Microphone array library

The XMOS microphone array library is designed to allow interfacing to PDM microphones coupled with efficient decimation to user selectable output sample rates. Additionally, a high resolution delay can be introduced to each of the individual PDM microphones allowing for individual time shifts. This library is only avaliable for XS2 devices.

Features

The microphone array library has the following features:

- 48kHz, 24kHz, 16kHz, 12kHz and 8kHz output sample rate by default (3.072MHz PDM clock),
- 44.1kHz, 22.05kHz, 14.7kHz, 11.025kHz and 7.35kHz output sample rate by default (2.8224MHz PDM clock),
- 4, 8, 12 or 16 PDM interfaces per tile,
- No less than 80dB of stop band attenuation for all output sample frequencies,
- Configurable latency, ripple and bandwidth,
- Framing, configurable frame size from 1 sample to 8192 samples plus 50% overlapping frames option,
- Windowing and sample index bit reversal within a frame,
- Individual microphone gain compensation,
- DC offset removal,
- Up to 3.072MHz input sample rate,
- High resolution (2.63 microsecond) microphone specific delay lines,
- Every task requires only a 62.5 MIPS core to run.

Components

- PDM interface,
- Four channel decimators,
- High resolution delay block.

Software version and dependencies

This document pertains to version 2.0.0 of this library. It is known to work on version 14.1.1 of the xTIMEcomposer tools suite, it may work on other versions.

This library depends on the following other libraries:

- lib_logging (>=2.0.0)
- lib_xassert (>=2.0.0)

Related application notes

None

1 Overview

This guide is designed so that the user can understand how to use `lib_mic_array` by reading up to §13. §14 and on are designed to explain implementation details of `lib_mic_array`, but do not need to be understood to use it effectively.

Up to sixteen PDM microphones can be attached to each high channel count PDM interface (`mic_array_pdm_rx()`). One to four processing tasks, `mic_array_decimate_to_pcm_4ch()`, each process up to four channels. For 1-4 channels the library requires two logical cores:

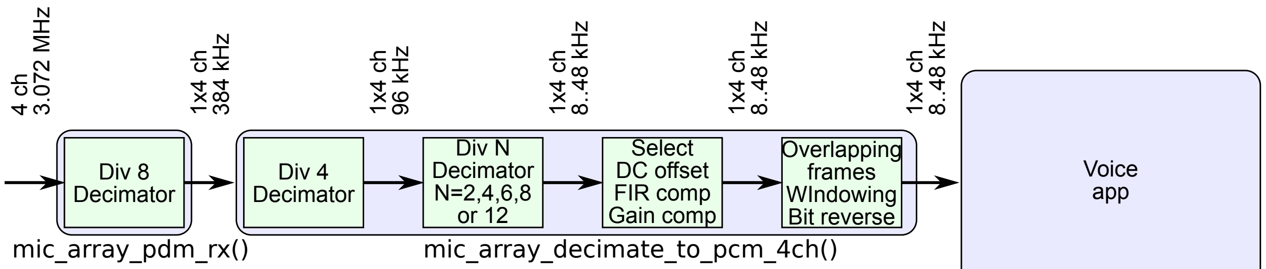


Figure 1: One to four channel count PDM interface

or for 5-8 channels three logical cores as shown below:

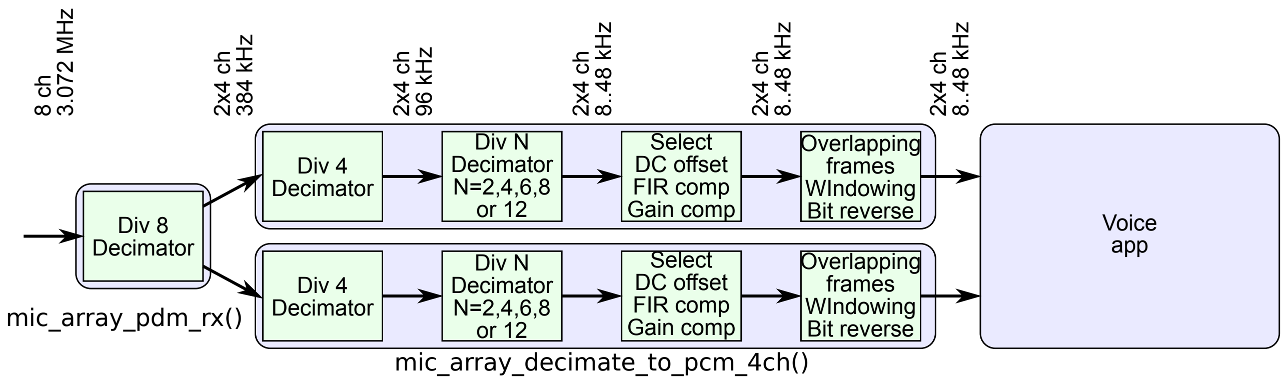

8 ch 3.072 MHz
2x4 ch 384 kHz
2x4 ch 96 kHz
2x4 ch 8..48 kHz
2x4 ch 8..48 kHz
2x4 ch 8..48 kHz
Div 8 Decimator
Div 4 Decimator
Div N Decimator N=2,4,6,8 or 12
Select DC offset FIR comp Gain comp
Overlapping frames WIndowing Bit reverse
Div 4 Decimator
Div N Decimator N=2,4,6,8 or 12
Select DC offset FIR comp Gain comp
Overlapping frames WIndowing Bit reverse
Voice app
mic_array_pdm_rx()
mic_array_decimate_to_pcm_4ch()

Figure 2: Five to eight count PDM interface

9-12 channels requires 5 logical cores and for 13-16 channels three six cores as shown below:

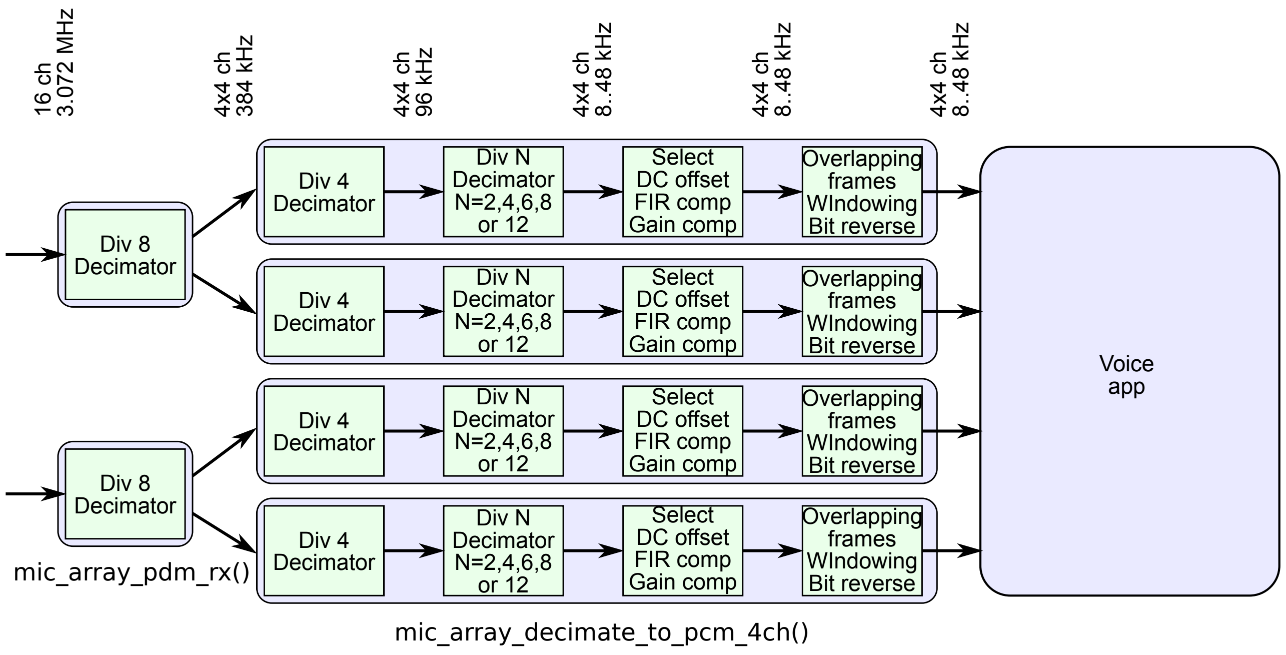


Figure 3: Thirteen to sixteen count PDM interface

The left most task, `mic_array_pdm_rx()`, samples up to 8 microphones and filters the data to provide up to eight 384 KHz data streams, split in two streams of four channels. The processing thread decimates the signal to a user chosen sample rate (one of 48, 24, 16, 12, or 8 KHz). If more than 8 channels are required then another `mic_array_pdm_rx` task can be created. After the decimation to the output sample rate the following steps takes place:

- Any DC offset is eliminated on the each channel.
- The gain is corrected so that a maximum signal on the PDM microphone corresponds to a maximum signal on the PCM signal.
- The individual gain of each microphone can be compensated. This can be used to compensate any differences in gains between the microphones in a system.
- Frames of data are generated (with a frame size of 1 to 8192 in powers of two). Frames can be set to overlap by half a frame.
- An optional windowing function is applied to each frame.
- The data can be stored in an index bit-reversed manner, so that it can be passed into an FFT without having to do any preprocessing.

There is also an optional high resolution delay, running at up to 384kHz, that can be used to add to the signal path channel specific delays. This can be used for high resolution delay and sum beamforming. The task diagrams for 4 and 8 channel microphone arrays are given in Figure 4 and Figure 5 respectivly.

Higher channel counts are simple extensions of the above task diagrams. The high resolution delay task requires a single extra core for up to 16 channels. All tasks requires a 62.5 MIPS core to run correctly, therefore, all eight cores can be used simultaneously without timing problems developing within `lib_mic_array`.

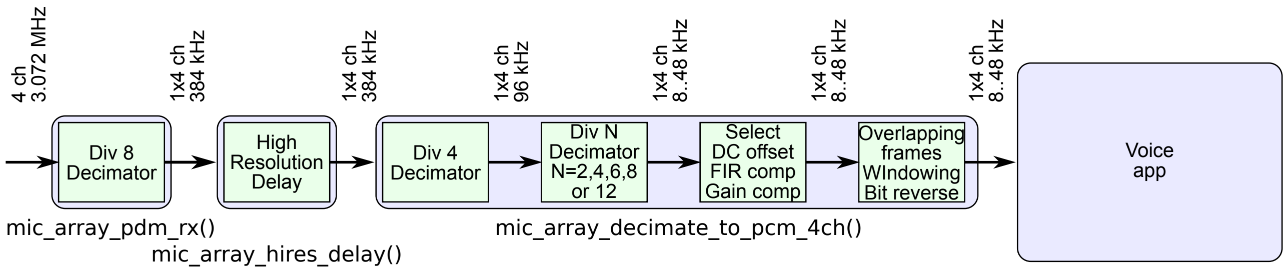


Figure 4: One to four channel count PDM interface with hires delay lines

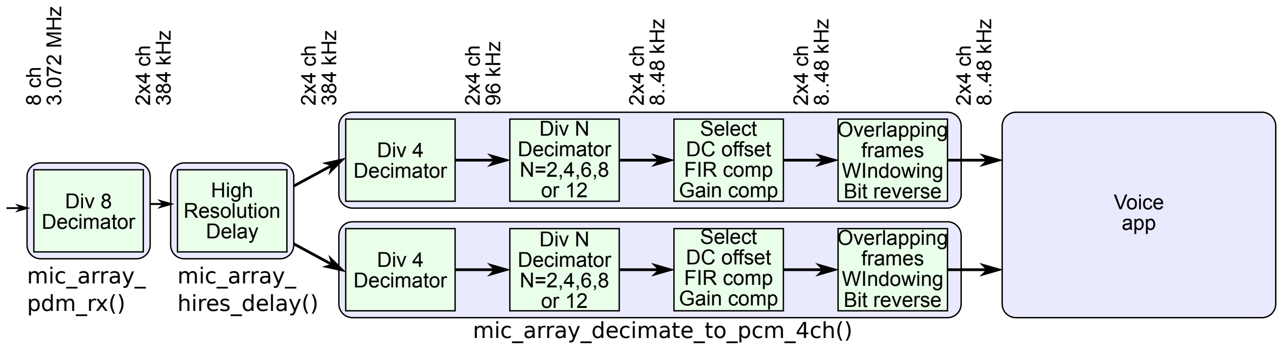


Figure 5: Five to eight count PDM interface with hires delay lines

2 Typical memory usage

The memory usage of `lib_mic_array` is mostly dependent on the desired output rates and the maximum number of channels. As lower output rates require greater decimation from the input PDM their memory requirements are proportionally greater also. Below is a table of the approximate memory usage against the channel count and decimation factor for each final stage divider.

Decimation factor	Channel count	Approx memory usage (kB)
2	4	12.1
2	8	14.1
2	12	16.1
2	16	18.1
4	4	13.1
4	8	16.1
4	12	19.1
4	16	21.1
6	4	14.1
6	8	18.1
6	12	22.1
6	16	26.1
8	4	15.1
8	8	20.1
8	12	25.1
8	16	30.1
12	4	17.1
12	8	24.1
12	12	31.1
12	16	38.1

These valuse should be use as a guide as actual memory usage may vary slightly.

3 Hardware characteristics

The PDM microphones need a *clock input* and provide the PDM signal on a *data output*. All PDM microphones share the same clock signal (buffered on the PCB as appropriate), and output onto eight data wires that are connected to a single 8-bit port:

CLOCK	Clock line, the PDM clock the used by the microphones to drive the data out.
DQ_PDM	The data from the PDM microphones on an 8 bit port.

Table 1: PDM microphone data and signal wires

The only port needed by the library is the 8-bit data port. The library assumes that the input port is clocked using the PDM clock, and the library does not know where the PDM clock comes from. If a clock block `pdmclk` is clocked at a 3.072 MHz rate, and the 8-bit port is p_pdm_mics then the following statements will ensure that the PDM data port is clocked by the PDM clock:

```
configure_in_port(p_pdm_mics, pdmclk);
start_clock(pdmclk);
```

The input clock for the microphones can be generated in a multitude of ways. For example, a 3.072MHz clock can be generated on the board, or the xCORE can divide down 12.288 MHz master clock. Or, if clock accuracy is not important, the internal 100 MHz reference can be divided down to provide an approximate clock.

If an external master clock is input to the xCORE on a 1-bit port p_`mclk` that is running at 4x the desired PDM clock, then the PDM clock can be generated by using the divider in a clock block:

```
configure_clock_src_divide(pdmclk, p_mclk, 4);
configure_port_clock_output(p_pdm_clk, pdmclk);
configure_in_port(p_pdm_mics, pdmclk);
start_clock(pdmclk);
```

An approximate clock can be generated from the 500 MHz xCORE clock as follows:

```
configure_clock_xcore(pdmclk, 82);
configure_port_clock_output(p_pdm_clk, pdmclk);
configure_in_port(p_pdm_mics, pdmclk);
start_clock(pdmclk);
```

It should be noted that this is a 3.048 MHz clock, which is 0.75% off a true 3.072 MHz clock. Finally, an approximate clock can also be generated from the 100 MHz reference clock as follows:

```
configure_clock_ref(pdmclk, 17);
configure_port_clock_output(p_pdm_clk, pdmclk);
configure_in_port(p_pdm_mics, pdmclk);
start_clock(pdmclk);
```

This gives a 2.941 MHz clock, which is 4.25% off a true 3.072 MHz clock. This may be acceptable to simple Voice User Interfaces (VUIs).

3.1 PDM microphones

PDM microphones typically have an initialization delay in the order of about 28ms. They also typically have a DC offset. Both of these will be specified in the datasheet.

4 Usage

All PDM microphone functions are accessed via the `mic_array.h` header:

```
#include <mic_array.h>
```

You also have to add `lib_mic_array` to the USED_MODULES field of your application Makefile.

A project must also include an extra header `mic_array_conf.h` which is used to describe the mandatory configuration described later in this document.

The PDM microphone interface and 4-channel decimators are instantiated as parallel tasks that run in a `par` statement. For example, in an eight channel setup the two 4-channel decimators must connect to the PDM interface via streaming channels:

```
#include <mic_array.h>

clock pdmclk;
in buffered port:32 p_pdm_mics  = XS1_PORT_8B;
in port             p_mclk      = XS1_PORT_1E;
out port            p_pdm_clk   = XS1_PORT_1F;

int main() {
      streaming chan c_pdm_to_dec[2];
      streaming chan c_ds_output[2];

      configure_clock_src_divide(pdmclk, p_mclk, 4);
      configure_port_clock_output(p_pdm_clk, pdmclk);
      configure_in_port(p_pdm_mics, pdmclk);
      start_clock(pdmclk);

      par {
          mic_array_pdm_rx(p_pdm_mics, c_pdm_to_dec[0], c_pdm_to_dec[1]);

          mic_array_decimate_to_pcm_4ch(c_pdm_to_dec[0], c_ds_output[0]);
          mic_array_decimate_to_pcm_4ch(c_pdm_to_dec[1], c_ds_output[1]);

          application(c_ds_output);
      }
      return 0;
}
```

There is a further requirement that any application of a `mic_array_decimate_to_pcm_4ch()` task must be on the same tile as the `mic_array_decimate_to_pcm_4ch()` task due to the sharaed frame memory.

As the PDM interface `mic_array_pdm_rx()` communicates over channels then the placement of it is not restricted to the same tile as the decimators.

Additionally, the high resolution delay task can be inserted between the PDM interface and the decimators in the following fashion:

```
#include <mic_array.h>

clock pdmclk;
in buffered port:32 p_pdm_mics  = XS1_PORT_8B;
in port             p_mclk      = XS1_PORT_1E;
out port            p_pdm_clk   = XS1_PORT_1F;

int main() {
      streaming chan c_pdm_to_hires[2];
      streaming chan c_hires_to_dec[2];
      streaming chan c_ds_output[2];
      chan c_cmd;

      configure_clock_src_divide(pdmclk, p_mclk, 4);
      configure_port_clock_output(p_pdm_clk, pdmclk);
      configure_in_port(p_pdm_mics, pdmclk);
      start_clock(pdmclk);

      par {
          mic_array_pdm_rx(p_pdm_mics, c_pdm_to_hires[0], c_pdm_to_hires[1]);

          mic_array_hires_delay(c_pdm_to_hires, c_hires_to_dec, 2, c_cmd);

          mic_array_decimate_to_pcm_4ch(c_hires_to_dec[0], c_ds_output[0]);
          mic_array_decimate_to_pcm_4ch(c_hires_to_dec[1], c_ds_output[1]);

          application(c_ds_output, c_cmd);
      }
      return 0;
}
```

Note, using the high resolution delay consumes an extra logical core.

5 High resolution delay task

The high resolution delay task, mic_array_hires_delay(), is capable of implementing delays with a resolution down to 2.3 microseconds (384kHz). It implements up to 16 delays lines of length MIC_ARRAY_HIRES_MAX_DELAY, which has a default of 256. The delay line length can be overridden by redefining it in mic_array_conf.h. Each delay line sample is clocked at the PDM clock rate divided by 8, that is, 384kHz for a 3.072MHz PDM clock and 352.8kHz for an PDM clock of 2.8224MHz.

By setting a positive delay of N samples on a channel then an input sample will take N extra clocks to propagate to the decimators. Setting of the taps is done through the function mic_array_hires_delay_set_taps() which will do an atomic update of all the active delay lines tap positions. The default delay on each channel is zero. When the high resolution delay task is in use the define MIC_ARRAY_HIRES_MAX_DELAY should be minimised for the application specific requirements as longer delay lines require more memory.

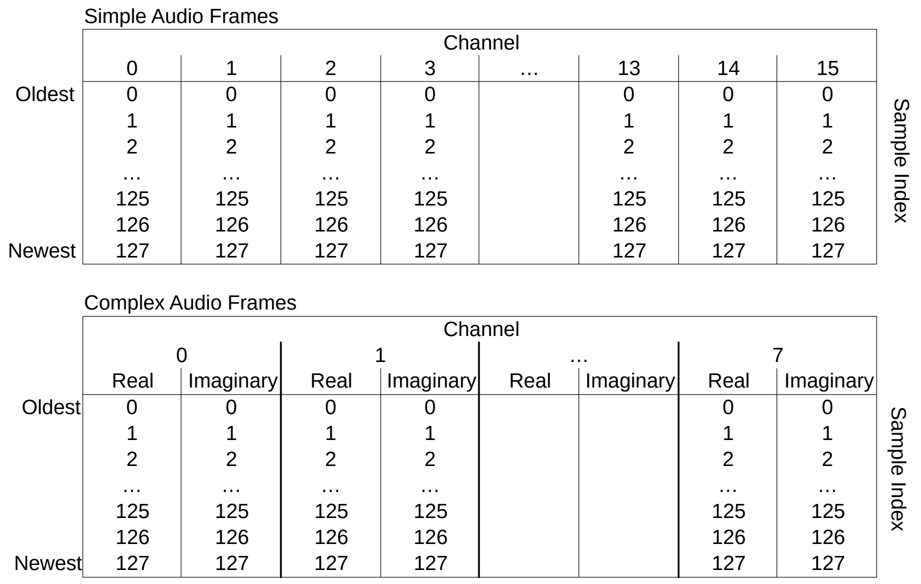
See §16 for the API.

6 Accessing the samples

Samples are accessed in the form of frames. A frame is returned from the decimators in either the time domain format, `mic_array_frame_time_domain`, or in the FFT ready format, `mic_array_frame_fft_preprocessed`.

Time domain frames contain a single two-dimensional array, `data`,with the first dimension being the channel ID and the second dimension being the sample number. Samples are ordered 0 as the oldest sample and increasing number being newer.

FFT ready frames also contain a single two-dimensional array, `data`. The data is preprocessed by the decimators in such a way that the frames that are ready for direct processing by an DIT FFT.


Simple Audio Frames

	Channel								
	0	1	2	3	…	13	14	15	
Oldest	0	0	0	0		0	0	0	Sample Index
	1	1	1	1		1	1	1	
	2	2	2	2		2	2	2	
	…	…	…	…		…	…	…	
	125	125	125	125		125	125	125	
	126	126	126	126		126	126	126	
Newest	127	127	127	127		127	127	127	

Complex Audio Frames

	Channel								
	0		1		…		7		
	Real	Imaginary	Real	Imaginary	Real	Imaginary	Real	Imaginary	
Oldest	0	0	0	0			0	0	Sample Index
	1	1	1	1			1	1	
	2	2	2	2			2	2	
	…	…	…	…			…	…	
	125	125	125	125			125	125	
	126	126	126	126			126	126	
Newest	127	127	127	127			127	127	

Figure 6: Memory layout of simple audio and complex frames.

Frames in the `mic_array_frame_fft_preprocessed` are not intended to be directly accessed by a user. Instead when the frame has been processed by the FFT and cast to a `mic_array_frame_frequency_domain` then the data can be manipulated.

7 Frames

The four channel decimators (`mic_array_pdm_rx()`) output frames of either *time domain audio* or *FFT ready audio* prepared for an FFT. The define MIC_ARRAY_MAX_FRAME_SIZE_LOG2 (found in `mic_array_conf.h`) should be used to allocate the arrays to store the frames. This means that all frames structures will allocate enough memory to allow for a frame size of two to the power of MIC_ARRAY_MAX_FRAME_SIZE_LOG2 regardless of the size used in the decimator_config_common. It is recommended that the `frame_size_log2` field of decimator_config_common is always set to MIC_ARRAY_MAX_FRAME_SIZE_LOG2. Equally, the define MIC_ARRAY_NUM_MICS is used for allocating the memory for the frame structure. This must be set to a multiple of 4.

All frame types contain a two-dimensional `data` array.

For simplicity of reading `M` will represent two to the power of MIC_ARRAY_MAX_FRAME_SIZE_LOG2 and F will represent two to the power of `frame_size_log2`.

7.1 Time domain frames

If *time domain audio* output is used (`index_bit_reversal` is set to 0), then data is stored into arrays in real time ordering. The arrays are of length `M` where the first F (see §16) entries contain valid data. All entries between F and `M` are undefined. The first index of the data element of the structure `mic_array_frame_time_domain` is used to address the channel and the second index is used for the sample number with zero being the oldest sample.

Frames are initialised by the application with a call to `mic_array_init_time_domain_frame()`. Pass it:

- `c_from_decimators`: An array of channels to the decimators.
- `decimator_count`: A count of the number of decimators (the number of elements in the above array).
- `buffer`: used internally to maintain ownership of the shared memory between the application and the decimators.
- `audio`: the array of audio frames, one (or two) of which will be owned by the decimators at all times.
- `dc`: the configuration array to the decimators.

Calls to `mic_array_get_next_time_domain_frame()` should be made to retrieve subsequent audio frames. These calls require the exact same parameters as `mic_array_init_time_domain_frame()`.

7.2 FFT ready audio

If *FFT ready audio* output is used (`index_bit_reversal` is set to 1), then the data is stored in frames that are designed to be processed with an FFT. The data is stored in arrays of length `M` where the first two F entries contain valid data, each element storing a real and an imaginary part. The data is stored in a bit reversed order (ie, the oldest element is at index 0b0000....0000, the next oldest is at element 0b1000...0000, the next one at element 0b0100...0000, etc up to element 0b1111...1111), and the real elements store the even channels, whereas the imaginary elements store the odd channels. A postprocess function must be applied after the Decimate-in-Time (DIT) FFT in order to recover the frequency bins.

Frames are initialised by the application by a call to `mic_array_init_frequency_domain_frame()`. Pass it:

- `c_from_decimator`: An array of channels to the decimators
- `decimator_count`: A count of the number of decimators (the number of elements in the above array).
- `buffer`: used internally to maintain ownership of the shared memory between the application and the decimators.
- `f_complex`: the array of complex frames, one (or two) of which will be owned by the decimators at

all times.
- dcc: the configuration to the decimators.

Calls to `mic_array_get_next_frequency_domain_frame()` should be made to retrieve subsequent audio frames. These calls require the exact same parameters as `mic_array_init_frequency_domain_frame()`.

8 Using the decimators

The decimators reduce the high rate PCM down to lower rate PCM. They also prepare the audio for subsequenct algorithms, i.e. framing, windowing, etc.

8.1 Setting up the decimators

All decimators attach to an application via streaming channels and are configured simultaneously with the `mic_array_decimator_configure()` function. The parameters to the `mic_array_decimator_configure()` function are described in a § 16. To start the frame exchange process `mic_array_init_frequency_domain_frame()` or `mic_array_init_time_domain_frame()` must be called. Now the decimators are running and will be outputting frames at the rate given by their configuration.

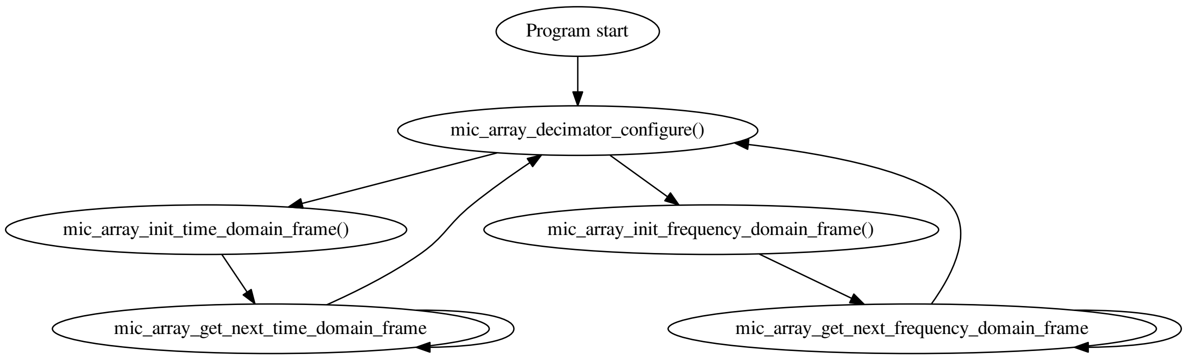

Program start
mic_array_decimator_configure()
mic_array_init_time_domain_frame()
mic_array_init_frequency_domain_frame()
mic_array_get_next_time_domain_frame
mic_array_get_next_frequency_domain_frame

Figure 7: Order of the function calls allowed to the decimators.

The configuration of the decimators can be changed at any time so long as the function calls respect the control flow given in Figure 7. Mixing of time and frequency domain functions is not supported without first calling `mic_array_decimator_configure()`.

8.2 Changing decimator configuration

Once the decimators are running the configuration of the decimators remains constant. If a change of configuration is required then a call to `mic_array_decimator_configure()` allows a complete reconfigure. This will reconfigure and reset all attached decimators. The only configuration that will survive reconfiguration is the DC offset memory. It is assumed that the microphone specific DC offset remains fairly constant between reconfigurations.

9 mic_array_conf.h

An application that uses `lib_mic_array` must define the header file `mic_array_conf.h`. This header must define:

- MIC_ARRAY_MAX_FRAME_SIZE_LOG2
  This defines the maximum frame size (log 2) that the application could request to use. The application may request frame sizes from 0 to `MIC_ARRAY_MAX_FRAME_SIZE_LOG2`. This should be kept small as it governs the memory required for a frame.
- MIC_ARRAY_NUM_MICS
  This defines the number of microphones in use. It is used for allocating memory in the frame structures.

Optionally, `mic_array_conf.h` may define

- MIC_ARRAY_DC_OFFSET_LOG2
  The DC offset is removed with a high pass filter. DC_OFFSET_DIVIDER_LOG2 can be used to control the responsiveness of the filter vs the cut off frequency. The default is 13, but setting this will override it. The value must not exceed 31. See §14 DC offset removal for further explanation.
- MIC_ARRAY_MIC_ARRAY_HIRES_MAX_DELAY
  This defines the length of the high resolution delay lines. This should be set to a power of two for efficiency. The default is 256. Increasing values will result in increasing memory usage.
- MIC_ARRAY_WORD_LENGTH_SHORT
  If this define is set to non-zero then this configures the output word length to be a 16 bit short otherwise its left as 32 bit word length output. All internal processing will be done at 32 bits, only during the write to frame memory will the truncation happen.

10 Four Channel Decimator

The four channel decimator tasks are highly configurable tasks for outputting frames of various sizes and formats. They can be used to produce frames suitable for time domain applications or pre-process the frames ready for an FFT for frequency domain applications. The four channel decimators, `mic_array_decimate_to_pcm_4ch()`, have a number of configuration options controlled by the structure `decimator_config` through the function `mic_array_decimator_configure()`. The decimators are controlled by two structures: `decimator_config_common` and `decimator_config`, where the former configuration is common to all microphones and the later is specific to the batch of 4 microphones it interfaces to. The application has the option to control the following settings through `decimator_config_common`:

- `frame_size_log2`: This sets the frame size to a power of two. A frame will contain 2 to the power of frame_size_log2 samples of each channel. Set this to a maximum of MIC_ARRAY_MAX_FRAME_SIZE_LOG2.
- `apply_dc_offset_removal`: This controls if the DC offset removal should be enabled or not. Set to non-zero to enable, or 0 to not apply DC offset removal.
- `output_decimation_factor`: This specifies the decimation factor to apply to the PDM input after an 8x decimtor and 4x decimator has already been applied, i.e. for s 3.072MHz PDM clock the `output_decimation_factor` will apply to a 96kHz sample rate. The valid values are 2, 4, 6, 8 and 12. Common sample rates can be achieved by using these decimation factors as follows:

output_decimation_factor	**PDM clock**	**Sample rate**
2	3.072 MHz	48 KHz
4	3.072 MHz	24 KHz
6	3.072 MHz	16 KHz
8	3.072 MHz	12 KHz
12	3.072 MHz	8 KHz
2	2.8224 MHz	44.1 KHz
4	2.8224 MHz	22.05 KHz

  For other decimation factors see §15.5.
- `coefs`: This is a pointer to an array of arrays containing the coefficients for the final stage of decimation. Set this to FIR_LUT(d) where d is the output_decimation_factor; FIR_LUT() is defined in `fir_decimator.h`. If you wish to supply your own FIR coefficients; the array should have the same number of entries as `output_decimation_factor`.
- `fir_gain_compensation`: single value to compensate the gain of all the previous decimators. This must be set to a value that depends on the `output_decimation_factor` as follows:

output_decimation_factor	**fir_gain_compensation**
2	FIR_COMPENSATOR_DIV_2
4	FIR_COMPENSATOR_DIV_4
6	FIR_COMPENSATOR_DIV_6
8	FIR_COMPENSATOR_DIV_8
12	FIR_COMPENSATOR_DIV_12

  If you wish to supply your own, this is a fixed point number in 5.27 format. To apply a unity gain set to 0.

- apply_mic_gain_compensation: Set this to 1 if microphone gain compensation is required. The compensation applied is controlled through the mic_gain_compensation array in decimator_config below.
- A windowing function can be passed in through windowing_function. It is a pointer to an array of integers that defines the windowing operator. Each sample in the frame is multiplied by its associated window value and shifted right by 31 places. This is performed before any index bit reversal (see the next entry). The window function data is in 1.31 fixed point format and only the first half of the window function is required.
- If the data is going to be post processed using an FFT, then index_bit_reversal can be set to 1. This will store the data elements reordered according to a reversed bit pattern, suitable for an FFT without "index bit reversing". As a side effect, it stores the data as complex numbers, in such a way that a single complex FFT operates on two microphones in parallel.
- buffering_type: DECIMATOR_HALF_FRAME_OVERLAP is used to specify half frame overlapping or sequential frames is selected with DECIMATOR_NO_FRAME_OVERLAP.
- number_of_frame_buffers: is used to specify the number of frames used by the application plus decimators. This number should be at least two when DECIMATOR_NO_FRAME_OVERLAP is in effect or three when DECIMATOR_HALF_FRAME_OVERLAP is in effect. This is due to the double buffered nature of the decimators, i.e. the decimators are writing to (one or two) frames whilst the application is using at least one.

decimator_config configures the per-channel information:

- dcc: This is a pointer to the common decimator configuration.
- data: This is the memory used to save the FIR samples. It must be an array of size (4 channels x THIRD_STAGE_COEFS_PER_STAGE x sizeof(int) x output_decimation_factor bytes).
- mic_gain_compensation: This is an array with four elements specifying the relative compensation to apply to each microphone. Unity gain is given by the value INT_MAX. To equalise the gain of all microphones, the quietest microphone should be given unity gain, and the gain of all other microphones should be set proportionally lower.
- channel_count: this is the number of channels that is enabled. Set this to 4 to enable all channels. If set to a value less than 4, only the first channel_count channels are enabled.

The decimator configuration is applied by calling the function mic_array_decimator_configure() with an array of chanends referring to the decimators, a count of the number of decimators, and an array of decimator configurations.

The output of the decimator is 32-bit or 16-bit(MIC_ARRAY_WORD_LENGTH_SHORT != 0) PCM audio at the requested sample rate.

11 Intended usage model

The library has been designed with the intention of being able to dynamically change its configuration, however, for minimal memory footprint choosing a single output rate means the fewest FIR coefficient end up in the binary. A typical code structure will contain the following:

```
unsigned buffer;
mic_array_init_time_domain_frame(c_ds_output, 2, buffer, audio, dc);

while(1){
  mic_array_frame_time_domain *  latest_frame = mic_array_get_next_time_domain_frame(c_ds_output, 2, buffer,
     ↪ audio, dc);

}
```

When a reconfigure is performed then there will be a short interval (to flush the FIR data buffers) before the audio continues.

Overlapping frames are supported so that frequency domain algorithms can be converted back into the time domain without artifacts. See `lib_dsp` for FFT functions.

The `number_of_frame_buffers` member of `decimator_config_common` is required so that a frame buffer (array) can be used in a round-robin fashion. This means that the when the application calls either `mic_array_init_time_domain_frame()` or `mic_array_init_frequency_domain_frame()` then the ownership of one or two of the fames (depending on the overlapping scheme) will be passed to the decimators. When a decimator has finished writing the oldest frame it is returned to the application and the next in line is sent to the decimators. This means that by declaring larger frame buffers and increasing `number_of_frame_buffers` then the application can have visibility of longer periods of time at the expense of memory.

Due to the round-robin nature of the library the application must be finished with the data in the oldest frame before the decimators need it again. This is the nature of real time audio processing.

12 FIR memory

For each decimator a block of memory must be allocated for storing FIR data. The size of the data block must be:

```
Number of channels * THIRD_STAGE_COEFS_PER_STAGE * Decimation factor * sizeof(int)
```

bytes. The data must also be double word aligned. For example, if the decimation factor was set to DECIMATION_FACTOR and two decimators were in use, then the memory allocation for the FIR memory would look like:

```
int data[CHANNELS][THIRD_STAGE_COEFS_PER_STAGE*DECIMATION_FACTOR];
```

The FIR memory must also be initialized in order to prevent a spurious click during startup. Normally initializing to all zeros is sufficient. Memset is a highly efficient way of doing this.

Note, globally declared memory is always double word aligned.

13 Example Applications

Two stand alone applications showing the minimum code required to build a functioning microphone array are given in AN00217_app_high_resolution_delay_example and in AN00220_app_phase_aligned_example.

A worked example of a fixed beam delay and sum beamformer is given in the application AN00219_app_lores_DAS_fixed. Also examples of of how to set up high resolution delayed sampling can be seen in the high resolution fixed beam delay and sum beamformer given in the application AN00218_app_hires_DAS_fixed.

14 DC offset removal

The DC offset removal is implemented as a single pole IIR filer obeying the relation:

```
Y[n] = Y[n-1] * alpha + x[n] - x[n-1]
```

Where `alpha` is defined as `1 - 2^MIC_ARRAY_DC_OFFSET_LOG2`. Increasing `MIC_ARRAY_DC_OFFSET_LOG2` will increase the stability of the filter and decrease the cut off point at the cost of increased settling time. Decreasing `MIC_ARRAY_DC_OFFSET_LOG2` will increase the cut off point of the filter.

15 Signal Characteristics

15.1 Definition of terms

15.2 Passband

This specifies the bandwidth, from DC, in Hz over which a signal will not be attenuated by more than 3dB.

15.3 Stopband

This specifies the start frequency to the input Nyquest sample rate that the input signal should be attenuated over.

15.4 Characteristics

The output signal has been decimated from the original PDM in such a way to introduce no more than -80dB of noise into the passband for all output sample rates.

output_decimation_factor	PDM Sample Rate(Hz)	Output sample rate(Hz)
2	3072000	48000
4	3072000	24000
6	3072000	16000
8	3072000	12000
12	3072000	8000
2	2822400	44100
4	2822400	22050
6	2822400	14700
8	2822400	11025
12	2822400	7350

output_decimation_factor	Passband(Hz)	Stopband(Hz)	Ripple(dB)	THD+N(dB)
2	18240	24000	1.93	-144.63
4	9600	12000	0.64	-142.61
6	6400	8000	0.37	-139.10
8	4800	6000	0.24	-136.60
12	3200	4000	0.18	-133.07
2	16758	22050	1.93	-144.63
4	8820	11025	0.64	-142.61
6	5880	7350	0.37	-139.10
8	4410	5512.5	0.24	-136.60
12	2940	3675	0.18	-133.07

The decimation is achieved by applying three poly-phase FIR filters sequentially. The design of these filters can be viewed in the python script `fir_design.py`. The default magnitude responses of the first to third stages are given as Figure 13 through to Figure 17 in the appendix. The first stage and second stage can be viewed in Figure 11 and Figure 12.

The phase delay of the default filters is 18 output clock cycles. This can be shortened by either using a minimum phase FIR as the final stage decimation FIR and/or by reducing the number of taps on the final stage decimation FIR.

15.5 Advanced filter design

The above table has been generated to provide 80dB of stopband attenuation for all decimation factors whilst maintaining a fairly flat passband and wide bandwidth. However for a given specification the filter characteristics can be optimised to reduce latency, increase passband, lower the passband ripple and increase the signal to noise ratio. For example, in a system where a 16kHz output is required then limiting the passband to 8kHz would improve the other properties. Equally, if the noise floor of the PDM microphone is 65dB then there is little advantage exceeding that in the filter.

15.6 `fir_design.py` usage

In order generate custom filters the `fir_design.py` can be executed. The purpose of this script is to design and generate the FIR coefficients for the three stages of decimation. `fir_design.py` is a command line tool that takes a number of options to control each parameter of the filter design. As previously illustrated the PDM to PCM conversion is divided into three stages. The overall noise floor is governed with the option `--stopband-attenuation`. This should be a positive number of decibels between 20 and 120. In the first stage the designer is able to tune:

- passband bandwidth (`--first-stage-pass-bw`) - The bandwidth of the passband, in kHz.
- stopband bandwidth (`--first-stage-stop-bw`) - The bandwidth of the bands around the regions that will alias with the pass band after decimation, in kHz.

These are illustrated in Figure 8.

In the second stage the same options are available:

- passband bandwidth (`--second-stage-pass-bw`) - The bandwidth of the passband, in kHz.
- stopband bandwidth (`--second-stage-stop-bw`) - The bandwidth of the bands around the regions that will alias with the pass band after decimation, in kHz.

These are illustrated in Figure 9.

In the third stage the designer can provide custom decimation factors in addition to the pass and stop band parameters. Also the delay of the filter can be controlled by tuning the number of taps to allocate for each phase of the poly-phase FIR (`--third-stage-num-taps`). The fewer the number of taps per phase then the shorter the delay of the filter but the harder the design will be to meet other criteria.

To add a custom third stage filter `--add-third-stage` has to be called. It requires the following arguments:

- decimation factor - the ratio of input samples to output samples.
- normalized output passband - this specifies where the passband ends.
- normalized output stopband - this specified where the stopband starts.
- filter_name - this assigns a name to the custom filter.

For example to add a third stage decimator called “my_filter” with a final stage decimation factor of 2, normalized output passband of 0.4 and normalized output stopband of 0.5 then the argument `--add-third-stage 2 0.4 0.5 my_filter` would need to be passed to the script.

These are illustrated in figure Figure 10.

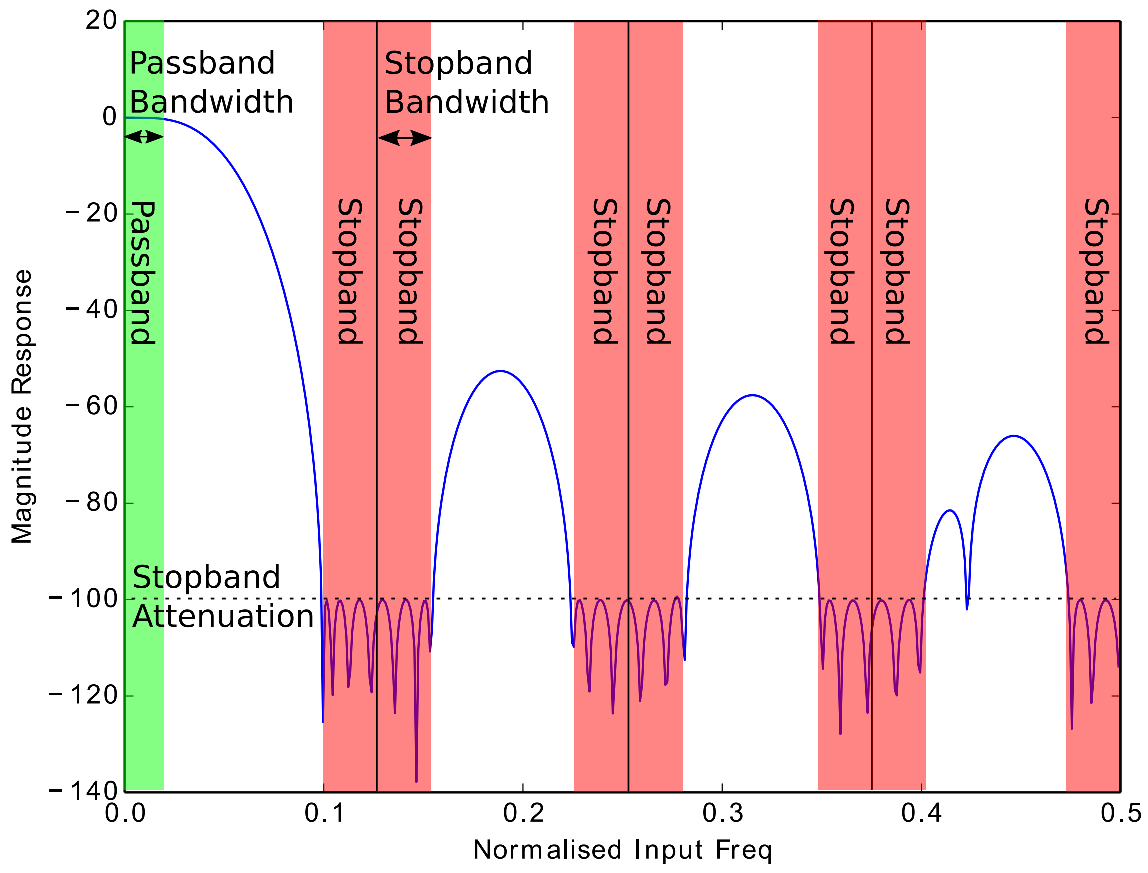


Figure 8: First stage design parameters.

The filter name is used to generate the defines and coefficient arrays used to implement the filter in lib_mic_array. The defines DECIMATION_FACTOR_ + (filter_name) and FIR_COMPENSATOR_ + (filter_name) will be generated to represent the filter designed. In the generated defines the name will be in all caps and the FIR coefficients array will be in all lowercase. Additionally, the array const int g_third_stage_ + (filter_name) _fir[] will also be generated and will contain all the coefficients to implement the filter. For example, if fir_design.py was passed the option --add-third-stage 2 0.4 0.5 my_filter then available in lib_mic_array would be:

```
#define DECIMATION_FACTOR_MY_FILTER (2)
#define FIR_COMPENSATOR_MY_FILTER (301451293)
extern const int g_third_stage_my_filter_fir[126];
```

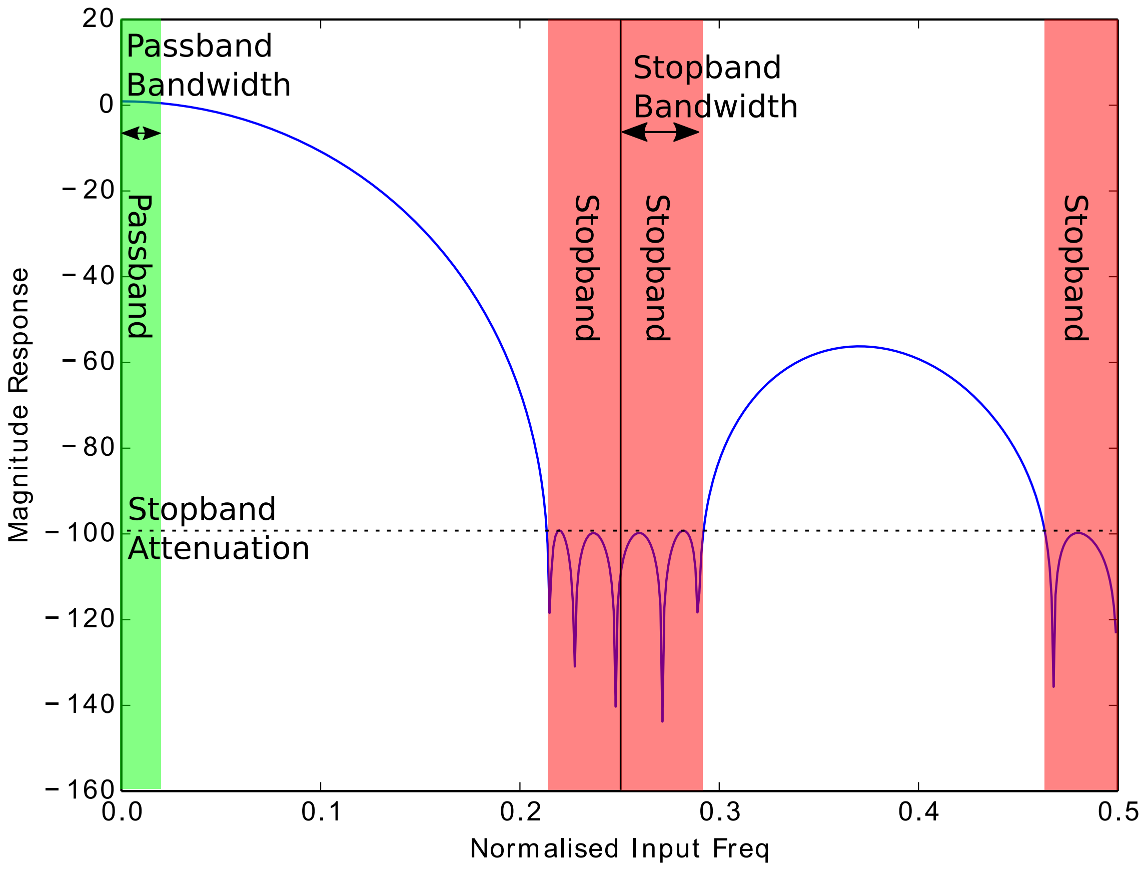


Figure 9: Second stage design parameters.

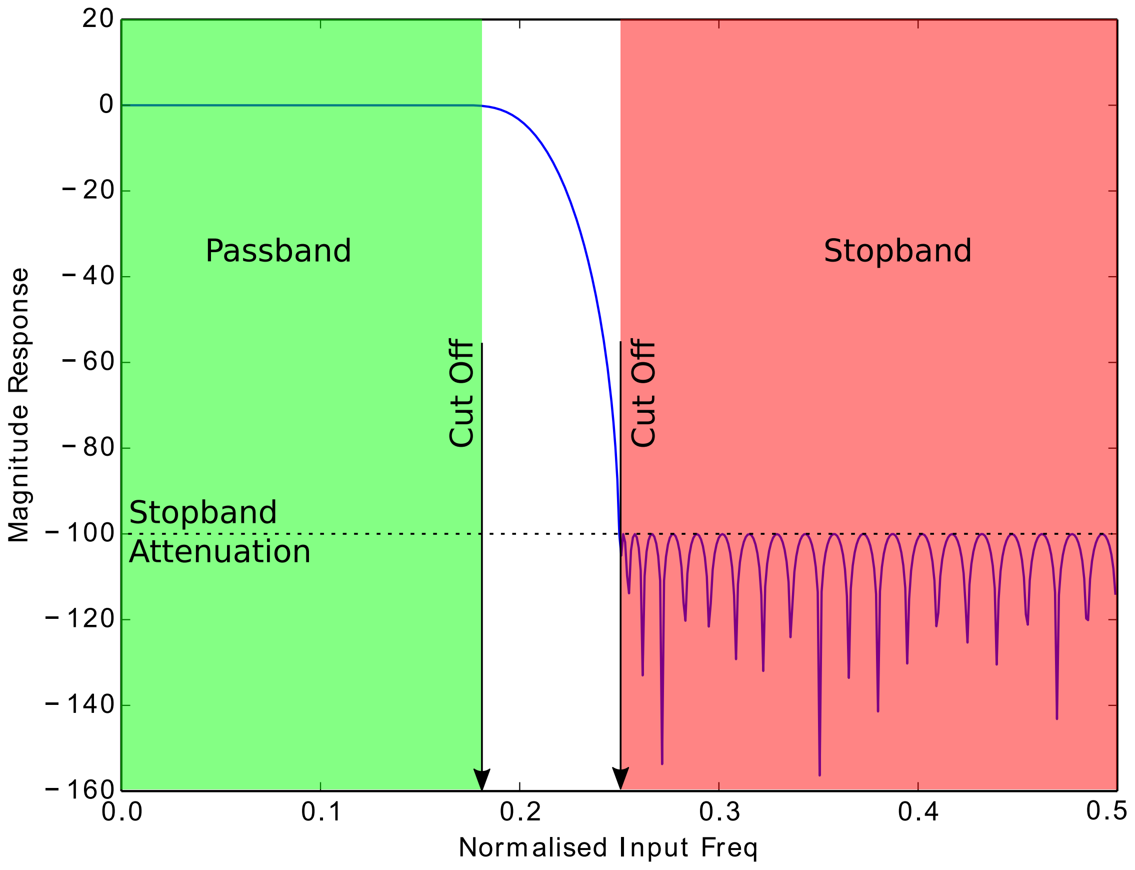


Figure 10: Third stage design parameters.

16 API

16.1 Creating an PDM microphone interface instance

Function	**mic_array_pdm_rx**
Description	PDM Microphone Interface component. This task handles the interface to up to 8 PDM microphones whilst also decimating the PDM data by a factor of 8. The output is sent via two channels to two receiving tasks.

Type	`void`
	`mic_array_pdm_rx(in buffered port:32 p_pdm_mics,`
	`   streaming chanend c_4x_pdm_mic_0,`
	`   streaming chanend ?c_4x_pdm_mic_1)`

Parameters	`p_pdm_mics`
	The 8 bit wide port connected to the PDM microphones.
	
	`c_4x_pdm_mic_0`
	The channel where the decimated PDM of microphones 0-3 will be outputted.
	
	`c_4x_pdm_mic_1`
	The channel where the decimated PDM of microphones 4-7 will be outputted. This can be null for 4 channel output.

16.2 PDM microphone processing

Function	mic_array_decimate_to_pcm_4ch
Description	Four Channel Decimation component. This task decimates the four channel input down to the desired output sample rate. The decimator has a fixed divide by 4 followed by a divide by decimation_factor where decimation_factor is greater than or equal to 2. The channel c_frame_output is used to transfer data and control information between the application and this task. It relies of shared memory for so the client of this task must be on the same tile as this task.

Type	`void`
	`mic_array_decimate_to_pcm_4ch(`
	`   streaming chanend c_from_pdm_interface,`
	`   streaming chanend c_frame_output)`

Parameters	`c_from_pdm_interface`
	The channel where the decimated PDM from pdm_rx task will be inputted.
	
	`c_frame_output`
	The channel used to transfer data and control information between the client of this task and this task.

Function	mic_array_decimator_configure
Description	Decimator configuration. This function initializes the decimators and configures them as per the decimator configuration structure thay are passed.

Type	`void`
	`mic_array_decimator_configure(`
	`   streaming chanend c_from_decimators[],`
	`   unsigned decimator_count,`
	`   mic_array_decimator_config_t dc[])`

Parameters	`c_from_decimators`
	The channels used to transfer pointers between the application and the mic_array_decimate_to_pcm_4ch() task.
	
	`decimator_count`
	The count of mic_array_decimate_to_pcm_4ch() tasks.
	
	`dc` The array cointaining the decimator configuration for each decimator.

Type	**mic_array_decimator_config_t**
Description	Configuration structure unique to each of the 4 channel deciamtors. This contains configuration that is channel specific, i.e. Gain compensation, etc.

Fields	`mic_array_decimator_conf_common_t *unsafe dcc`
	
	`int *unsafe data`
	The data for the FIR decimator.
	
	`int mic_gain_compensation`
	An array describing the relative gain compensation to apply to the microphones.
	The microphone with the least gain is defined as 0x7fffffff (INT_MAX), all others are given as INT_MAX*min_gain/current_mic_gain.
	
	`unsigned channel_count`
	The count of enabled channels (0->4).

Type	**mic_array_decimator_conf_common_t**
Description	Four Channel decimator configuration structure. This is used to describe the configuration that the group of synchronous decimators will use to process the PCM audio.

Fields	`unsigned frame_size_log2`
	The output frame size log2, i.e.
	A frame will contain 2 to the power of frame_size_log2 samples of each channel.
	
	`int apply_dc_offset_removal`
	Remove the DC offset from the audio before the final decimation.
	Set to non-zero to enable.
	
	`int index_bit_reversal`
	If non-zero then bit reverse the index of the elements within the frame.
	Used in the case of preparing for an FFT.
	
	`int *unsafe windowing_function`
	If non-null then this will apply a windowing function to the frame.
	Used in the case of preparing for an FFT.

Continued on next page

	`unsigned output_decimation_factor` Final stage FIR Decimation factor.
	`const int *unsafe coefs` The coefficients for the FIR decimator.
	`int apply_mic_gain_compensation` Set to non-zero to apply microphone gain compensation.
	`int fir_gain_compensation` 5.27 format for the gain compensation for the three stages of FIR filter.
	`mic_array_decimator_buffering_t buffering_type` The buffering type used for frame exchange.
	`unsigned number_of_frame_buffers` The count of frames used between the decimators and the application.

16.3 PCM frame interfacing

Type	**mic_array_decimator_buffering_t**
Description	Four Channel decimator buffering type. This type is used to describe the buffering mode. Note: to use a windowing function the constant-overlap-and-add property must be obeyed, i.e. Coef[n] = 1-Coef[N-n] where N is the array length. Only half the array need be specified as the windowing function is assumed to be symmetric.

Values	`DECIMATOR_NO_FRAME_OVERLAP`
	The frames have no overlap.
	
	`DECIMATOR_HALF_FRAME_OVERLAP`
	The frames have a 50% overlap betweeen sequential frames.

Function	**mic_array_init_time_domain_frame**
Description	Four Channel Decimation initializer for raw audio frames. This function call sets up the four channel decimators. After this has been called there will be a real time requirement on this task, i.e. this task must call mic_array_get_next_time_domain_frame() at the output sample rate multiplied by the frame size.

Type	`void`
	`mic_array_init_time_domain_frame(`
	`   streaming chanend c_from_decimators[],`
	`   unsigned decimator_count,`
	`   unsigned &buffer,`
	`   mic_array_frame_time_domain audio[],`
	`   mic_array_decimator_config_t dc[])`

Parameters	`c_from_decimators`
	The channels used to transfer pointers between the application and the mic_array_decimate_to_pcm_4ch() tasks.
	
	`decimator_count`
	The count of mic_array_decimate_to_pcm_4ch() tasks.
	
	`buffer` The buffer index. Always points to the index that is accessible to the application (initialized internally)
	
	`audio` An array of audio frames.
	
	`dc` The array cointaining the decimator configuration for each decimator.

Function	mic_array_get_next_time_domain_frame
Description	Four Channel Decimation raw audio frame exchange function. This function handles the frame exchange between the mic_array_decimate_to_pcm_4ch() tasks and the application. It returns a pointer to the most recently written frame. After this point the oldest frame is assumed out of scope to the application.

Type	`mic_array_frame_time_domain* alias`
	`mic_array_get_next_time_domain_frame(`
	`   streaming chanend c_from_decimators[],`
	`   unsigned decimator_count,`
	`   unsigned &buffer,`
	`   mic_array_frame_time_domain *alias audio,`
	`   mic_array_decimator_config_t dc[])`

Parameters	`c_from_decimators` The channels used to transfer pointers between the application and the mic_array_decimate_to_pcm_4ch() tasks. `decimator_count` The count of mic_array_decimate_to_pcm_4ch() tasks. `buffer` The buffer index (Used internally) `audio` An array of audio frames. `dc` The array cointaining the decimator configuration for each decimator.
Returns	A pointer to the frame now owned by the application. That is, the most recently written samples.

Function	mic_array_init_frequency_domain_frame
Description	Four Channel Decimation initializer for complex frames. This function call sets up the four channel decimators. After this has been called there will be a real time requirement on this task, i.e. this task must call mic_array_get_next_frequency_domain_frame() at the output sample rate multiplied by the frame size.

Type	`void`
	`mic_array_init_frequency_domain_frame(`
	`   streaming chanend c_from_decimators[],`
	`   unsigned decimator_count,`
	`   unsigned &buffer,`
	`   mic_array_frame_fft_preprocessed f_fft_preprocessed[],`
	`   mic_array_decimator_config_t dc[])`

Continued on next page

Parameters	`c_from_decimators` The channels used to transfer pointers between the application and the mic_array_decimate_to_pcm_4ch() tasks. `decimator_count` The count of mic_array_decimate_to_pcm_4ch() tasks. `buffer` The buffer index. Always points to the index that is accessible to the application (initialized internally) `f_fft_preprocessed` An array of complex frames. `dc` The array cointaining the decimator configuration for each decimator.

Function	**mic_array_get_next_frequency_domain_frame**
Description	Four Channel Decimation complex frame exchange function. This function handles the frame exchange between the mic_array_decimate_to_pcm_4ch() tasks and the application. It returns a pointer to the most recently written frame. After this point the oldest frame is assumed out of scope to the application.

Type	`mic_array_frame_fft_preprocessed* alias`
	`mic_array_get_next_frequency_domain_frame(`
	`   streaming chanend c_from_decimators[],`
	`   unsigned decimator_count,`
	`   unsigned &buffer,`
	`   mic_array_frame_fft_preprocessed *alias f_fft_preprocessed,`
	`   mic_array_decimator_config_t dc[])`

Parameters	`c_from_decimators` The channels used to transfer pointers between the application and the mic_array_decimate_to_pcm_4ch() tasks. `decimator_count` The count of mic_array_decimate_to_pcm_4ch() tasks. `buffer` The buffer index (Used internally) `f_fft_preprocessed` An array of complex frames. `dc` The array cointaining the decimator configuration for each decimator.
Returns	A pointer to the frame now owned by the application. That is, the most recently written samples.

16.4 Frame types

Type	**mic_array_complex_t**
Description	Complex number in Cartesian coordinates.

Fields	`int32_t re`
	The real component.
	
	`int32_t im`
	The imaginary component.

Type	**mic_array_frame_time_domain**
Description	A frame of raw audio.

Fields	`long long alignment`
	Used to force double work alignment.
	
	`int32_t data`
	Raw audio data.
	
	`mic_array_metadata_t metadata`
	Frame metadata (Used internally).

Type	**mic_array_frame_frequency_domain**
Description	A frame of frequency domain audio in Cartesian coordinates.

Fields	`long long alignment`
	Used to force double work alignment.
	
	`mic_array_complex_t data`
	Complex audio data (Cartesian).
	
	`mic_array_metadata_t metadata`
	Frame metadata (Used internally).

Type	**mic_array_frame_fft_preprocessed**
Description	A frame of audio preprocessed for direct insertion into an FFT.

Continued on next page

Fields	`long long alignment` Used to force double work alignment. `mic_array_complex_t data` Complex audio data (Cartesian). `mic_array_metadata_t metadata` Frame metadata (Used internally).

16.5 High resolution delay task

Function	**mic_array_hires_delay**
Description	High resolution delay component. This task handles the application of individual delays for up to 16 channels. Each unit of delay represents one sample at the input sample rate, i.e. the rate at which the circular buffer is being updated. The maximum delay is given by the size of the circular buffer.

Type	`void`
	`mic_array_hires_delay(streaming chanend c_from_pdm_frontend[],`
	`   streaming chanend c_to_decimator[],`
	`   unsigned n,`
	`   streaming chanend c_cmd)`

Parameters	`c_from_pdm_frontend` The channels connecting to the output of the PDM interface `c_to_decimator` The channels connecting to the input of the 4 channel decimators. `n` The size of the two channel arrays, they must be the same. `c_cmd` The channel connecting the application to this task used for setting the delays.

Function	**mic_array_hires_delay_set_taps**
Description	Application side interface to high resolution delay. This function is used by the client of the high resolution delay to set the delays.

Type	`void`
	`mic_array_hires_delay_set_taps(streaming chanend c_cmd,`
	`                                unsigned delays[],`
	`                                unsigned num_channels)`

Continued on next page

Parameters		
	c_cmd	The channel connecting the application to this task used for setting the delays.
	delays	An array of the delays to be set. These must all be less than MIC_ARRAY_HIRES_MAX_DELAY.
	num_channels	The number of microphones. This must be the same as the delays array.

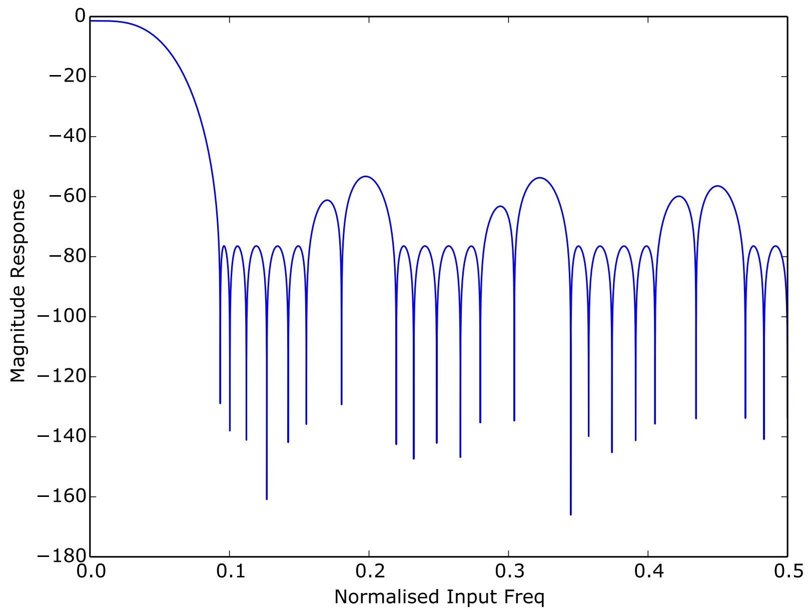


Figure 11: First stage FIR magnitude response.

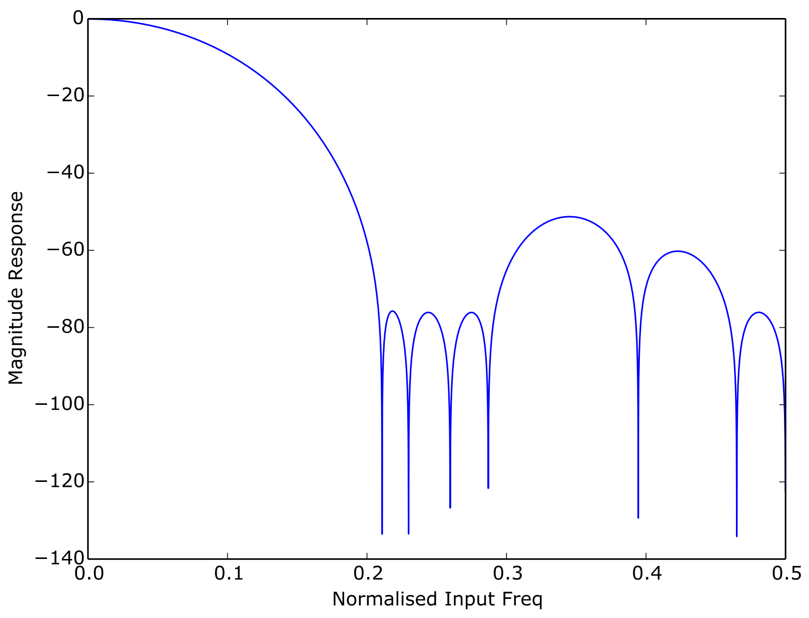


Figure 12: Second stage FIR magnitude response.

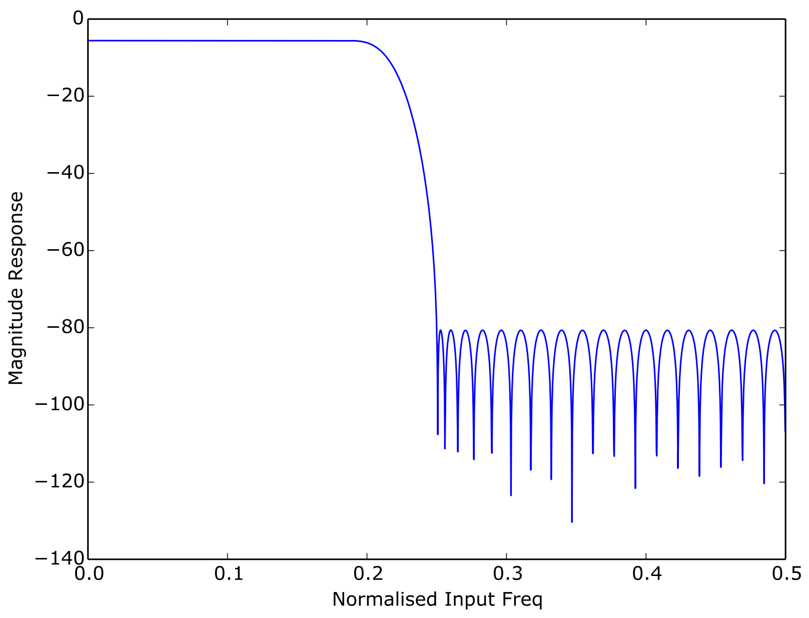


Figure 13: Third stage FIR magnitude response for a divide of 2.

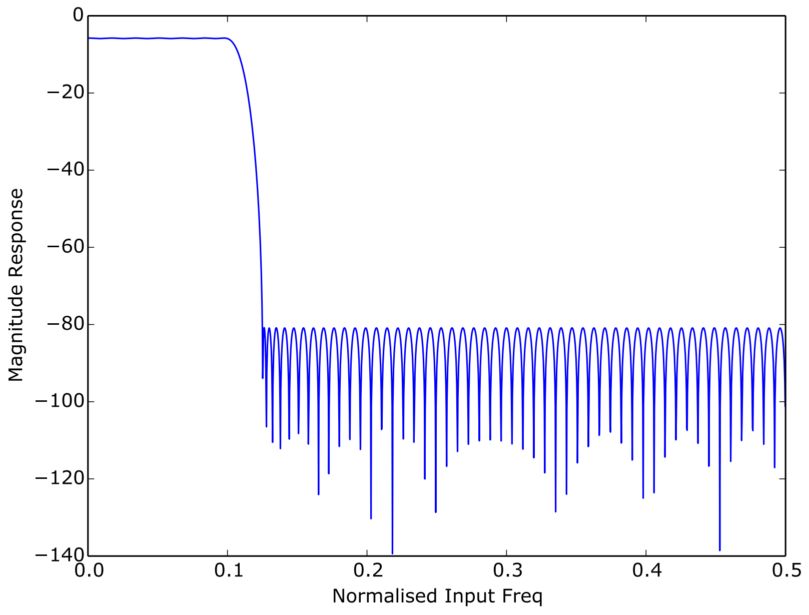


Figure 14: Third stage FIR magnitude response for a divide of 4.

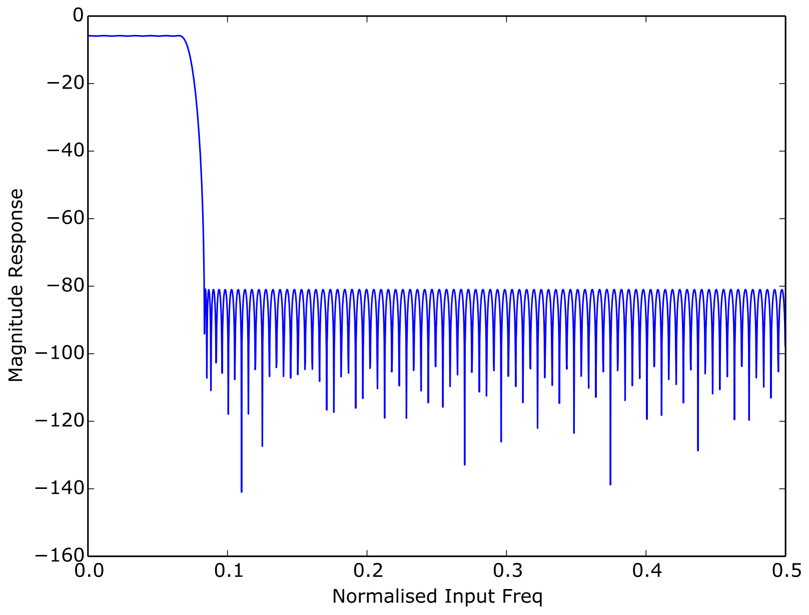


Figure 15: Third stage FIR magnitude response for a divide of 6.

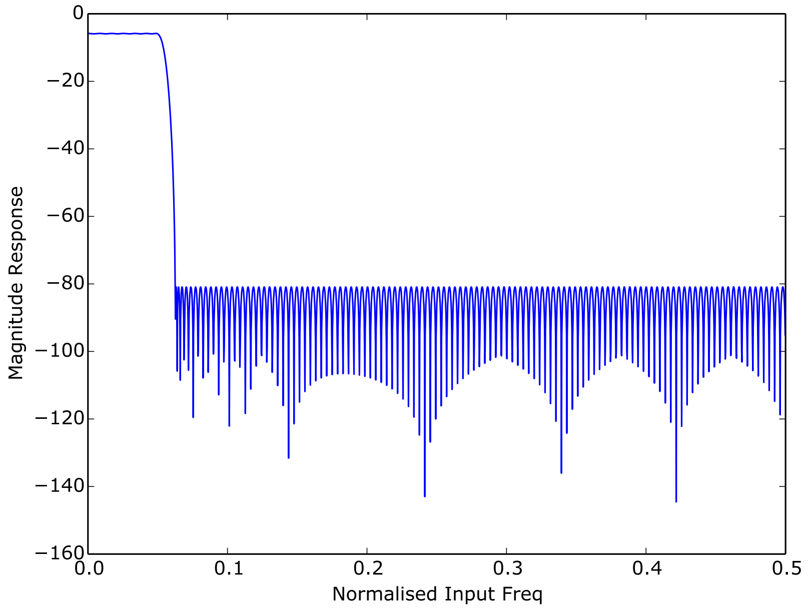


Figure 16: Third stage FIR magnitude response for a divide of 8.

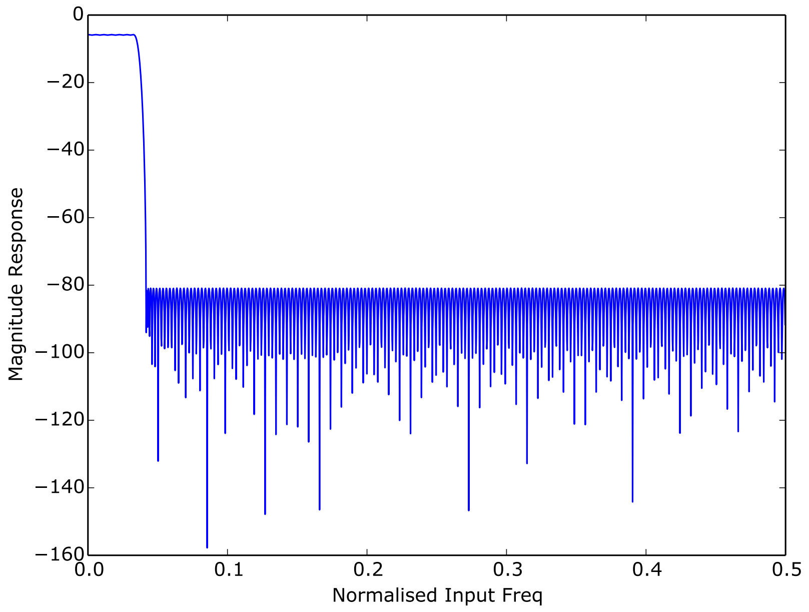


Figure 17: Third stage FIR magnitude response for a divide of 12.

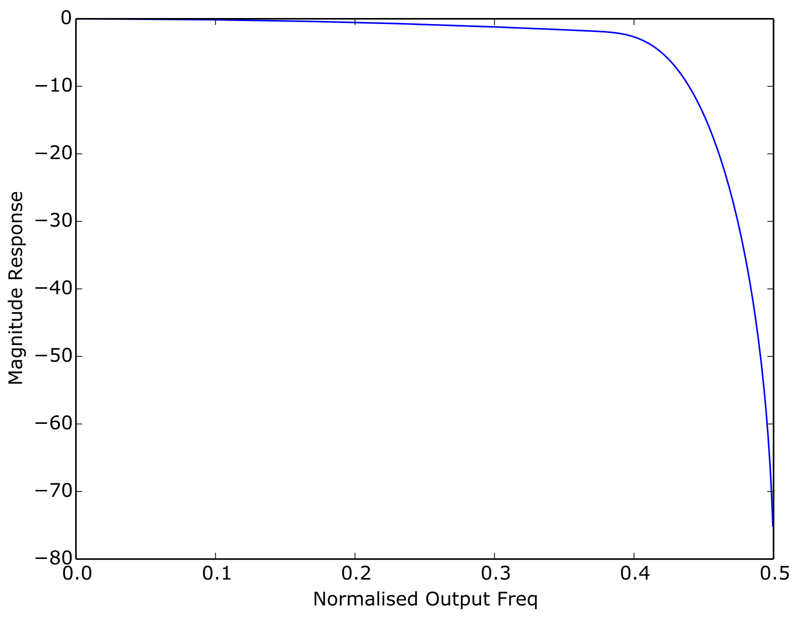


Figure 18: Final frequency response for a divide of 2.

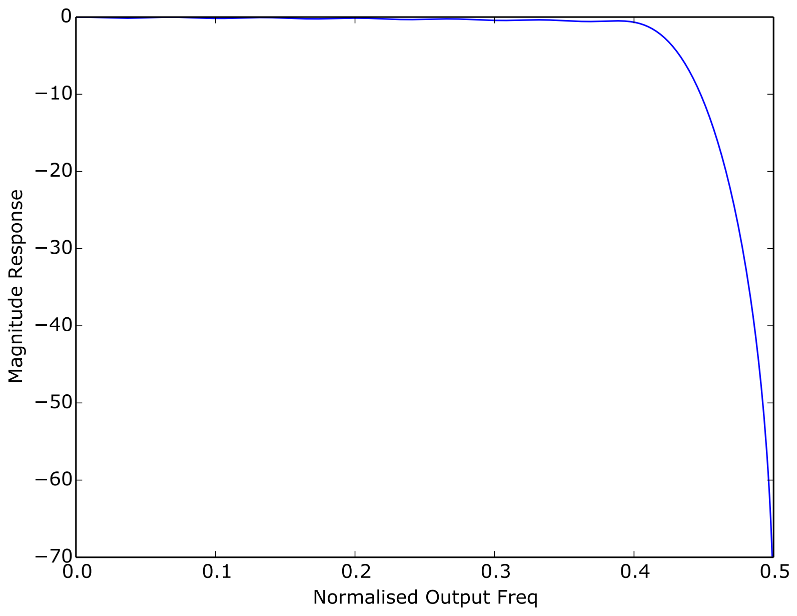


Figure 19: Final frequency response for a divide of 4.

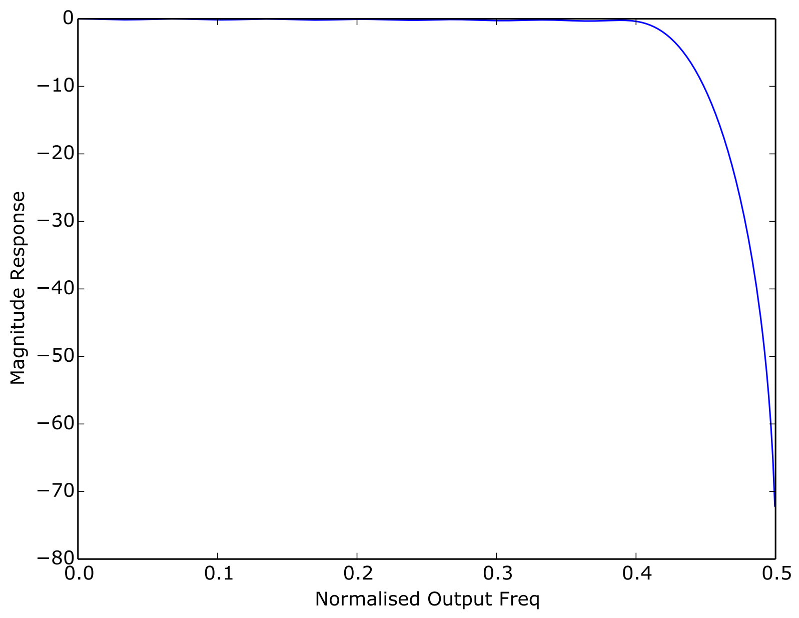


Figure 20: Final frequency response for a divide of 6.

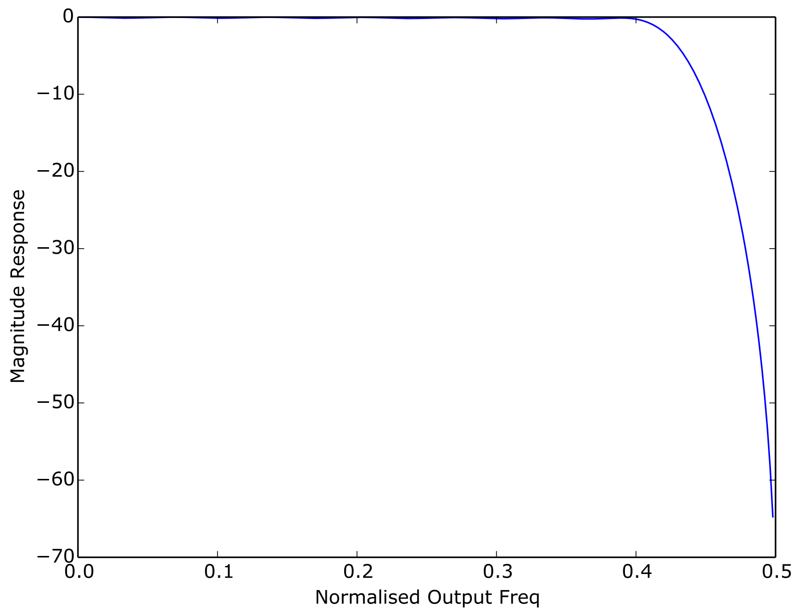


Figure 21: Final frequency response for a divide of 8.

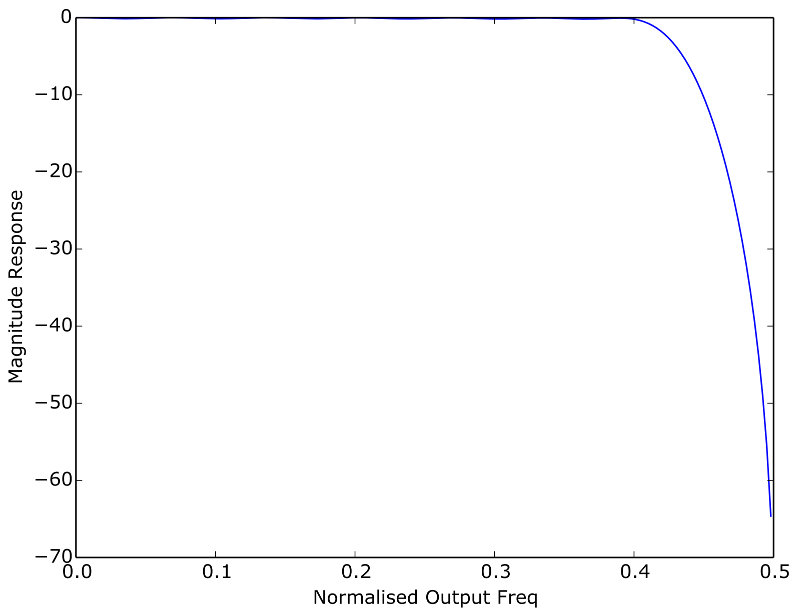


Figure 22: Final frequency response for a divide of 12.

APPENDIX A - Known Issues

- decimator_config channel count is tested for 4 channels per decimator, fewer than 4 is untested.

APPENDIX B - lib_mic_array change log

B.1 2.0.0

- Renamed all functions to match library structure
- Decimator interface functions now take the array of mic_array_decimator_config structure rather than mic_array_decimator_config_common
- All defines renames to match library naming policy
- DC offset simplified
- Added optional MIC_ARRAY_NUM_MICS define to save memory when using less than 16 microphones

B.2 1.0.1

- Added dynamic DC offset removal at startup to eliminate slow convergance
- Mute first 32 samples to allow DC offset to adapt before outputting signal
- Fixed XTA scripte to ensure timing is being met
- Now use a 64-bit accumulator for DC offset removal
- Consolidated generators into a single python generator
- Produced output frequency response graphs
- Added 16 bit output mode

B.3 1.0.0

- Major refactor
- FRAME_SIZE_LOG2 renamed MAX_FRAME_SIZE_LOG2
- Decimator interface now takes arrays of streaming channels
- Decimators now take channel count as a parameter
- Added filter designer script
- Documentation updates
- First stage now uses a FIR decimator
- Changed decimation flow
- Removed high res delay module
- Added generator for FIR coefficients
- Added ability to reduce number of channels active in a decimator
- Increased number of FIR taps
- Increased output dynamic range

B.4 0.0.2

- Documentation fixes
- Fixed frame number fix
- Added frame metadata

B.5 0.0.1

- Initial Release
- Changes to dependencies:
  - lib_logging: Added dependency 2.0.0
  - lib_xassert: Added dependency 2.0.0