint32_t`, rather than an array of `int16_t`. There are two reasons for this. The first is that the VPU instructions require loaded data to start at a word-aligned (0 mod 4) address. `uint32_t` allocated on the heap or stack are guaranteed by the compiler to be at word-aligned addresses. The second reason is to mitigate possible confusion regarding the arrangement of the filter coefficients in memory. Not only are the 16-bit coefficients not stored in order (e.g. `b[0]`, `b[1]`, `b[2]`, ...), the bits of individual 16-bit coefficients are not stored together in memory. This is, again, due to the behavior of the VPU hardware.

`uint32_t stage1_coef[STAGE1_WORDS]`

Stage 1 PDM-to-PCM Decimation Filter Default Coefficients.

These are the default coefficients for the first stage filter.

Stage 2 - PCM Decimating FIR Filter

Decimation Factor: (configurable)
Tap Count: (configurable)

The second stage decimation FIR filter filters and downsamples the 32-bit PCM output stream from the first stage filter into another 32-bit PCM stream with sample rate reduced by the stage 2 decimation factor.

A typical first stage output sample rate will be 96k samples/sec, a decimation factor of 6 (i.e. using the default stage 2 filter) will mean a second stage output sample rate of 16k samples/sec.

The second stage filter uses 32-bit coefficients for its taps. A complete description of the FIR implementation is outside the scope of this documentation, but it can be found in the `xs3_filter_fir_s32_t` documentation of `lib_xcore_math`.

In brief, the second stage filter coefficients are quantized to a Q1.30 fixed-point format with input samples treated as integers. The tap outputs are added into a 40-bit accumulator, and an output sample is produced by applying a rounding arithmetic right-shift to the accumulator and then clipping the result to the interval `[INT32_MAX, INT32_MIN]`.

For more information about the example second stage filter supplies with the library, including frequency response and steps for using a custom filter, see [Decimation filters](#).

STAGE2_DEC_FACTOR

Stage 2 Decimation Factor for default filter.

This is the ratio of input sample rate to output sample rate for the second filter stage.

While the second stage filter can be configured with a different decimation factor, this is the one used for the filter supplied with this library.



STAGE2_TAP_COUNT

Stage 2 Filter tap count for default filter.

This is the number of filter taps associated with the second stage filter supplied with this library.

int32_t **stage2_coef**[*STAGE2_TAP_COUNT*]

Stage 2 Decimation Filter Default Coefficients.

These are the default coefficients for the second stage filter.

right_shift_t **stage2_shr**

Stage 2 Decimation Filter Default Output Shift.

This is the non-negative, rounding, arithmetic right-shift applied to the 40-bit accumulator to produce an output sample.

pdm_resources.h

struct **pdm_rx_resources_t**

Collection of resources IDs required for PDM capture.

This struct is a container for the IDs of the XCore hardware resources used by the mic array unit's PdmRx component for capturing PDM data from a port.

An object of this type will be used for initializing and starting the mic array unit.

Public Members

port_t **p_mclk**

Resource ID of the 1-bit port on which the master audio clock signal is received.

The master audio clock will be divided by a clock block to produce the PDM sample clock.

This port will be configured as an input.

port_t **p_pdm_clk**

Resource ID of the 1-bit port through which the PDM sample clock is signaled.

The PDM sample clock is used by the PDM microphones to trigger sample conversion.

This port will be configured as an output.

port_t **p_pdm_mics**

Resource ID of the port on which PDM samples are received.

In an SDR configuration, the number of microphone channels is the width of this port. In a DDR configuration, the number of microphone channels is twice the width of this port.

This port will be configured as an input.

unsigned **mclk_freq**

Master clock frequency.

Application's master audio clock frequency in Hz. A typical value is 24576000 Hz



unsigned **pdm_freq**

PDM clock frequency.

Frequency in Hz of the physical PDM clock driving the microphones. This is the actual bit clock output to the PDM mics, not the PDM capture clock, which may differ when using DDR capture mode. Must be an integer factor of the MCLK frequency.

clock_t **clock_a**

used with mclk freq to calculate clock divider.

Resource ID of the clock block used to derive the PDM clock from the master audio clock.

In SDR configurations this is also the PDM data capture clock.

clock_t **clock_b**

Resource ID of the clock block used only in DDR configurations to trigger reads of the PDM data.

If operating in an SDR configuration, **clock_b** is 0. A value of 0 indicates an SDR configuration is being used.

PDM_RX_RESOURCES_SDR(P_MCLK, P_PDM_CLK, P_PDM_MICS, MCLK_FREQ, PDM_FREQ, CLOCK_A)

Construct a *pdm_rx_resources_t* for an SDR configuration.

pdm_rx_resources_t.clock_b is initialized to 0, indicating an SDR configuration.

Parameters

- ▶ **P_MCLK** – Master audio clock port resource ID.
- ▶ **P_PDM_CLK** – PDM sample clock port resource ID.
- ▶ **P_PDM_MICS** – PDM microphone data port resource ID.
- ▶ **MCLK_FREQ** – MCLK frequency
- ▶ **PDM_FREQ** – PDM frequency
- ▶ **CLOCK_A** – PDM clock and capture clock block resource ID.

PDM_RX_RESOURCES_DDR(P_MCLK, P_PDM_CLK, P_PDM_MICS, MCLK_FREQ, PDM_FREQ, CLOCK_A, CLOCK_B)

Construct a *pdm_rx_resources_t* for a DDR configuration.

Parameters

- ▶ **P_MCLK** – Master audio clock port resource ID.
- ▶ **P_PDM_CLK** – PDM sample clock port resource ID.
- ▶ **P_PDM_MICS** – PDM microphone data port resource ID.
- ▶ **MCLK_FREQ** – MCLK frequency
- ▶ **PDM_FREQ** – PDM frequency
- ▶ **CLOCK_A** – PDM clock clock block resource ID.
- ▶ **CLOCK_B** – PDM capture clock block resource ID.

setup.h

void **mic_array_resources_configure**(*pdm_rx_resources_t* *pdm_res, int divide)

Configure the hardware resources needed by the mic array.

Several hardware resources are needed to correctly run the mic array, including 3 ports and 1 or 2 clock blocks (depending on whether SDR or DDR mode is used). This function configures these resources for operation with the mic array.



The `pdm_rx_resources_t` struct is a container for identifying precisely these resources. All three ports are reset by this function; any existing port configuration will be clobbered.

The parameter `divide` is the ratio of the audio master clock to the desired PDM clock rate. For example, to generate a desired 3.072 MHz PDM clock from an audio master clock with frequency 24.576 MHz, a `divide` value of 8 is needed. Divide can also be calculated from the master and PDM clock frequencies using `mic_array_mclk_divider()`.

`pdm_res->p_mclk` is the resource ID for the 1-bit port on which the audio master clock is received. This function will enable this port and configure it as the source port for `pdm_res->clock_a` and for `pdm_res->clock_b` if operating in a DDR configuration.

`pdm_res->clock_a` is the resource ID for the first (in SDR configuration, the only) clock block required by the mic array. Clock A divides the audio master clock (by a factor of `divide`) to generate the PDM clock. This function enables it with the audio master clock as its source.

`pdm_res->p_pdm_clk` is the resource ID for the 1-bit port from which the PDM clock will be signaled to the microphones. This function enables it and configures Clock A as its source clock.

`pdm_res->clock_b` is the resource ID for a second clock block, which is only required by the mic array in a DDR configuration. In DDR mode, this function enables Clock B with the audio master clock as its source. The divider for Clock B is half of that for Clock A (so it runs at twice the frequency). In a DDR configuration Clock B is used as the PDM capture clock. In an SDR configuration, this field must be set to 0 (this is how SDR/DDR is determined).

`pdm_res->p_pdm_mics` is the resource ID for the port on which PDM data is received. This function enables it and configures it as a 32-bit buffered input. If operating in an SDR configuration, Clock A is used as the capture clock. If operating in a DDR configuration, Clock B is used as its capture clock.

This function only configures and does not start either Clock A or Clock B. A call to `mic_array_pdm_clock_start()` with `pdm_res` as the argument can be used to start the clock(s).

This function should be called during initialization, before any PDM data can be captured or processed.

Parameters

- ▶ `pdm_res` – The hardware resources used by the mic array.
- ▶ `divide` – The divider to generate the PDM clock from the master clock.

void `mic_array_pdm_clock_start(pdm_rx_resources_t *pdm_res)`

Start the PDM and capture clock(s).

This function starts Clock A, and if using a DDR configuration, Clock B.

`mic_array_resources_configure()` must have been called already to configure the resources indicated in `pdm_res`.

Clock A is the PDM clock. Starting Clock A will cause `pdm_res->p_pdm_clk` to begin strobing the PDM clock to the PDM microphones.

In an SDR configuration, Clock A is also the capture clock. In a DDR configuration, Clock B is the capture clock. In either case, the capture clock is also started, causing `pdm_res->p_pdm_mics` to begin storing PDM samples received on each period of the capture clock.

In DDR configuration, this function starts Clock B, waits for a rising edge, and then starts Clock A, ensuring that the rising edges of the two clocks are not in phase.



This function must be called prior to launching the decimator or PDM rx threads.

Warning

Once this function has been called, the port receiving PDM data will begin capturing samples. If the mic array unit is not started by the time the port buffer fills ((32/mic_count) sample times) samples will begin to be dropped.

Parameters

- **pdm_res** – The hardware resources used by the mic array.

```
static inline unsigned mic_array_mclk_divider(
    const unsigned master_clock_freq, const unsigned pdm_clock_freq,
)
```

Compute clock divider for PDM clock.

This is a convenience function which computes the required clock divider to derive a **pdm_clock_freq** Hz clock from a **master_clock_freq** Hz clock. This function is simple integer division.

Parameters

- **master_clock_freq** – The master audio clock frequency in Hz.
- **pdm_clock_freq** – The desired PDM clock frequency in Hz.

Returns

Required clock divider.

frame_transfer.h

```
unsigned ma_frame_tx(
    const chanend_t c_frame_out, const int32_t frame[], const unsigned channel_count, const unsigned sample_count,
)
```

Transmit 32-bit PCM frame over a channel.

This function transmits the 32-bit PCM frame **frame[]** over the channel **c_frame_out**.

This is a blocking call which will wait for a receiver to accept the data from the channel. Typically this will be accomplished with a call to **ma_frame_rx()** or **ma_frame_rx_transpose()**.

The receiver is not required to be on the same tile as the sender.

Note

Internally, a channel transaction is established to reduce the overhead of channel communication. Any custom functions are used to receive this frame in an application, they must wrap the channel reads in a (slave) channel transaction. See **xcore/channel_transaction.h**.



Warning

No protocol is used to ensure consistency between the frame layout of the transmitter and receiver. Disagreement about frame size will likely cause one side to block indefinitely. It is the responsibility of the application author to ensure consistency between transmitter and receiver.

Parameters

- ▶ **c_frame_out** – Channel over which to send frame.
- ▶ **frame** – Frame to be transmitted.
- ▶ **channel_count** – Number of channels represented in the frame.
- ▶ **sample_count** – Number of samples represented in the frame.

Returns

shutdown - 0 if no shutdown requested, 1 if shutdown requested

```
void ma_frame_rx(
    int32_t frame[], const chanend_t c_frame_in, const unsigned channel_count,
    const unsigned sample_count,
)
```

Receive 32-bit PCM frame over a channel.

This function receives a PCM frame over **c_frame_in**. Normally, the frame will have been transmitted using **ma_frame_tx()**. The received frame is stored in **frame[]**.

This is a blocking call which does not return until the frame has been fully received. The sender is not required to be on the same tile as the receiver.

Note

Internally, a channel transaction is established to reduce the overhead of channel communication. This function may only be used to receive the frame if the transmitter has wrapped the channel writes in a (master) channel transaction. See **xcore/channel_transaction.h**.

Warning

No protocol is used to ensure consistency between the frame layout of the transmitter and receiver. Disagreement about frame size will likely cause one side to block indefinitely. It is the responsibility of the application author to ensure consistency between transmitter and receiver.

Parameters

- ▶ **frame** – Buffer to store received frame.
- ▶ **c_frame_in** – Channel from which to receive frame.
- ▶ **channel_count** – Number of channels represented in the frame.
- ▶ **sample_count** – Number of samples represented in the frame.



```
void ma_frame_rx_transpose(
    int32_t frame[], const chanend_t c_frame_in, const unsigned channel_count,
    const unsigned sample_count,
)
```

Receive 32-bit PCM frame over a channel with transposed dimensions.

This function receives a PCM frame over **c_frame_in**. Normally, the frame will have been transmitted using **ma_frame_tx()**. The received frame is stored in **frame[]**.

Unlike **ma_frame_rx()**, this function reorders the frame elements as they are received. **ma_frame_tx()** always transmits the frame elements in memory order. This function swaps the channel and sample axes so that if the transmitter frame has shape (**CHANNEL**, **SAMPLE**), the caller's frame array will have shape (**SAMPLE**, **CHANNEL**).

This is a blocking call which does not return until the frame has been fully received. The sender is not required to be on the same tile as the receiver.

Note

Internally, a channel transaction is established to reduce the overhead of channel communication. This function may only be used to receive the frame if the transmitter has wrapped the channel writes in a (master) channel transaction. See **xcore/channel_transaction.h**.

Warning

No protocol is used to ensure consistency between the frame layout of the transmitter and receiver. Disagreement about frame size will likely cause one side to block indefinitely. It is the responsibility of the application author to ensure consistency between transmitter and receiver.

Parameters

- ▶ **frame** – Buffer to store received frame.
- ▶ **c_frame_in** – Channel from which to receive frame.
- ▶ **channel_count** – Number of channels represented in the frame.
- ▶ **sample_count** – Number of samples represented in the frame.

shutdown.h

```
void ma_shutdown(const chanend_t c_frame_in)
```

Shut down the mic array thread(s).

This function is called by the application to terminate the mic array threads. It causes the decimator (and PDM rx if running in thread context) thread(s) to exit. This is required if the mic array needs to be restarted with a different configuration (a different output sampling rate for instance). After shutdown, the mic array can be initialised and started with a different configuration.

Note

This function returns only after the mic array thread(s) have exited.



Warning

Ensure that `ma_frame_rx()` or `ma_frame_rx_transpose()` is not being called concurrently when calling `ma_shutdown()`.

Warning

The mic array threads check for shutdown only at PDM frame boundaries. If PDM data is not being received (for example, if MCLK is stopped), the threads will stall and fail to detect the shutdown request. Ensure that MCLK remains active during shutdown so that PDM frames continue to arrive.

Parameters

- **c_frame_in** – chanend used to receive audio frames from the mic array. This must be the same chanend passed to `ma_frame_rx()` or `ma_frame_rx_transpose()` when the application receives a frame from the mic array. Shutdown is signalled to the mic array over the same chanend.

dc_elimination.h

struct **dcoe_chan_state_t**

DC Offset Elimination (DCOE) State.

This is the required state information for a single channel to which the DC offset elimination filter is to be applied.

To apply the DC offset elimination filter to multiple channels simultaneously, an array of **dcoe_chan_state_t** should be used.

dcoe_state_init() is used once to initialize an array of state objects, and **dcoe_filter()** is used on each consecutive sample to apply the filter and get the resulting output sample.

DC offset elimination is an IIR filter. The state must persist between time steps.

Use in lib_mic_array

Typical users of `lib_mic_array` will not need to directly use this type or any functions which take it as a parameter.

The C++ class template `mic_array::DcoeSampleFilter`, if used in an application's mic array unit, will allocate, initialize and apply the DCOE filter automatically.

With default API

When using the default API, DCOE is enabled by default. To disable DCOE when using this API, add a preprocessor definition to the compiler flags, setting **MIC_ARRAY_CONFIG_USE_DC_ELIMINATION** to 0.



Public Members

int64_t **prev_y**

Previous output sample value.

void **dcoe_state_init**(*dcoe_chan_state_t* state[], const unsigned chan_count)

Initialize DCOE states.

The DC offset elimination state needs to be initialized before the filter can be applied. This function initializes it.

For correct behavior, the state vector **state** must persist between audio samples and is supplied with each call to **dcoe_filter()**.

Parameters

- ▶ **state** – [in] Array of *dcoe_chan_state_t* to be initialized.
- ▶ **chan_count** – [in] Number of elements in **state**.

void **dcoe_filter**(
 int32_t new_output[], *dcoe_chan_state_t* state[], int32_t new_input[], const unsigned chan_count,
)

Apply DCOE filter.

Applies the DC offset elimination filter to get a new output sample and updates the filter state.

For correct behavior, this function should be called once per sample (here “sample” refers to a vector-valued quantity containing one element for each audio channel) of that stream.

The index of each array (**state**, **new_input** and **new_output**) corresponds to the audio channel. The update associated with each audio channel is independent of each other audio channel.

The equation used for each channel is:

$$y[t] = R * y[t-1] + x[t] - x[t-1]$$

where **t** is the current sample time index, **y[]** is the output signal, **x[]** is the input signal, and **R** is (252.0/256).

To filter a sample in-place use the same array for both the **new_input** and **new_output** arguments.

Parameters

- ▶ **new_output** – [out] Array into which the output sample will be placed.
- ▶ **state** – [in] DC offset elimination state vector.
- ▶ **new_input** – [in] New input sample.
- ▶ **chan_count** – [in] Number of channels to be processed.

util.h

void **deinterleave2**(uint32_t*)

Perform deinterleaving for a 2-microphone subblock.

Assembly function.

Deinterleave the samples for 1 subblock of 2 microphones. Argument points to a 2 word buffer.



void **deinterleave4**(uint32_t*)

Perform deinterleaving for a 4-microphone subblock.

Assembly function.

Deinterleave the samples for 1 subblock of 4 microphones. Argument points to a 4 word buffer.

void **deinterleave8**(uint32_t*)

Perform deinterleaving for a 8-microphone subblock.

Assembly function.

Deinterleave the samples for 1 subblock of 8 microphones. Argument points to a 8 word buffer.

void **deinterleave16**(uint32_t*)

Perform deinterleaving for a 16-microphone subblock.

Assembly function.

Deinterleave the samples for 1 subblock of 16 microphones. Argument points to a 16 word buffer.



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.

