lib_mic_array: PDM microphone array library

Publication Date: 2024/10/30
Document Number: XM-010267-UG v5.5.0

XMOS

IN THIS DOCUMENT

1 Introduction

`lib_mic_array` is a library for interfacing with one or more PDM microphones on an XMOS device.

Version 5.0 of this library has been redesigned from scratch to make efficient usage of the XMOS XS3 architecture.

See *Getting Started* to get going.

Note: Version 5.0 does not currently 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

Find the latest version of `lib_mic_array` on GitHub.

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
  - Reference filter with total decimation factor of 192 is provided (16 kHz output sample rate w/ 3.072 MHz PDM clock).
  - Filter generation scripts and examples are included to support 32 kHz and 48 kHz.
- 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
- Optional DC offset elimination filter
- Sample framing with user selectable frame size (down to single samples)
- Most configurations require only a single hardware thread

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 behavior and does not describe any necessary initialization steps. The high level process view is depicted in the figure *Mic Array High Level Process*.

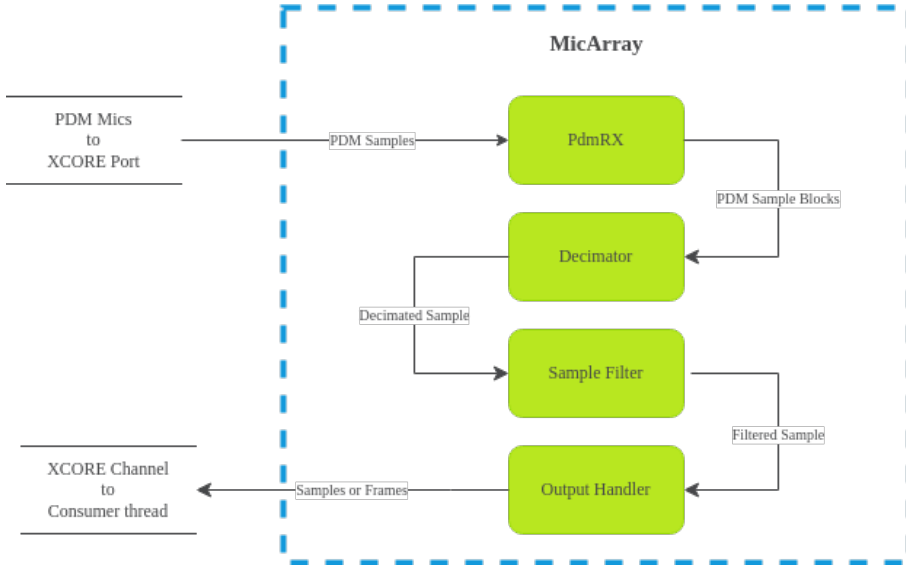

Fig. 1: Mic Array High Level Process

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 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 *Decimator Stages* 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 `65` 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, the output sample rate is 16 kHz.

See *Decimator Stages* 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 signaling 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 channel from the mic array module to the next stage of the processing pipeline.

Extending/Modifying Mic Array Behavior

At the core of `lib_mic_array` are several C++ class templates which are loosely coupled and intended to be easily overridden for modified behavior. The mic array unit itself is an object made by the composition of several smaller components which perform well-defined roles.

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 behavior, and then instantiating the mic array class template with the new class as the appropriate template parameter.

With that in mind, while most applications will have no need to modify the mic array behavior, it is nevertheless designed to be easy to do so.

3 Getting Started

There are three models for how the mic array unit can be included in an application. The details of how to allocate, initialize and start the mic array will depend on the chosen model.

In order of increasing complexity, these are:

- Vanilla Model - The simplest way to include the mic array. It is usually sufficient but offers comparatively little flexibility with respect to configuration and run-time control. Using this model (mostly) means modifying an application's build scripts.
- Prefab Model - This model involves a little more effort from the application developer, including writing a couple C++ wrapper functions, but gives the application access to any of the defined prefab mic array components.
- General Model - Any other case. This is necessary if an application wishes to use a customized mic array component.

The vanilla and prefab models for integrating the mic array into your application will be discussed in more detail below. The general model may involve customizing or extending the classes in `lib_mic_array` and is beyond the scope of this introduction.

Whichever model is chosen, the first step to integrate a mic array unit into an application is to *identify the required hardware resources*.

3.1 Identify 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. Your application is free to use any clock block that has not already been allocated for another purpose. The vanilla component model defaults to using `XS1_CLKBLK_1` and `XS1_CLKBLK_2`.

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 behavior.

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 your ports may have already been done for you in your 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"/>
  <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"/>
</Tile>
...
```

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 your package documentation.

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

Declaring 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_OUT,
                               PORT_PDM_CLK,
                               PORT_PDM_DATA,
                               XS1_CLKBLK_1,
                               XS1_CLKBLK_2);
```

Note that this is not necessary in applications using the vanilla model.

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.
- 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 *Mic Array Resource Usage*).
- Chanends - At least 4 chanends must be available for signaling between threads/subcomponents.

3.2 Vanilla Model

Mic array configuration with the vanilla model is achieved mostly through the application's build system configuration.

In the `/etc/vanilla` directory of the `lib_mic_array` repository are a source and header file which are not compiled with (or on the include path) of the library. Configuring the mic array using the vanilla model means adding those files to your *application*'s build (*not* the library target), and defining several compile options which tell it how to behave.

Vanilla - CMake Macro

To simplify this further, a CMake macro called `mic_array_vanilla_add()` has been included with the build system.

`mic_array_vanilla_add()` takes several arguments:

- `TARGET_NAME` - The name of the CMake application target that the vanilla mode source should be added to.
- `MCLK_FREQ` - The frequency of the master audio clock, in Hz.
- `PDM_FREQ` - The desired frequency of the PDM clock, in Hz.
- `MIC_COUNT` - The number of microphone channels to be captured.

- `SAMPLES_PER_FRAME` - The size of the audio frames produced by the mic array unit (frames will be 2 dimensional arrays with shape `(MIC_COUNT, SAMPLES_PER_FRAME)`).

Vanilla - Optional Configuration

Though not exposed by the `mic_array_vanilla_add()` macro, several additional configuration options are available when using the vanilla model. These are all configured by adding defines to the application target.

Vanilla - Initializing and Starting

Once the configuration options have been chosen, initializing and starting the mic array at run-time is easily achieved. Two function calls are necessary, both are included through `mic_array_vanilla.h` (which was added to your include path through your build configuration).

First, during application initialization, the function *`ma_vanilla_init()`*, which takes no arguments, must be called. This will configure the hardware resources and install the PDM rx service as an ISR, but will not actually start any threads or PDM capture.

Once any remaining application initialization is complete, PDM capture and processing is started by calling *`ma_vanilla_task()`*. `ma_vanilla_task()` is a blocking call which takes a single argument which is the chanend that will be used to transmit audio frames to subsequent stages of the processing pipeline. Usually the call to `ma_vanilla_task()` will be placed directly in a `par {...}` block along with other threads to be started on the tile.

Note: Both `ma_vanilla_init()` and `ma_vanilla_task()` must be called from the core which will host the decimation thread.

3.3 Prefab Model

The `lib_mic_array` library has a C++ namespace `mic_array::prefab` which contains class templates for typical mic array setups using common sub-components. The templates in the `mic_array::prefab` namespace hide most of the complexity (and unneeded flexibility) from the application author, so they can focus only on pieces they care about.

Note: As of version 5.0.1, only one prefab class template, *`BasicMicArray`*, has been defined.

To configure the mic array using a prefab, you will need to add a C++ source file to your application. NB: This will end up looking a lot like the contents of `mic_array_vanilla.cpp` when you are through.

Prefab - Declare Resources

The example in this section will use 2 microphones in a DDR configuration with DC offset elimination enabled, and using 128-sample frames. The resource IDs used may differ than those required for your application.

`pdm_res` will be used to identify the ports and clocks which will be configured for PDM capture.

Within a C++ source file:

```
#include "mic_array/mic_array.h"
...
#define MIC_COUNT    2    // 2 mics
#define DCOE_ENABLE  true // DCOE on
#define FRAME_SIZE   128  // 128 samples per frame
...
pdm_rx_resources_t pdm_res = PDM_RX_RESOURCES_DDR(
                               PORT_MCLK_IN_OUT,
                               PORT_PDM_CLK,
                               PORT_PDM_DATA,
                               MIC_ARRAY_CLK1,
                               MIC_ARRAY_CLK2);
...
```

Prefab - Allocate MicArray

The C++ class template *MicArray* is central to the mic array unit in this library. The class templates defined in the **mic_array::prefab** namespace each derive from **mic_array::MicArray**.

Define and allocate the specific implementation of **MicArray** to be used.

```
...
// Using the full name of the class could become cumbersome. Using an alias.
using TMicArray = mic_array::prefab::BasicMicArray<
                    MIC_COUNT, FRAME_SIZE, DCOE_ENABLED>
// Allocate mic array
TMicArray mics = TMicArray();
...
```

Now the mic array unit has been defined and allocated. The template parameters supplied (e.g. *MIC_COUNT* and *FRAME_SIZE*) are used to calculate the size of any data buffers required by the mic array, and so the **mics** object is self-contained, with all required buffers being statically allocated. Additionally, class templates will ultimately allow unused features to be optimized out at build time. For example, if DCOE is disabled, it will be optimized out at build time so that at run time it won't even need to check whether DCOE is enabled.

Prefab - Init and Start Functions

Now a couple functions need to be implemented in your C++ file. In most cases these functions will need to be callable from C or XC, and so they should not be static, and they should be decorated with **extern "C"** (or the **MA_C_API** preprocessor macro provided by the library).

First, a function which initializes the **MicArray** object and configures the port and clock block resources. The documentation for *BasicMicArray* indicates any parts of the **MicArray** object that need to be initialized.

```
#define MCLK_FREQ   24576000
#define PDM_FREQ    3072000
...
MA_C_API
void app_init() {
  // Configure clocks and ports
  const unsigned mclk_div = mic_array_mclk_divider(MCLK_FREQ, PDM_FREQ);
  mic_array_resources_configure(&pdm_res, mclk_div);

  // Initialize the PDM rx service
  mics.PdmRx.Init(pdm_res.p_pdm_mics);
}
...
```

app_init() can be called from an XC **main()** during initialization.

Assuming the PDM rx service is to be run as an ISR, a second function is used to actually start the mic array unit. This starts the PDM clock, install the ISR and enter the decimator thread's main loop.

```
MA_C_API
void app_mic_array_task(chanend_t c_audio_frames) {
  mics.SetOutputChannel(c_audio_frames);

  // Start the PDM clock
  mic_array_pdm_clock_start(&pdm_res);

  mics.InstallPdmRxISR();
  mics.UnmaskPdmRxISR();

  mics.ThreadEntry();
}
```

Now a call to `app_mic_array_task()` with the channel to send frames on can be placed inside a `par {...}` block to spawn the thread.

4 Decimator Stages

The mic array unit provided by this library uses a two-stage decimation process to convert a high sample rate stream of (1-bit) PDM samples into a lower sample rate stream of (32-bit) PCM samples.

Below is a *Simplified Decimator Model*.

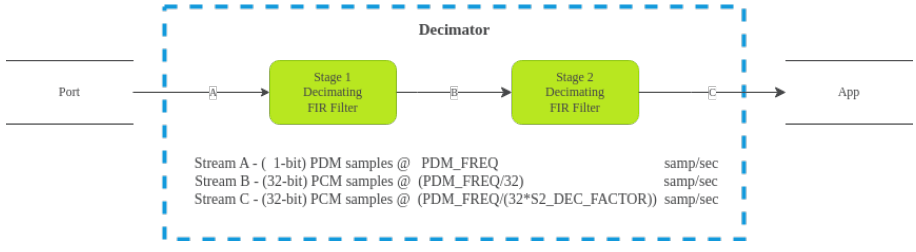

Fig. 2: Simplified Decimator Model

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`).

4.1 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`. This library also provides coefficients for the first stage filter, whose filter characteristics are adequate for most applications.

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 `PDM_FREQ/S1_DEC_FACTOR = 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.

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. See the section below on filter conversion if supplying a custom filter for stage 1.

Provided Filter (Stage 1)

This library provides filter coefficients that may be used with the first stage decimator. These coefficients are available in your application through the header `mic_array/etc/filters_default.h` as `stage1_coef`.

Filter Characteristics (Stage 1) The plot below indicates the frequency response of the provided first stage decimation filter *First stage decimation filter freq response*.

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 Python (3) script which does this process for you.

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

4.2 Decimator Stage 2

An application is free to supply its own second stage filter. This library also provides a second stage filter whose characteristics are adequate for many or most applications.

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 MHz / (32*6) = 16 kHz

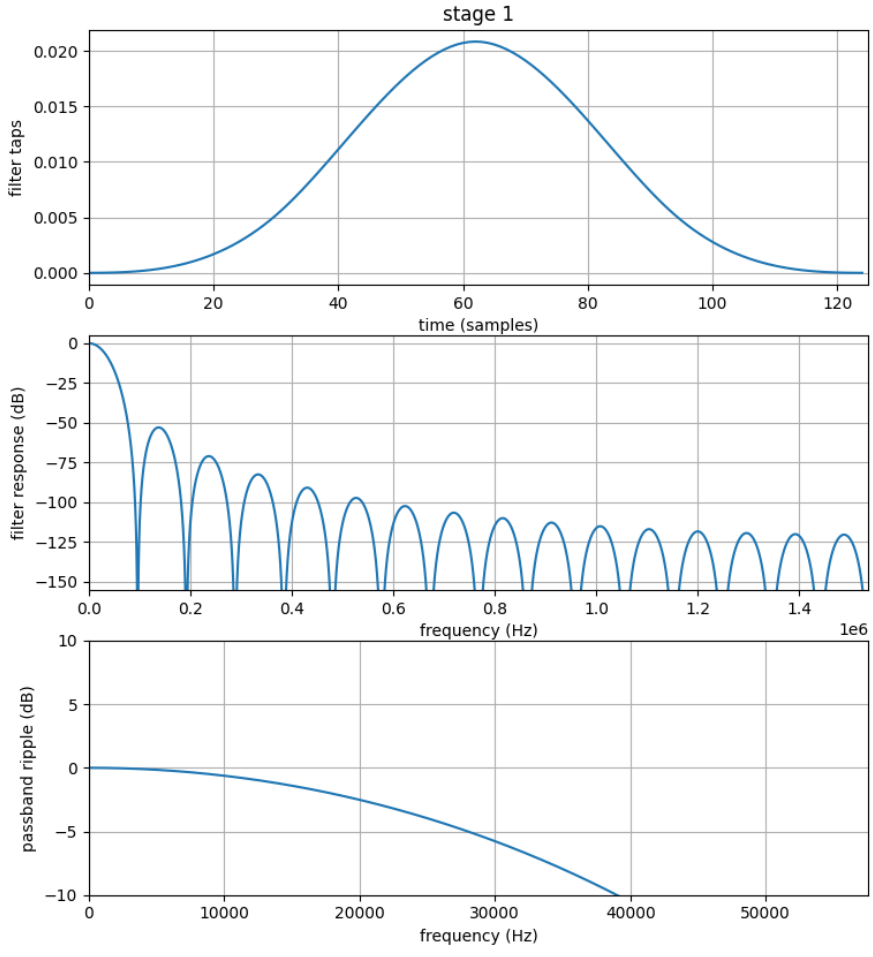

Fig. 3: First stage decimation filter freq response

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.

Provided Filter (Stage 2)

This library provides a filter suitable for the second stage decimator. It is available in your application through the header `mic_array/etc/filters_default.h`.

For the provided filter `S2_TAP_COUNT = 65`, and `S2_DEC_FACTOR = 6`.

Filter Characteristics (Stage 2) The plot below indicates the frequency response of the provided second stage decimation filter *Second stage decimation filter freq response*.

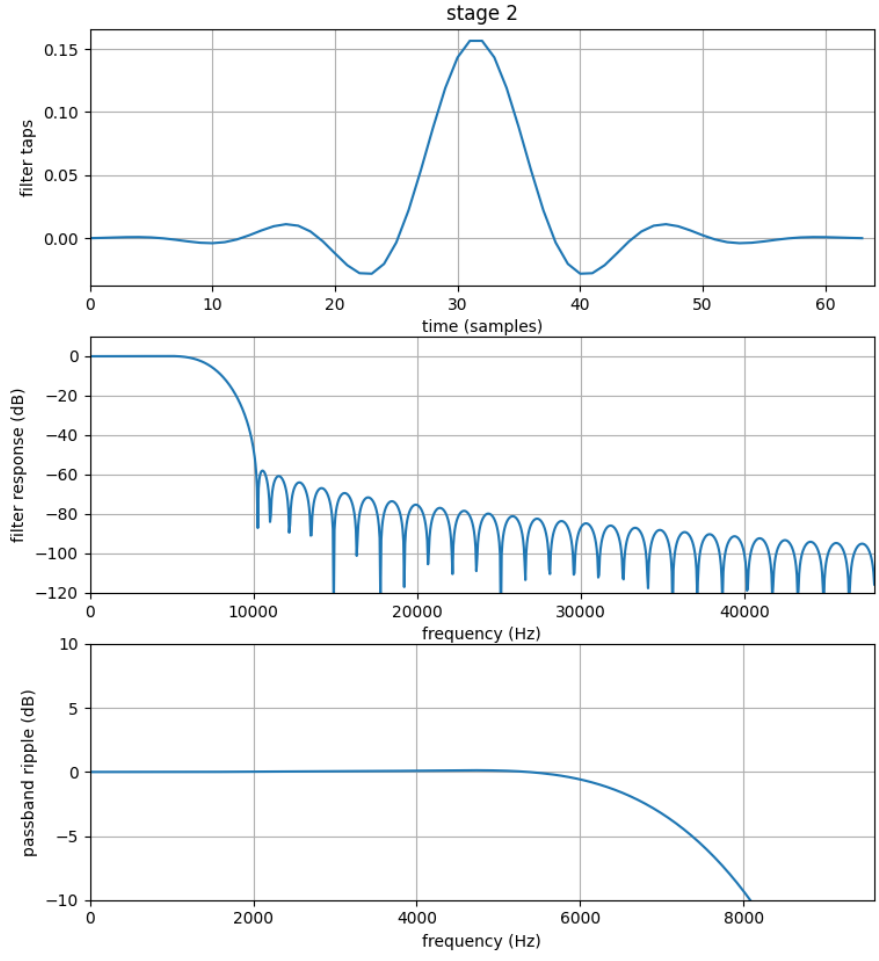

Fig. 4: Second stage decimation filter freq response

4.3 Custom Filters

Without writing a custom decimator implementation, the tap count and decimation factor for the first stage decimator are fixed to 256 and 32 respectively. These can be modified for the second stage, and the filter coefficients for both stages can be modified.

When using the C++ API to construct your application's mic array component, the decimator's metaparameters (tap count, decimation factor) are given as C++ template parameters for the decimator class template. Pointers to the coefficients are provided to the decimator when it is initialized.

To keep things simple, when using the vanilla API or when constructing the mic array component using *`BasicMicArray`*, it is assumed that the filter parameters will be those from `stage1_fir_coef.c`, `stage2_fir_coef.c` and `filters_default.h`. In

this case it is recommended to simple change those files directly with the updated coefficients. Otherwise you may need to use the C++ API directly.

Note that both the first and second stage filters are implemented using fixed-point arithmetic which requires the coefficients to be presented in a particular format. The Python scripts *stage1.py* and *stage2.py*, provided with this library, can be used to help with this formatting. See the associated README for usage details.

Configuring for 32 kHz or 48 kHz output

Filter design scripts are provided to support higher output sampling rates than the default 16 kHz.

Both stage 1 and stage 2 need to be updated because the first stage needs a higher cut off frequency before samples are passed to the downsample by three (32 kHz) or two (48 kHz) second stage decimator.

From the command line, follow these instructions:

```
python filter_design/design_filter.py # generate the filter .pkl files
python stage1.py good_32k_filter_int.pkl # convert the .pkl file to a C style array for stage 1
python stage2.py good_32k_filter_int.pkl # convert the .pkl file to a C style array for stage 2
```

Note: Use *good_48k_filter_int.pkl* instead of *good_32k_filter_int.pkl* to support 48 kHz.

Next copy the output from last two scripts into a source file. This could be your *mic_array.cpp* file which launches the mic array tasks. It may look something like this:

```
#define MIC_ARRAY_32K_STAGE_1_TAP_COUNT 148
#define MIC_ARRAY_32K_STAGE_1_FILTER_WORD_COUNT 128
static const uint32_t WORD_ALIGNED stage1_32k_coefs[MIC_ARRAY_32K_STAGE_1_FILTER_WORD_COUNT]
{
    .... the coeffs
};

#define MIC_ARRAY_32K_STAGE_2_TAP_COUNT 96
static constexpr right_shift_t stage2_32k_shift = 3;

static const int32_t WORD_ALIGNED stage2_32k_coefs[MIC_ARRAY_32K_STAGE_2_TAP_COUNT] = {
    .... the coeffs
};
```

The new decimation object must now be declared that references your new filter coefficients. Again, this example is for 32 kHz output since the decimation factor is 3.:

```
using TMicArray = mic_array::MicArray<mic_count,
    mic_array::TwoStageDecimator<mic_count,
                                 3,
                                 MIC_ARRAY_32K_STAGE_2_TAP_COUNT>,
    mic_array::StandardPdmRxService<MIC_ARRAY_CONFIG_MIC_IN_COUNT,
                                    mic_count,
                                    3>,
    typename std::conditional<MIC_ARRAY_CONFIG_USE_DC_ELIMINATION,
                                    mic_array::DcoeSampleFilter<mic_count>,
                                    mic_array::NopSampleFilter<mic_count>>::type,
    mic_array::FrameOutputHandler<mic_count,
                                    MIC_ARRAY_CONFIG_SAMPLES_PER_FRAME,
                                    mic_array::ChannelFrameTransmitter>>;
```

Next you need to change how you initialise and run the mic array task to reference your new mic array custom object. Normally the following code would be used in *ma_init()*:

```
mics.Init();
mics.SetPort(pdm_res.p_pdm_mics);
mic_array_resources_configure(&pdm_res, MIC_ARRAY_CONFIG_MCLK_DIVIDER);
mic_array_pdm_clock_start(&pdm_res);
```

however if you wish to use custom filters then the initialisation would look like this:

```
mics.Decimator.Init(stage1_32k_coefs, stage2_32k_coefs, stage2_32k_shift);
mics.PdmRx.Init(pdm_res.p_pdm_mics);
mic_array_resources_configure(&pdm_res, MIC_ARRAY_CONFIG_MCLK_DIVIDER);
mic_array_pdm_clock_start(&pdm_res);
```

Finally, the *ma_task()* function needs to be changed from the default way of calling:

```
mics.SetOutputChannel(c_frames_out);
mics.InstallPdmRxISR();
mics.UnmaskPdmRxISR();
mics.ThreadEntry();
```

to using the custom version of the object:

```
mics.OutputHandler.FrameTx.SetChannel(c_frames_out);
mics.PdmRx.InstallISR();
mics.PdmRx.UnmaskISR();
mics.ThreadEntry();
```

The increased sample rate will place a higher MIPS burden on the processor. The typical MIPS usage (see section *Mic Array 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 filer lengths to 148 and 96 for stages 1 and 2 respectively at 48 kHz will increase processor usage per channel to around 20 MIPS.

Filter Characteristics for *good_32k_filter_int.pkl* The plot below indicates the frequency response of the first and second stages of the provided 32 kHz filters as well as the cascaded overall response. Note that the overall combined response provides a nice flat passband as shown in the *good_32k_filter_int.pkl frequency response*.

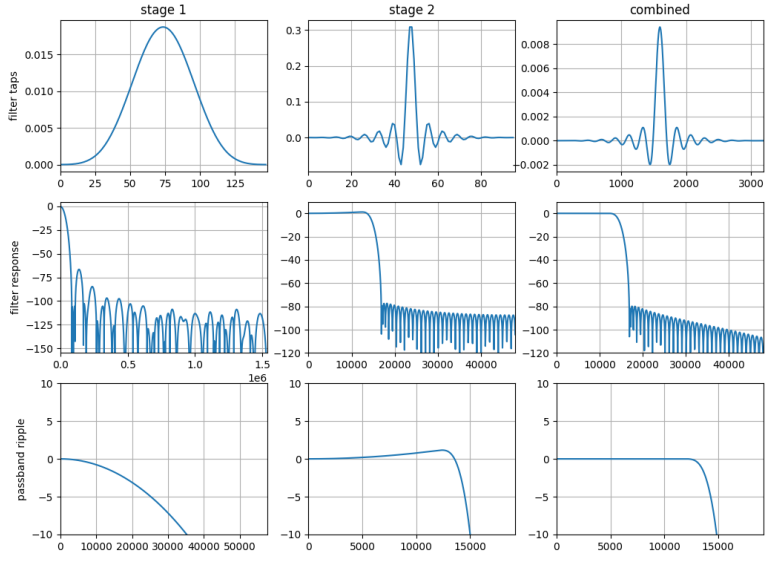

Fig. 5: good_32k_filter_int.pkl frequency response

Filter Characteristics for *good_48k_filter_int.pkl* The plot below indicates the frequency response of the first and second stages of the provided 48 kHz filters as well as the cascaded overall response. Note that the overall combined response provides a nice flat passband as shown *good_48k_filter_int.pkl frequency response*.

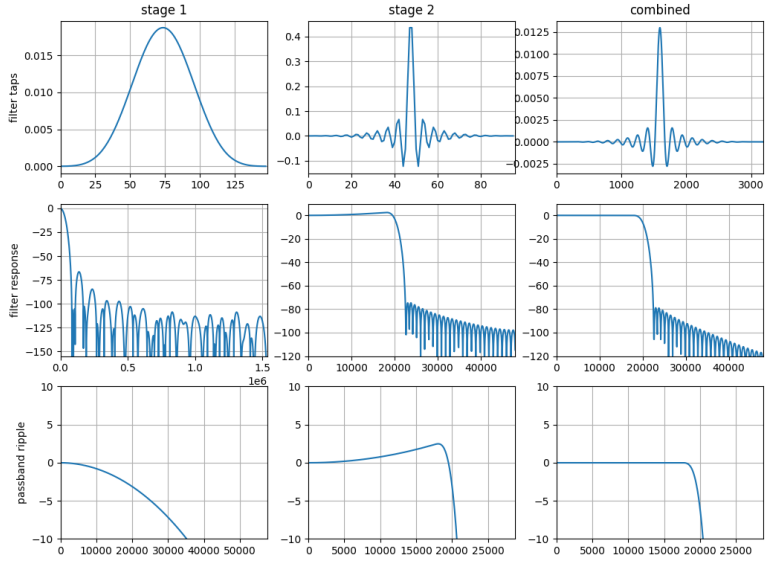

Fig. 6: good_48k_filter_int.pkl frequency response

5 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.

5.1 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.

Vanilla Model If your project uses the vanilla model (see *Vanilla API*) to include the mic array unit in your application, then DCOE is **enabled** by default. To disable DCOE your build script must add a compiler option to your application target that sets the `MIC_ARRAY_CONFIG_USE_DC_ELIMINATION` preprocessor macro to the value `0`.

For example, in a typical application's `CMakeLists.txt`, that may look like the following.

```
# Gather sources and create application target
# ...
# Add vanilla source to application build
mic_array_vanilla_add(my_app  ${MCLK_FREQ} ${PDM_FREQ}
                              ${MIC_COUNT} ${FRAME_SIZE} )
# ...
# Disable DCOE
target_compile_definitions(my_app
    PRIVATE MIC_ARRAY_CONFIG_USE_DC_ELIMINATION=0 )
```

Prefab Model If your project instantiates the *`BasicMicArray`* class template to include the mic array unit, DC offset elimination is enabled or disabled with the `USE_DCOE` boolean template parameter (there is no default).

```
template <unsigned MIC_COUNT, unsigned FRAME_SIZE, bool USE_DCOE>
    class BasicMicArray : public ...
```

The sample filter chosen is based on the `USE_DCOE` template parameter when the class template gets instantiated. If `true`, *`DcoeSampleFilter`* will be selected as the `MicArray` `SampleFilter` sub-component. Otherwise *`NopSampleFilter`* will be used.

Note: `NopSampleFilter` is a no-op filter – it does not modify the samples given to it and ultimately will be completely optimized out at compile time.

For example, in your application source:

```
#include "mic_array/mic_array.h"
...
// Controls whether DCOE is enabled
static constexpr bool enable_dcoe = true;
auto mics = mic_array::prefab::BasicMicArray<MICS, FRAME_SIZE, enable_dcoe>();
...
```

General Model If your project does not use either the vanilla or prefab models to include the mic array unit in your application, then precisely how the DCOE filter is included may depend on the specifics of your 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*.

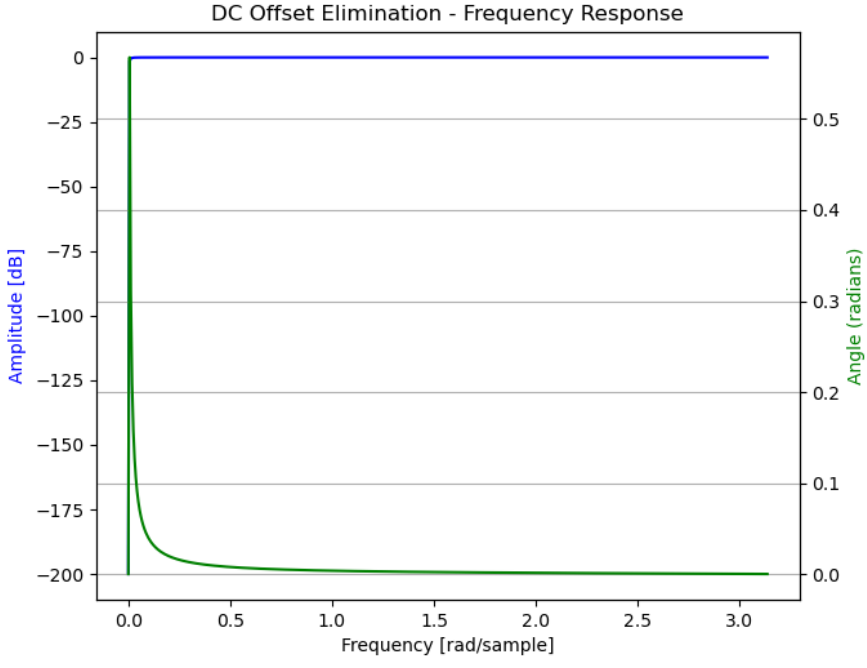

Fig. 7: DCOE filter frequency response

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 behavior 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:

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.

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).

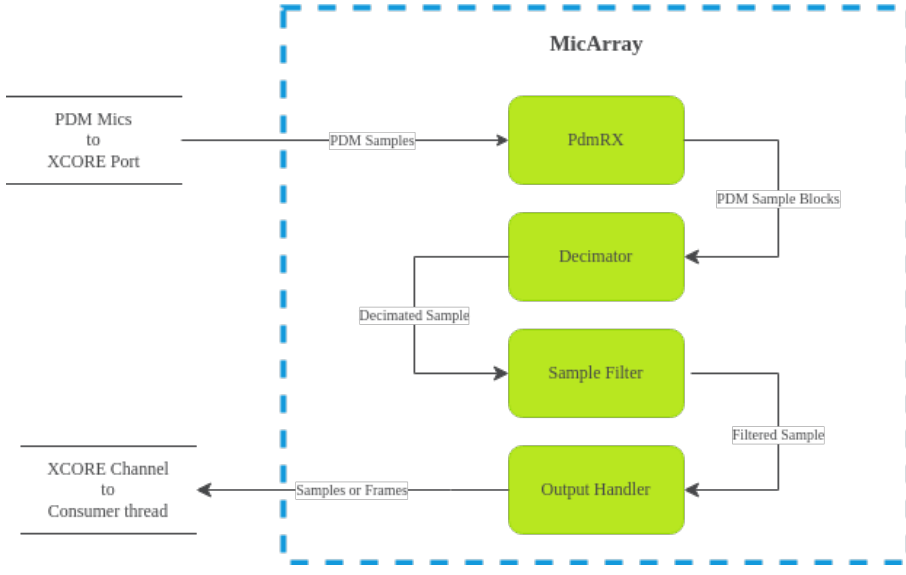

Fig. 8: Mic Array High Level Process

```
void mic_array::MicArray<MIC_COUNT,TDecimator,TPdmRx,
                         TSampleFilter,
                         TOutputHandler>::ThreadEntry()
{
  int32_t sample_out[MIC_COUNT] = {0};

  while(1){
    uint32_t* pdm_samples = PdmRx.GetPdmBlock();
    Decimator.ProcessBlock(sample_out, pdm_samples);
    SampleFilter.Filter(sample_out);
    OutputHandler.OutputSample(sample_out);
  }
}
```

The thread loops forever, 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.

Curiously Recurring Template Pattern

The C++ API of this library makes heavy use of the Curiously Recurring Template Pattern (CRTP).

Instead of providing flexibility through abstract classes or polymorphism, CRTP achieves flexibility through the use of class templates with type template parameters. As with derived classes and virtual methods, the CRTP template parameter must follow a contract with the class template where it implements one or more methods with specific names and signatures that the class template directly calls.

There are a couple notable advantages of using CRTP over polymorphic behavior. With CRTP flexibility does not generally come with the same run-time costs (in terms of both compute and memory) as polymorphic solutions. This is because the CRTP class template always knows the concrete type of any objects it uses at compile time. This avoids the need for run time type information or virtual function tables. This allows compile time optimizations can be made which may not be otherwise available. This in-turn allows many function calls to be inlined, or in some cases, entirely eliminated.

Additionally, while not strictly an example of CRTP, integer template parameters are also heavily used in class templates. The two main advantages of this are that it allows objects to encapsulate their own (statically allocated) memory, and that it allows the compiler to make compile time loop optimizations that it may not otherwise be able to make.

The downside to CRTP is that it tends to lead to highly verbose class type names, where templated classes end up with type parameter assignments are themselves templated classes with their own template parameters.

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 table above). 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.

Generally speaking, `PdmRx` will derive from the *`PdmRxService`* class template. `PdmRxService` 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.

`PdmRxService` has 2 template parameters. The first is the `BLOCK_SIZE`, which specifies the size of a PDM sample block (in words). The second, `SubType`, is the type of the sub-class being derived from `PdmRxService`. This is the CRTP (Curiously Recurring Template Pattern), which allows a base class to use polymorphic-like behaviors while ensuring that all types are known at compile-time, avoiding the drawbacks of using virtual functions.

There is currently one class template which derives from `PdmRxService`, called *`StandardPdmRxService`*. `StandardPdmRxService` 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 *Decimator Stages*.

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 *`DcoeSampleFilter`* 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.

Prefabs

One of the drawbacks to broad use of class templates is that concrete class names can unfortunately become excessively verbose and confusing. For example, the following is the fully qualified name of a (particular) concrete `MicArray` implementation:

```
mic_array::MicArray<2,
    mic_array::TwoStageDecimator<2,6,65>,
    mic_array::StandardPdmRxService<2,2,6>,
    mic_array::DcoeSampleFilter<2>,
    mic_array::FrameOutputHandler<2,256,
        mic_array::ChannelFrameTransmitter>>
```

This library also provides a C++ namespace `mic_array::prefab` which is intended to simplify construction of `MicArray` objects where common configurations are needed.

The *BasicMicArray* class template uses the most typical component implementations, where PDM rx can be run as an interrupt or as a stand-alone thread, and where audio frames are transmitted to subsequent processing stages using a channel.

To demonstrate how **BasicMicArray** simplifies this process, observe that the following **MicArray** type is behaviorally identical to the above:

```
mic_array::prefab::BasicMicArray<2,256,true>
```

7 Mic Array Resource Usage

The mic array unit requires several kinds of hardware resources, including ports, clock blocks, chanends, hardware threads, compute time (MIPS) and memory. Compared to previous versions of this library, the biggest advantage to the current version with respect to hardware resources is a greatly reduced compute requirement. This was made possible by the introduction of the VPU in the XMOS XS3 architecture. The VPU can do certain operations in a single instruction which would take many, many instructions on previous architectures.

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

Warning: The usage information below applies when the Vanilla API or prefab APIs are used. Resource usage in an application which uses custom mic array subcomponents will depend crucially on the specifics of the customization.

7.1 Discrete Resources

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

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 transfering completed frames from the mic array unit to other application components.

Threads

The prefab API can run the PDM rx service either as a stand-alone thread or as an interrupt in another thread. The Vanilla API only supports running it as an interrupt. The Vanilla API requires only on hardware thread. The prefab API requires 1 thread if PDM rx is used in interrupt mode, and 2 if PDM rx is 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.

7.2 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 pipline. 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 (~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. With a core clock rate of 600 MHz, that means that each core should expect at least 75 MIPS.

The MIPS values in the table below are estimates obtained using the demo applications in `demo/measure_mips`.

PDM Freq	S2DF	S2TC	PdmRx	1 mic MIPS	2 mic MIPS	4 mic MIPS	8 mic MIPS
3.072 MHz	6	65	ISR	10.65	22.00	43.70	N/A
3.072 MHz	6	65	Thread	9.33	19.37	38.48	75.90
6.144 MHz	6	65	ISR	21.26	43.89	TBD	TBD
6.144 MHz	6	65	Thread	18.66	38.73	TBD	TBD
3.072 MHz	3	65	ISR	12.90	26.44	TBD	TBD
3.072 MHz	3	65	Thread	11.62	23.85	TBD	TBD
3.072 MHz	6	130	ISR	11.17	23.04	TBD	TBD
3.072 MHz	6	130	Thread	9.86	20.42	TBD	TBD

PDM Freq
Frequency of the PDM clock.

S2DF
Stage 2 decimation factor. Output sample rate is `(PDM Freq / (32 * S2DF))`.

S2TC
Stage 2 tap count.

PdmRx
Whether PDM capture is done in a stand-alone thread or in an ISR.

Measurements indicate that enabling or disabling the DC offset removal filter has little effect on the MIPS usage. The selected frame size has only a slight negative correlation with MIPS usage.

7.3 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 persistant objects, variables and constants.

The stack memory requirement is minimal. The code memory requirement depends on the particular configuration, but ranges from about `1600` bytes in a 1 mic configuration to about `2000` bytes in an 8 mic configuration.

Not included in the table is the space allocated for the first and second stage filter coefficients. The first stage filter coefficients take a constant `523` bytes. The second stage filter coefficients use `4*S2TC` bytes, where `S2TC` is the stage 2 decimator tap count. The value shown in the 'data' column of the table is the `sizeof()` the *`BasicMicArray`* that is instantiated. The table below indicates the data size for various configurations.

Mics	S2DF	S2TC	SPF	DCOE	Data Memory
1	6	65	16	On	504 B
2	6	65	16	On	968 B
4	6	65	16	On	1888 B
8	6	65	16	On	3728 B
1	6	65	16	On	768 B
2	6	130	16	On	1488 B
1	6	130	16	On	576 B
2	12	65	16	On	1112 B
1	12	65	160	On	1080 B
2	6	65	160	On	2120 B
1	6	65	16	Off	496 B
2	6	65	16	Off	948 B

S2DF
Stage 2 decimator's decimation factor.

S2TC
Stage 2 decimator's tap count.

SPF
Samples per frame in frames delivered by the mic array unit.

DCOE
DC Offset Elimination

8 Vanilla API

The Vanilla API is a small optional API which greatly simplifies the process of including a mic array unit in an xcore.ai application. Most applications that make use of a PDM mic array will not have complicated needs from the mic array software component beyond delivery of frames of audio data from a configurable set of microphones at a configurable rate. This API targets that majority of applications.

The prefab API requires the application developer to have at least some minimal understanding of the objects and classes associated with the mic array unit, and requires the developer to write some application-specific code to configure and start the mic array. The Vanilla API (which builds on top of the prefab model) by contrast, requires as little as two standard function calls, and instead moves the majority of the application logic into the application's build project.

Note: Why "Vanilla"? "Vanilla" was originally meant as a generic placeholder name, but no better name was ever suggested.

8.1 How It Works

The Vanilla API comprises two code files, `etc/vanilla/mic_array_vanilla.cpp` and `etc/vanilla/mic_array_vanilla.h` which are not compiled as part of this library. Instead, if used, these are added to the application target's build. To control configuration, the source file relies on a set of pre-processor macros (added via compile flags) which determine how the mic array unit will be instantiated.

The API is included in an application by using a CMake macro (`mic_array_vanilla_add()`) provided in this library. The macro updates the application's sources, includes and compile definitions to include the API.

In the application code, two function calls are needed. First, *`ma_vanilla_init()`* is called to initialize the various mic array sub-components, preparing for capture of PDM data. Then, to start capture the decimation thread is started with *`ma_vanilla_task()`* as entrypoint. *`ma_vanilla_task()`* takes an XCore **chanend** as a parameter, which tells it where completed audio frames should be routed.

Note: The Vanilla API runs the PDM rx service as an interrupt in the decimation thread. To run it as a separate thread (for reduced total MIPS consumption) one of the lower-level APIs must be used.

As with the prefab API, audio frames are extracted from the mic array unit over a (non-streaming) channel using the *`ma_frame_rx()`* or *`ma_frame_rx_transpose()`* functions.

Note: The Vanilla API uses the default filters provided with this library, and does not currently provide a way to override this. To use custom filters, you must either use a lower-level API or modify the vanilla API.

8.2 Configuration

Configuration with the Vanilla API is achieved through compile definitions. The required definitions are provided through the `mic_array_vanilla_add()` macro. There are several additional optional definitions.

mic_array_vanilla_add()

`mic_array_vanilla_add()` is the CMake macro used to add the Vanilla API to an application.

```
macro( mic_array_vanilla_add
         TARGET_NAME
         MCLK_FREQ
         PDM_FREQ
         MIC_COUNT
         SAMPLES_PER_FRAME )
```

`TARGET_NAME`
: The name of the application's CMake target. It is the target the Vanilla API is added to.

`MCLK_FREQ`
: The known frequency, in Hz, of the application's master audio clock. A typical frequency is 24576000 Hz. Note that this parameter is not *configuring* the master audio clock. (Equivalent compile definition: `MIC_ARRAY_CONFIG_MCLK_FREQ`)

`PDM_FREQ`
: The desired frequency, in Hz, of the PDM clock. This should be an integer factor of `MCLK_FREQ` between `1` and `510`. (Equivalent compile definition: `MIC_ARRAY_CONFIG_PDM_FREQ`)

`MIC_COUNT`
: The number of PDM microphone channels to be captured. This API supports values of `1` (SDR), `2` (DDR), `4` (SDR) and `8` (SDR/DDR). This value must match the

configuration (SDR/DDR) and port width of the PDM capture port. That is, in an SDR port configuration, `MIC_COUNT` must equal the capture port width, and in DDR port configuration, `MIC_COUNT` must be twice the port width. (Equivalent compile definition: `MIC_ARRAY_CONFIG_MIC_COUNT`)

Note: This API does not support capturing only a subset of the capture port's channels, e.g. capturing only 3 channels on a 4-bit port. To accomplish this the prefab API should be used.

Note: Though listed under Optional Configuration below, if the microphones are in a DDR configuration and `MIC_COUNT` is not `2`, the application must also define `MIC_ARRAY_CONFIG_USE_DDR`.

`SAMPLES_PER_FRAME` is the number of samples (for each microphone channel) that will be delivered in each (non-overlapping) frame retrieved by `ma_frame_rx()`. A minimum value of `1` is supported, to deliver samples one at a time. The larger this value, the looser the real-time constraint on the thread receiving the mic array unit's output (while also increasing the amount of audio data to be processed).

Optional Configuration

These are configuration parameters that receive default values but can be optionally overridden by an application. These can be defined in your application's `CMakeLists.txt` using CMake's built-in `target_compile_definitions()` command.

MIC_ARRAY_CONFIG_USE_DDR

Indicates whether the microphones are arranged in an SDR (`0`) or DDR (`1`) configuration. An SDR configuration is one in which each port pin is connected to a single PDM microphone. A DDR configuration is one which each port pin is connected to two PDM microphones. Defaults to `0` (SDR), unless `MIC_ARRAY_CONFIG_MIC_COUNT` is `2` in which case it defaults to `1` (DDR).

MIC_ARRAY_CONFIG_USE_DC_ELIMINATION

Indicates whether the *DC offset elimination* filter should be applied to the output of the decimator. Set to `0` to disable or `1` to enable. Defaults to `1` (filter on).

The next three parameters are the identifiers for hardware port resources used by the mic array unit. They can be specified as either the identifier listed in your device's datasheet (e.g. `XS1_PORT_1D`) or as an alias for the port listed in your application's XN file (e.g. `PORT_MCLK_IN_OUT`). For example:

```
...
<Tile Number="0" Reference="tile[0]">
...
  <Port Location="XS1_PORT_1D"  Name="PORT_MCLK_IN_OUT"/>
...
</Tile>
...
```

MIC_ARRAY_CONFIG_PORT_MCLK

Identifier of the 1-bit port on which the device is receiving the master audio clock. Defaults to `PORT_MLCK_IN_OUT`.

MIC_ARRAY_CONFIG_PORT_PDM_CLK

Identifier of the 1-bit port on which the device will signal the PDM clock to the microphones. Defaults to `PORT_PDM_CLK`.

MIC_ARRAY_CONFIG_PORT_PDM_DATA
Identifier of the port on which the device will capture PDM sample data. The port width of this port must match the `MIC_COUNT` parameter given to `mic_array_vanilla_add()` and the value of `MIC_ARRAY_CONFIG_USE_DDR`. Defaults to `PORT_PDM_DATA`.

The final two parameters indicate which clock block resource(s) should be used to generate the PDM clock and the capture clock. An xcore.ai device provides 5 hardware clock blocks for application use, identified as `XS1_CLKBLK_1` through `XS1_CLKBLK_5`. The device's clock blocks are interchangeable, but if another component of your application uses one of these defaults, you may need to change these parameters.

MIC_ARRAY_CONFIG_CLOCK_BLOCK_A
Clock block used as 'clock A' (see *Getting Started*). This clock block is used in both SDR and DDR configurations.

MIC_ARRAY_CONFIG_CLOCK_BLOCK_B
Clock block used as 'clock B' (see *Getting Started*). This clock block is only needed in DDR configurations and is ignored (not configured) in SDR configurations.

Vanilla API with other Build Systems

Using the Vanilla API with build systems other than CMake is simple.

- Add the file `etc/vanilla/mic_array_vanilla.cpp` to the application's source files.
- Add `etc/vanilla/` (relative to repository root) to the application include paths.
- Add the compile definitions for the parameters listed in the previous sections (each parameter beginning with `MIC_ARRAY_CONFIG_`) to the compile options for `mic_array_vanilla.cpp`.

9 API Reference

9.1 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.

Like many classes in this library, *FrameOutputHandler* uses the *Curiously Recurring Template Pattern*.

Template Parameters

- **MIC_COUNT** – The number of microphones to be *captured* by the *MicArray*'s `PdmRx` component. 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) `MIC_COUNT` 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.
- **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*.

inline **MicArray**(*TPdmRx* pdm_rx, *TSampleFilter* sample_filter, *TOutputHandler* output_handler)

Construct a *MicArray*.

This constructor uses the default constructor for its *Decimator* component.

The remaining components are initialized with the supplied objects.

Parameters

- **pdm_rx** – The PDM rx object.
- **sample_filter** – The SampleFilter object.
- **output_handler** – The OutputHandler object.

inline **MicArray**(*TPdmRx* pdm_rx, *TOutputHandler* output_handler)

Construct a *MicArray*

This constructor uses the default constructor for its *Decimator* and *SampleFilter* components.

The remaining components are initialized with the supplied objects.

Parameters

- **pdm_rx** – The PDM rx object.
- **output_handler** – The OutputHandler object.

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[BLOCK_SIZE]);
```

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

A concrete class based on the *mic_array::TwoStageDecimator* class template is used in the *prefab::BasicMicArray* prefab.

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.

A concrete class based on either the *DcoeSampleFilter* class template or the *NopSampleFilter* class template is used in the *prefab::BasicMicArray* prefab, depending on the `USE_DCOE` parameter of that class template.

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.

A concrete class based on the *FrameOutputHandler* class template is used in the *prefab::BasicMicArray* prefab.

Public Static Attributes

static constexpr unsigned **`MicCount`** = *MIC_COUNT*

Number of microphone channels.

BasicMicArray

template<unsigned **MIC_COUNT**, unsigned **FRAME_SIZE**, bool **USE_DCOE**, unsigned **MICS_IN** = *MIC_COUNT*>
class **BasicMicArray** : public mic_array::*MicArray*<*MIC_COUNT*, *TwoStageDecimator*<*MIC_COUNT*, STAGE2_DEC_FACTOR, STAGE2_TAP_COUNT>, *StandardPdmRxService*<*MIC_COUNT*, *MIC_COUNT*, STAGE2_DEC_FACTOR>, std::conditional<*USE_DCOE*, *DcoeSampleFilter*<*MIC_COUNT*>, *NopSampleFilter*<*MIC_COUNT*»::type, *FrameOutputHandler*<*MIC_COUNT*, *FRAME_SIZE*, *ChannelFrameTransmitter*»

Class template for a typical bare-metal mic array unit.

This prefab is likely the right starting point for most applications.

With this prefab, the decimator will consume one device core, and the PDM rx service can be run either as an interrupt, or as an additional thread. Normally running as an interrupt is recommended.

For the first and second stage decimation filters, this prefab uses the coefficients provided with this library. The first stage uses a decimation factor of 32, and the second stage is configured to use a decimation factor of 6.

To get 16 kHz audio output from the *`BasicMicArray`* prefab, then, the PDM clock must be configured to `3.072 MHz` (`3.072 MHz / (32 * 6) = 16 kHz`).

Sub-Components

Being derived from *mic_array::MicArray*, an instance of *`BasicMicArray`* has 4 sub-components responsible for different portions of the work being done. These sub-components are `PdmRx`, `Decimator`, `SampleFilter` and `OutputHandler`. See the documentation for *`MicArray`* for more details about these.

Template Parameters Details

The template parameter `MIC_COUNT` is the number of microphone channels to be processed and output.

The template parameter `FRAME_SIZE` is the number of samples in each output frame produced by the mic array. Frame data is communicated using the API found in `mic_array/frame_transfer.h`. Typically *`ma_frame_rx()`* will be the right function to use in a receiving thread to retrieve audio frames. `ma_frame_rx()` receives audio frames with shape `(MIC_COUNT, FRAME_SIZE)`, meaning that all samples corresponding to a given channel will end up in a contiguous block of memory. Instead of `ma_frame_rx()`, *`ma_frame_rx_transpose()`* can be used to swap the dimensions, resulting in the shape `(FRAME_SIZE, MIC_COUNT)`.

Note that calls to `ma_frame_rx()` or `ma_frame_rx_transpose()` will block until a frame becomes available on the specified chanend.

If the receiving thread is not waiting to retrieve the audio frame from the mic array when it becomes available, the pipeline may back up and cause samples to be dropped. It is the responsibility of the application developer to ensure this does not happen.

The boolean template parameter `USE_DCOE` indicates whether the DC offset elimination filter should be applied to the output of the second stage decimator. DC offset elimination is an IIR filter intended to ensure audio samples on each channel tend towards zero-mean.

For more information about DC offset elimination, see *Sample Filters* .

If **USE_DCOE** is **false**, no further filtering of the second stage decimator's output will occur.

The template parameter **MICS_IN** indicates the number of microphone channels to be captured by the **PdmRx** component of the mic array unit. This will often be the same as **MIC_COUNT**, but in some applications, **MIC_COUNT** microphones must be physically connected to an XCore port which is not **MIC_COUNT** (SDR) or **MIC_COUNT/2** (DDR) bits wide.

In these cases, capturing the additional channels (likely not even physically connected to PDM microphones) is unavoidable, but further processing of the additional (junk) channels can be avoided by using **MIC_COUNT < MICS_IN**. The mapping which tells the mic array unit how to derive output channels from input channels can be configured during initialization by calling *StandardPdmRxService::MapChannels()* on the **PdmRx** subcomponent of the **BasicMicarray**.

If the application uses an SDR microphone configuration (i.e. 1 microphone per port pin), then **MICS_IN** must be the same as the port width. If the application is running in a DDR microphone configuration, **MICS_IN** must be twice the port width. **MICS_IN** defaults to **MIC_COUNT**.

Allocation

Before a mic array unit can be started or initialized, it must be allocated.

Instances of *BasicMicArray* are self-contained with respect to memory, needing no external buffers to be supplied by the application. Allocating an instance is most easily accomplished by simply declaring the mic array unit. An example follows.

```
#include "mic_array/cpp/Prefab.hpp"
...
using AppMicArray = mic_array::prefab::BasicMicArray<MICS,SAMPS,DCOE>;
AppMicArray mics;
```

Here, **mics** is an allocated mic array unit. The example (and all that follow) assumes the macros used for template parameters are defined elsewhere.

Initialization

Before a mic array unit can be started, it must be initialized.

BasicMicArray reads PDM samples from an XCore port, and delivers frames of audio data over an XCore channel. To this end, an instance of *BasicMicArray* needs to be given the resource IDs of the port to be read and the chanend to transmit frames over. This can be accomplished in either of two ways.

If the resource IDs for the port and chanend are available as the mic array unit is being allocated, one option is to explicitly construct the *BasicMicArray* instance with the required resource IDs using the two-argument constructor:

```
using AppMicArray = mic_array:prefab::BasicMicArray<MICS,SAMPS,DCOE>;
AppMicArray mics(PORT_PDM_MICS, c_frames_out);
```

Otherwise (typically), these can be set using *BasicMicArray::SetPort(port_t)* and *BasicMicArray::SetOutputChannel(chanend_t)* to set the port and channel respectively.

```
AppMicArray mics;
...
void app_init(port_t p_pdm_mics, chanend_t c_frames_out)
{
 mics.SetPort(p_pdm_mics);
 mics.SetOutputChannel(p_pdm_mics);
}
```

Next, the ports and clock block(s) used by the PDM rx service need to be configured appropriately. This is not accomplished directly through the *BasicMicArray* object. Instead, a *pdm_rx_resources_t* struct representing these hardware resources is constructed and passed to `mic_array_resources_configure()`. See the documentation for *pdm_rx_resources_t* and `mic_array_resources_configure()` for more details.

Finally, if running *BasicMicArray*'s PDM rx service within an ISR, before the mic array unit can be started, the ISR must be installed. This is accomplished with a call to *BasicMicArray::InstallPdmRxISR()*. Installing the ISR will *not* unmask it.

Begin Processing (PDM rx ISR)

After it has been initialized, starting the mic array unit with the PDM rx service running as an ISR, three steps are required.

First, the PDM clock must be started. This is accomplished with a call to `mic_array_pdm_clock_start()`. The same *pdm_rx_resources_t* that was passed to `mic_array_resources_configure()` is given as an argument here.

Second, the PDM rx ISR that was installed during initialization must be unmasked. This is accomplished by calling *BasicMicArray::UnmaskPdmRxISR()* on the mic array unit.

Finally, the mic array processing thread must be started. The entry point for the mic array thread is *BasicMicArray::ThreadEntry()*.

A typical pattern will include all three of these steps in a single function which wraps the mic array thread entry point.

```
AppMicArray mics;
pdm_rx_resources_t pdm_res;
...
MA_C_API  // alias for 'extern "C"'
void app_mic_array_task()
{
 mic_array_pdm_clock_start(&pdm_res);
 mics.UnmaskPdmRxISR();
 mics.ThreadEntry();
}
```

Using this pattern, `app_mic_array_task()` is a C-compatible function which can be called from a multi-tile `main()` in an XC file. Then, `app_mic_array_task()` is called directly from a `par {...}` block. For example,

```
main(){
 ...
 par {
   on tile[1]: {
     ... // Do initialization stuff

     par {
       app_mic_array_task();
       ...
       other_thread_on_tile1(); // other threads
```

(continues on next page)

(continued from previous page)

```
    }
   }
  }
 }
```

Begin Processing (PDM Rx Thread)

The procedure for running the mic array unit with the PDM rx component running as a stand-alone thread is much the same with just a couple key differences.

When running PDM rx as a thread, no call to *BasicMicArray::UnmaskPdmRxISR()* is necessary. Instead, the application spawns a second thread (the first being the mic array processing thread) using *BasicMicArray::PdmRxThreadEntry()* as the entry point.

`mic_array_pdm_clock_start()` must still be called, but here the requirement is that it be called from the hardware thread on which the PDM rx component is running (which, of course, cannot be the mic array thread).

A typical application with a multi-tile XC `main()` will provide two C-compatible functions - one for each thread:

```
MA_C_API
void app_pdm_rx_task()
{
 mic_array_pdm_clock_start(&pdm_res);
 mics.PdmRxThreadEntry();
}

MA_C_API
void app_mic_array_task()
{
 mics.ThreadEntry();
}
```

Notice that `app_mic_array_task()` above is a thin wrapper for `mics.ThreadEntry()`. Unfortunately, because the type of `mics` is a C++ class, `mics.ThreadEntry()` cannot be called directly from an XC file (including the one containing `main()`). Further, because a C++ class template was used, this library cannot provide a generic C-compatible call wrapper for the methods on a *MicArray* object. This unfortunately means it is necessary in some cases to create a thin wrapper such as `app_mic_array_task()`.

The threads are spawned from XC main using a `par {...}` block:

```
main(){
 ...
 par {
   on tile[1]: {
     ... // Do initialization stuff

     par {
       app_mic_array_task();
       app_pdm_rx_task();
       ...
       other_thread_on_tile1(); // other threads
     }
   }
 }
}
```

Real-Time Constraint

Once the PDM rx thread is launched or the PDM rx interrupt has been unmasked, PDM data will start being collected and reported to the decimator

thread. The application then must start the decimator thread within one output sample time (i.e. sample time for the output of the second stage decimator) to avoid issues.

Once the mic array processing thread is running, the real-time constraint is active for the thread consuming the mic array unit's output, and it must waiting to receive an audio frame within one frame time.

Examples

This library comes with examples which demonstrate how a mic array unit is used in an actual application. If you are encountering difficulties getting *BasicMicArray* to work, studying the provided examples may help.

Note: *BasicMicArray::InstallPdmRxISR()* installs the ISR on the hardware thread that calls the method. In most cases, installing it in the same thread as the decimator is the right choice.

Template Parameters

- **MIC_COUNT** – Number of microphone channels.
- **FRAME_SIZE** – Number of samples in each output audio frame.
- **USE_DCOE** – Whether DC offset elimination should be used.

Public Types

using **TParent** = *MicArray*<*MIC_COUNT*, *TwoStageDecimator*<*MIC_COUNT*, STAGE2_DEC_FACTOR, STAGE2_TAP_COUNT>, *StandardPdmRxService*<*MICS_IN*, *MIC_COUNT*, STAGE2_DEC_FACTOR>, typename std::conditional<*USE_DCOE*, *DcoeSampleFilter*<*MIC_COUNT*>, *NopSampleFilter*<*MIC_COUNT*»::type, *FrameOutputHandler*<*MIC_COUNT*, *FRAME_SIZE*, *ChannelFrameTransmitter*»

`TParent` is an alias for this class template from which this class template inherits.

Public Functions

inline constexpr **BasicMicArray**() noexcept

No-argument constructor.

This constructor allocates the mic array and nothing more.

Call *BasicMicArray::Init()* to initialize the decimator.

Subsequent calls to *BasicMicArray::SetPort()* and *BasicMicArray::SetOutputChannel()* will also be required before any processing begins.

void **Init**()

Initialize the decimator.

BasicMicArray(port_t p_pdm_mics, chanend_t c_frames_out)

Initialzing constructor.

If the communication resources required by *BasicMicArray* are known at construction time, this constructor can be used to avoid further initialization steps.

This constructor does *not* install the ISR for PDM rx, and so that must be done separately if PDM rx is to be run in interrupt mode.

Parameters

- **p_pdm_mics** – Port with PDM microphones
- **c_frames_out** – (non-streaming) chanend used to transmit frames.

void **SetPort**(port_t p_pdm_mics)

Set the PDM data port.
This function calls `this->PdmRx.Init(p_pdm_mics)`.
This should be called during initialization.

Parameters

p_pdm_mics – The port to receive PDM data on.

void **SetOutputChannel**(chanend_t c_frames_out)

Set the audio frame output channel.
This function calls `this->OutputHandler.FrameTx.SetChannel(c_frames_out)`.
This must be set prior to entrying the decimator task.

Parameters

c_frames_out – The channel to send audio frames on.

void **PdmRxThreadEntry**()

Entry point for PDM rx thread.
This function calls `this->PdmRx.ThreadEntry()`.

Note: This call does not return.

---

void **InstallPdmRxISR**()

Install the PDM rx ISR on the calling thread.
This function calls `this->PdmRx.InstallISR()`.

void **UnmaskPdmRxISR**()

Unmask interrupts on the calling thread.
This function calls `this->PdmRx.UnmaskISR()`.

PdmRxService

template<unsigned **BLOCK_SIZE**, class **SubType**>

class **PdmRxService**

Collects PDM sample data from a port.

Derivatives of this class template are intended to be used for the `TPdmRx` template parameter of *MicArray*, where it represents the *MicArray::PdmRx* component of the mic array.

An object derived from *`PdmRxService`* collects blocks of PDM samples from a port and makes them available to the decimation thread as the blocks are completed.

`PdmRxService` is a base class using CRTP. Subclasses extend *`PdmRxService`* providing themselves as the template parameter `SubType`.

This base class provides the logic for aggregating PDM data taken from a port into blocks, and a subclass is required to provide methods `SubType::ReadPort()`, `SubType::SendBlock()` and `SubType::GetPdmBlock()`.

`SubType::ReadPort()` is responsible for reading 1 word of data from `p_pdm_mics`. See *StandardPdmRxService::ReadPort()* as an example.

`SubType::SendBlock()` is provided a block of PDM data as a pointer and is responsible for signaling that to the subsequent processing stage. See *StandardPdmRxService::SendBlock()* as an example.

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

`SubType::GetPdmBlock()` responsible for receiving a block of PDM data from `SubType::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 *StandardPdmRxService::GetPdmBlock()* as an example.

`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.

Template Parameters

- **BLOCK_SIZE** – Number of words of PDM data per block.
- **SubType** – Subclass of *`PdmRxService`* actually being used.

Public Functions

void **SetPort**(port_t p_pdm_mics)

Set the port from which to collect PDM samples.

void **ProcessNext**()

Perform a port read and if a new block has completed, signal.

void **ThreadEntry**()

Entry point for PDM processing thread.

This function loops forever, calling *`ProcessNext()`* with each iteration.

Public Static Attributes

static constexpr unsigned **BlockSize** = *BLOCK_SIZE*

Number of words of PDM data per block.

Typically (e.g. *TwoStageDecimator*) `BLOCK_SIZE` will be exactly the number of words of PDM samples required to produce exactly one new output sample for the mic array unit's output stream.

Once `BlockSize` words have been read into one of the block_data, buffers, PDM rx will signal to the decimator thread that new PDM data is available for processing.

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 an instance of *mic_array::PdmRxService*. 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`**, unsigned **`SUBBLOCKS`**>

class **StandardPdmRxService** : public mic_array::*PdmRxService*<*CHANNELS_IN* * *SUBBLOCKS*, *StandardPdmRxService*<*CHANNELS_IN*, *CHANNELS_OUT*, *SUBBLOCKS*»

PDM rx service which uses a streaming channel to send a block of data by pointer.

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.

Layouts

The buffer transferred by *SendBlock()* contains `CHANNELS_IN*SUBBLOCKS` 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 samples 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.
- **`CHANNELS_OUT`** – The number of microphone channels to be delivered by this *`StandardPdmRxService`* instance.
- **`SUBBLOCKS`** – The number of 32-sample sub-blocks to be captured for each microphone channel.

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[*CHANNELS_IN* * *SUBBLOCKS*]`)`

Send a block of PDM data to a listener.

Parameters

`block` – PDM data to send.

void **`Init`**`(`port_t p_pdm_mics`)`

Initialize this object with a channel and port.

Parameters

`p_pdm_mics` – Port to receive PDM data on.

void **`MapChannels`**`(`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**.

TwoStageDecimator

template<unsigned **MIC_COUNT**, unsigned **S2_DEC_FACTOR**, unsigned **S2_TAP_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.
- **S2_DEC_FACTOR** – Stage 2 decimation factor.
- **S2_TAP_COUNT** – Stage 2 tap count.

Public Functions

void **Init**(const uint32_t *s1_filter_coef, const int32_t *s2_filter_coef, const right_shift_t s2_filter_shr)

Initialize the decimator.

Sets the stage 1 and 2 filter coefficients. The decimator must be initialized before any calls to *ProcessBlock()*.

`s1_filter_coef` points to a block of coefficients for the first stage decimator. This library provides coefficients for the first stage decimator; see `mic_array/etc/filters_default.h`.

`s2_filter_coef` points to an array of coefficients for the second stage decimator. This library provides coefficients for the second stage decimator where the second stage decimation factor is 6; see `mic_array/etc/filters_default.h`.

`s2_filter_shr` is the final right-shift applied to the stage 2 filter's accumulator prior to output. See lib_xcore_math's documentation of `filter_fir_s32_t` for more details.

Parameters

- **s1_filter_coef** – Stage 1 filter coefficients.
  This points to a block of coefficients for the first stage decimator. This library provides coefficients for the first stage decimator.
  See *stage1_coef*.
- **s2_filter_coef** – Stage 2 filter coefficients.
  This points to a block of coefficients for the second stage decimator. This library provides coefficients for the second stage decimator.
  See *stage2_coef*.
- **s2_filter_shr** – Stage 2 filter right-shift.
  This is the output shift used by the second stage decimator.
  See *stage2_shr*.

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

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

unsigned **`DecimationFactor`** = *S2_DEC_FACTOR*

Stage 2 decimator decimation factor.

unsigned **`TapCount`** = *S2_TAP_COUNT*

Stage 2 decimator tap count.

const uint32_t ***`filter_coef`**

Pointer to filter coefficients for Stage 1

uint32_t **`pdm_history`**[*MIC_COUNT*][8]

Filter state (PDM history) for stage 1 filters.

filter_fir_s32_t **`filters`**[*MIC_COUNT*]

Stage 2 FIR filters

int32_t **`filter_state`**[*MIC_COUNT*][*S2_TAP_COUNT*] = {{0}}

Stage 2 filter stage.

Public Static Attributes

static constexpr unsigned **`BLOCK_SIZE`** = *MIC_COUNT* * *S2_DEC_FACTOR*

Size of a block of PDM data in words.

static constexpr unsigned **`MicCount`** = *MIC_COUNT*

Number of microphone channels.

static const struct mic_array::*TwoStageDecimator*::[anonymous] **`Stage2`**

Stage 2 decimator parameters

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.
  Like many classes in this library, *FrameOutputHandler* uses the *Curiously Recurring Template Pattern*.
- **`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 usualy 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.

void **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.

Public Members

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

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

`FrameTransmitter` is the CRTP type 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.

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

Transmit the specified frame.

See *ChannelFrameTransmitter* for additional details.

Parameters

frame – Frame to be transmitted.

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.

9.2 C API Reference

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 *Decimator Stages*.

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.

Note: In version 5.0 of lib_mic_array, this value is fixed (even if you choose not to use the default filter coefficients).

STAGE1_TAP_COUNT

Macro indicating Stage 1 Filter Tap Count.

This is the number of filter taps in the first stage filter.

Note: In version 5.0 of lib_mic_array, this value is fixed (even if you choose not to use the default 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.

---

const 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 *Decimator Stages*.

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.

const int32_t **stage2_coef**[*STAGE2_TAP_COUNT*]

Stage 2 Decimation Filter Default Coefficients.

These are the default coefficients for the second stage filter.

const 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.

clock_t **clock_a**

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` is what indicates an SDR configuration is being used.

PDM_RX_RESOURCES_SDR(P_MCLK, P_PDM_CLK, P_PDM_MICS, 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.
- **CLOCK_A** – PDM clock and capture clock block resource ID.

PDM_RX_RESOURCES_DDR(P_MCLK, P_PDM_CLK, P_PDM_MICS, 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.
- **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

void **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.

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.

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 MicArray Prefabs

The MicArray prefab *mic_array::prefab::BasicMicArray* has a `bool` template parameter `USE_DCOE` which indicates whether the *mic_array::DcoeSampleFilter* should be used. If `true`, DCOE will be enabled.

With Vanilla API

When using the 'vanilla' 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 intialized 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.

mic_array_vanilla.h

void **ma_vanilla_init**()

Initializes the mic array module. (Vanilla API only)

Initializes the contexts for the decimator thread and configures the clocks and ports for PDM reception.

After calling this, the PDM clock is active and signaling, but the PDM rx service (ISR) has not yet been activated, so received PDM samples are ignored. The real-time condition is not yet active.

Parameters

- **pdm_res** – Hardware resources required by the mic array module.

void **ma_vanilla_task**(chanend_t c_frames_out)

Entry point for decimator thread and PDM rx. (Vanilla API only)

This function sets up and activates the PDM rx service in ISR mode, and then immediately begins executing the decimator.

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.