Application Note: AN00219

Low Resolution Delay and Sum

Required tools and libraries

The code in this application note is known to work on version ??? of the xTIMEcomposer tools suite, it may work on other versions.

The application depends on the following libraries:

- lib_i2s
- lib_mic_array_board_support
- lib_mic_array
- lib_i2c

Required hardware

The example code provided with the application has been implemented and tested on the Microphone Array Ref Design v1.

Prerequisites

- This document assumes familiarity with the XMOS xCORE architecture, the XMOS tool chain and the xC language. Documentation related to these aspects which are not specific to this application note are linked to in the references appendix.
- The `lib_mic_array` user guide should be thoroughly read and understood.
- For a description of XMOS related terms found in this document please see the XMOS Glossary[1].

[1] http://www.xmos.com/published/glossary

1 Overview

1.1 Introduction

This demo application shows a simple Delay and Sum (DAS) bemformer. It shows the setup of the I^{2}S for audio output via the DAC and very simple processing of multi-channel audio frames to produce a single channel output based on a simple single steering direction.

1.2 Block diagram

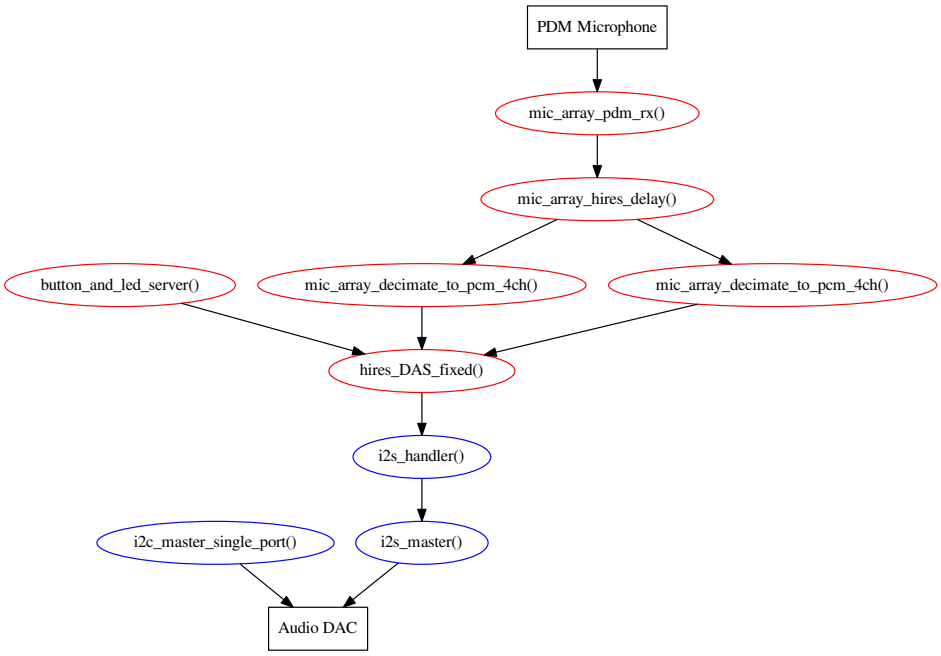

Figure 1: Application block diagram

2 How to use lib_mic_array

2.1 The Makefile

To start using the microphone array library, you need to add `lib_mic_array` to you Makefile:

```
USED_MODULES = .. lib_mic_array ...
```

This demo also uses the logging library (`lib_logging`) for the debug_printf function. This is a faster, but more limited version of the C-Standard Library `printf` function. So the Makefile also includes:

```
USED_MODULES = .. lib_logging ..
```

The logging library is configurable at compile-time allowing calls to debug_printf() to be easily enabled or disabled. For the prints to be enabled it is necessary to add the following to the compile flags:

```
XCC_FLAGS = .. -DDEBUG_PRINT_ENABLE=1 ..
```

2.2 Includes

This application requires the system headers that defines XMOS xCORE specific defines for declaring and initialising hardware:

```
#include <platform.h>
#include <xs1.h>
#include <string.h>
#include <xclib.h>
```

The microphone array library functions are defined in `lib_mic_array.h`. This header must be included in your code to use the library. The support functions for the board are defined in `mic_array_board_support.h` and the logging functions are provided by debug_print.h.

```
#include "mic_array.h"
#include "mic_array_board_support.h"
#include "debug_print.h"
```

Also required is support for I^2S and I^2C through the headers:

```
#include "i2c.h"
#include "i2s.h"
```

2.3 Allocating hardware resources

A PDM microphone requires a clock and a data pin. For eight PDM mocrophones a single clock can be shared between all microphones and the data can be sampled on a single 8 bit port. On an xCORE the pins are controlled by ports. The application therefore declares one 1-bit port and one 8-bit port:

```
on tile[0]: in port p_pdm_clk               = XS1_PORT_1E;
on tile[0]: in buffered port:32 p_pdm_mics  = XS1_PORT_8B;
```

To generate the PDM clock a 24.576MHz master clock is divided by 8 using a clock block. These two hardware resources are declared with:

```
on tile[0]: in port p_mclk                  = XS1_PORT_1F;
on tile[0]: clock pdmclk                    = XS1_CLKBLK_1;
```

and are configured with:

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

The result begin a 3.072MHz PDM clock is used for clocking the microphone data into the xCORE. Additionally, the leds and buttons are declared by

```
on tile[0]:p_leds leds = DEFAULT_INIT;
on tile[0]:in port p_buttons =  XS1_PORT_4A;
```

And the I^2S is declared with:

```
out buffered port:32 p_i2s_dout[1]  = on tile[1]: {XS1_PORT_1P};
in port p_mclk_in1                  = on tile[1]: XS1_PORT_1O;
out buffered port:32 p_bclk         = on tile[1]: XS1_PORT_1M;
out buffered port:32 p_lrclk        = on tile[1]: XS1_PORT_1N;
port p_i2c                          = on tile[1]: XS1_PORT_4E; // Bit 0: SCLK, Bit 1: SDA
port p_rst_shared                   = on tile[1]: XS1_PORT_4F; // Bit 0: DAC_RST_N, Bit 1: ETH_RST_N
clock mclk                          = on tile[1]: XS1_CLKBLK_3;
clock bclk                          = on tile[1]: XS1_CLKBLK_4;
```

3 Demo Hardware Setup

To run the demo, connect a USB cable to power the Microphone Array Ref Design v1 and plug the xTAG to the board and connect the xTAG USB cable to your development machine. You will also need to connect headphones to the audio jack.

TODO

4 Launching the demo application

Once the demo example has been built either from the command line using xmake or via the build mechanism of xTIMEcomposer studio it can be executed on the Microphone Array Ref Design v1.

Once built there will be a bin/ directory within the project which contains the binary for the xCORE device. The xCORE binary has a XMOS standard .xe extension.

4.1 Launching from the command line

From the command line you use the xrun tool to download and run the code on the xCORE device:

```
xrun --xscope bin/app_lores_DAS_fixed.xe
```

Once this command has executed the application will be running on the Microphone Array Ref Design v1.

4.2 Launching from xTIMEcomposer Studio

From xTIMEcomposer Studio use the run mechanism to download code to xCORE device. Select the xCORE binary from the bin/ directory, right click and go to Run Configurations. Double click on xCORE application to create a new run configuration, enable the xSCOPE I/O mode in the dialog box and then select Run.

Once this command has executed the application will be running on the Microphone Array Ref Design v1.

4.3 Running the application

Once the application is started using either of the above methods there will be the output of the microphones through the headphones.

Buttons A and D rotate the direction of the beam which is indicated by the LEDs. Buttons B and C decrease and increase the gain on the output signal respectively.

5 Task setup

The PDM microphones interface task and the decimators have to be connected together and to the application (`lores_DAS_fixed()`). There needs to be one `mic_array_decimate_to_pcm_4ch()` task per four channels that need processing. The PDM interface task, `mic_array_pdm_rx()` can process eight channels so only one is needed for this application. The PDM interface needs to be connected to the decimators via two streaming channels. Finally, the decimators have to be connected to the application.

Note that the decimators have to be on the same tile as the application due to shared frame memory.

6 Frame memory

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

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

bytes. The data must also be double word aligned. For example:

```
int data[8][THIRD_STAGE_COEFS_PER_STAGE*DECIMATION_FACTOR];
```

Note that on the xCORE-200 all global arrays are guaranteed to be double-word aligned.

7 Configuration

Configuration of the microphone array for the example is achieved through:

```
mic_array_decimator_config_common dcc = {0, 1, 0, 0, DECIMATION_FACTOR,
       g_third_stage_div_2_fir, 0, FIR_COMPENSATOR_DIV_2,
       DECIMATOR_NO_FRAME_OVERLAP, FRAME_BUFFER_COUNT};
mic_array_decimator_config dc[2] = {
  {&dcc, data[0], {INT_MAX, INT_MAX, INT_MAX, INT_MAX}, 4},
  {&dcc, data[4], {INT_MAX, INT_MAX, INT_MAX, INT_MAX}, 4}
};

mic_array_decimator_configure(c_ds_output, DECIMATOR_COUNT, dc);
```

All configuration options are described in the Microphone array library guide. Once configured then the decimators require initialization via:

```
mic_array_init_time_domain_frame(c_ds_output, DECIMATOR_COUNT, buffer, audio, dc);
```

The the decimators will start presenting samples in the form of frames that can be accessed with:

```
mic_array_frame_time_domain * current =
                  mic_array_get_next_time_domain_frame(c_ds_output, DECIMATOR_COUNT, buffer, audio, dc);
```

The return value of `mic_array_get_next_time_domain_frame()` is a pointer to the frame that the application is allowed to access. The `current` frame contains the frame data in the `data` member. `data` is a 2D array with the first index denoting the channel number and the second index denoting the frame index. The frame index used 0 for the oldest samples and increasing indicies for newer samples.

8 Delay taps

The delays on the microphones are calculated in a spread sheet included at the root folder of the application, `mic_array_das_beamformer_calcs.xls`. The beam is focused to a point of one meter away at an angle of thirty degrees from the plane of the microphone array in the direction indicated by the LEDs.

9 References

XMOS Tools User Guide

http://www.xmos.com/published/xtimecomposer-user-guide

XMOS xCORE Programming Guide

http://www.xmos.com/published/xmos-programming-guide

XMOS Microphone Array Library

http://www.xmos.com/support/libraries/lib_mic_array

XMOS I^2C Library

http://www.xmos.com/support/libraries/lib_i2c

XMOS I^2S Library

http://www.xmos.com/support/libraries/lib_i2s

10 Full source code listing

10.1 Source code for app_lores_DAS_fixed.xc

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved
#include <platform.h>
#include <xs1.h>
#include <string.h>
#include <xclib.h>

#include "mic_array.h"
#include "mic_array_board_support.h"
#include "debug_print.h"

#include "i2c.h"
#include "i2s.h"

//If the decimation factor is changed the the coefs array of decimator_config must also be changed.
#define DECIMATION_FACTOR   2   //Corresponds to a 48kHz output sample rate
#define DECIMATOR_COUNT     2   //8 channels requires 2 decimators
#define FRAME_BUFFER_COUNT  2   //The minimum of 2 will suffice for this example

on tile[0]:p_leds leds = DEFAULT_INIT;
on tile[0]:in port p_buttons =  XS1_PORT_4A;

on tile[0]: in port p_pdm_clk               = XS1_PORT_1E;
on tile[0]: in buffered port:32 p_pdm_mics  = XS1_PORT_8B;
on tile[0]: in port p_mclk                  = XS1_PORT_1F;
on tile[0]: clock pdmclk                    = XS1_CLKBLK_1;

out buffered port:32 p_i2s_dout[1]  = on tile[1]: {XS1_PORT_1P};
in port p_mclk_in1                  = on tile[1]: XS1_PORT_1O;
out buffered port:32 p_bclk         = on tile[1]: XS1_PORT_1M;
out buffered port:32 p_lrclk        = on tile[1]: XS1_PORT_1N;
port p_i2c                          = on tile[1]: XS1_PORT_4E; // Bit 0: SCLK, Bit 1: SDA
port p_rst_shared                   = on tile[1]: XS1_PORT_4F; // Bit 0: DAC_RST_N, Bit 1: ETH_RST_N
clock mclk                          = on tile[1]: XS1_CLKBLK_3;
clock bclk                          = on tile[1]: XS1_CLKBLK_4;

// Based on the spreadsheet mic_array_das_beamformer_calcs.xls,
// which can be found in the root directory of this app
static const one_meter_thirty_degrees[6] = {0, 3, 8, 11, 8, 3};

static void set_dir(client interface led_button_if lb,
                    unsigned dir, unsigned delay[]) {

    for(unsigned i=0;i<13;i++)
        lb.set_led_brightness(i, 0);
    delay[0] = 5;
    for(unsigned i=0;i<6;i++)
        delay[i+1] = one_meter_thirty_degrees[(i - dir + 3 +6)%6];

    switch(dir){
    case 0:{
        lb.set_led_brightness(0, 255);
        lb.set_led_brightness(1, 255);
        break;
    }
    case 1:{
        lb.set_led_brightness(2, 255);
        lb.set_led_brightness(3, 255);
        break;
    }
    case 2:{
        lb.set_led_brightness(4, 255);
        lb.set_led_brightness(5, 255);
        break;
    }
    case 3:{
        lb.set_led_brightness(6, 255);
        lb.set_led_brightness(7, 255);
        break;
    }
```

```
    case 4:{
        lb.set_led_brightness(8, 255);
        lb.set_led_brightness(9, 255);
        break;
    }
    case 5:{
        lb.set_led_brightness(10, 255);
        lb.set_led_brightness(11, 255);
        break;
    }
    }
}

int data[8][THIRD_STAGE_COEFS_PER_STAGE*DECIMATION_FACTOR];

void lores_DAS_fixed(streaming chanend c_ds_output[DECIMATOR_COUNT],
        client interface led_button_if lb, chanend c_audio) {

    unsafe{
        unsigned buffer;
        memset(data, 0, 8*THIRD_STAGE_COEFS_PER_STAGE*DECIMATION_FACTOR*sizeof(int));

        mic_array_frame_time_domain audio[FRAME_BUFFER_COUNT];

        #define MAX_DELAY 16
        unsigned gain = 8;
        unsigned delay[7];
        int delay_buffer[MAX_DELAY][7];
        memset(delay_buffer, 0, sizeof(int)*MAX_DELAY*7);
        unsigned delay_head = 0;
        unsigned dir = 0;
        set_dir(lb, dir, delay);

        mic_array_decimator_config_common dcc = {0, 1, 0, 0, DECIMATION_FACTOR,
               g_third_stage_div_2_fir, 0, FIR_COMPENSATOR_DIV_2,
               DECIMATOR_NO_FRAME_OVERLAP, FRAME_BUFFER_COUNT};
        mic_array_decimator_config dc[2] = {
          {&dcc, data[0], {INT_MAX, INT_MAX, INT_MAX, INT_MAX}, 4},
          {&dcc, data[4], {INT_MAX, INT_MAX, INT_MAX, INT_MAX}, 4}
        };

        mic_array_decimator_configure(c_ds_output, DECIMATOR_COUNT, dc);

        mic_array_init_time_domain_frame(c_ds_output, DECIMATOR_COUNT, buffer, audio, dc);

        while(1){

            mic_array_frame_time_domain *  current =
                                mic_array_get_next_time_domain_frame(c_ds_output, DECIMATOR_COUNT, buffer,
                                  ↪ audio, dc);

            // Copy the current sample to the delay buffer
            for(unsigned i=0;i<7;i++)
                delay_buffer[delay_head][i] = current->data[i][0];

            // light the LED for the current direction
            int t;
            select {
                case lb.button_event():{
                    unsigned button;
                    e_button_state pressed;
                    lb.get_button_event(button, pressed);
                    if(pressed == BUTTON_PRESSED){
                        switch(button){
                        case 0:
                            dir--;
                            if(dir == -1)
                                dir = 5;
                            set_dir(lb, dir, delay);
                            debug_printf("dir %d\n", dir+1);
                            for(unsigned i=0;i<7;i++)
                                debug_printf("delay[%d] = %d\n", i, delay[i]);
                            debug_printf("\n");
                            break;
```

```
                        case 1:
                            gain = ((gain<<3) - gain)>>3;
                            debug_printf("gain: %d\n", gain);
                            break;

                        case 2:
                            if (gain == 0)
                                gain = 5;
                            gain = ((gain<<3) + gain)>>3;
                            debug_printf("gain: %d\n", gain);
                            break;

                        case 3:
                            dir++;
                            if(dir == 6)
                                dir = 0;
                            set_dir(lb, dir, delay);
                            debug_printf("dir %d\n", dir+1);
                            for(unsigned i=0;i<7;i++)
                                debug_printf("delay[%d] = %d\n", i, delay[i]);
                            debug_printf("\n");
                            break;
                        }
                    }
                    break;
                }
                default:break;
            }
            int output = 0;
            for(unsigned i=0;i<7;i++)
                output += (delay_buffer[(delay_head - delay[i])%MAX_DELAY][i]>>3);
            output *= gain;

            // Update the center LED with a volume indicator
            unsigned value = output >> 20;
            unsigned magnitude = (value * value) >> 8;
            lb.set_led_brightness(12, magnitude);

            c_audio <: output;
            c_audio <: output;
            delay_head++;
            delay_head%=MAX_DELAY;
        }
    }
}
void init_cs2100(client i2c_master_if i2c){

    #define CS2100_DEVICE_CONFIG_1      0x03
    #define CS2100_GLOBAL_CONFIG        0x05
    #define CS2100_FUNC_CONFIG_1        0x16
    #define CS2100_FUNC_CONFIG_2        0x17

    i2c_regop_res_t res;
    res = i2c.write_reg(0x9c>>1, CS2100_DEVICE_CONFIG_1, 0);
    res = i2c.write_reg(0x9c>>1, CS2100_GLOBAL_CONFIG, 0);
    res = i2c.write_reg(0x9c>>1, CS2100_FUNC_CONFIG_1, 0);
    res = i2c.write_reg(0x9c>>1, CS2100_FUNC_CONFIG_2, 0);
}

#define MASTER_TO_PDM_CLOCK_DIVIDER 4
#define MASTER_CLOCK_FREQUENCY 24576000
#define PDM_CLOCK_FREQUENCY (MASTER_CLOCK_FREQUENCY/(2*MASTER_TO_PDM_CLOCK_DIVIDER))
#define OUTPUT_SAMPLE_RATE (PDM_CLOCK_FREQUENCY/(32*DECIMATION_FACTOR))

[[distributable]]
void i2s_handler(server i2s_callback_if i2s,
                 client i2c_master_if i2c, chanend c_audio) {
  p_rst_shared <: 0xF;

  init_cs2100(i2c);
  i2c_regop_res_t res;
  int i = 0x4A;
  uint8_t data = i2c.read_reg(i, 1, res);

  data = i2c.read_reg(i, 0x02, res);
```

```
  data |= 1;
  res = i2c.write_reg(i, 0x02, data); // Power down

  // Setting MCLKDIV2 high if using 24.576MHz.
  data = i2c.read_reg(i, 0x03, res);
  data |= 1;
  res = i2c.write_reg(i, 0x03, data);

  data = 0b01110000;
  res = i2c.write_reg(i, 0x10, data);

  data = i2c.read_reg(i, 0x02, res);
  data &= ~1;
  res = i2c.write_reg(i, 0x02, data); // Power up

#define CS2100_I2C_DEVICE_ADDR      (0x9c>>1)
  res = i2c.write_reg(CS2100_I2C_DEVICE_ADDR, 0x3, 0); // Reset the PLL to use the aux out


  while (1) {
    select {
    case i2s.init(i2s_config_t &?i2s_config, tdm_config_t &?tdm_config):
      i2s_config.mode = I2S_MODE_LEFT_JUSTIFIED;
      i2s_config.mclk_bclk_ratio = (MASTER_CLOCK_FREQUENCY/OUTPUT_SAMPLE_RATE)/64;
      break;

    case i2s.restart_check() -> i2s_restart_t restart:
      restart = I2S_NO_RESTART;
      break;

    case i2s.receive(size_t index, int32_t sample):
      break;

    case i2s.send(size_t index) -> int32_t sample:
      c_audio:> sample;
      break;
    }
  }
}

int main() {

    i2s_callback_if i_i2s;
    i2c_master_if i_i2c[1];
    chan c_audio;
    par{

        on tile[1]: {
          configure_clock_src(mclk, p_mclk_in1);
          start_clock(mclk);
          i2s_master(i_i2s, p_i2s_dout, 1, null, 0, p_bclk, p_lrclk, bclk, mclk);
        }

        on tile[1]:  [[distribute]]i2c_master_single_port(i_i2c, 1, p_i2c, 100, 0, 1, 0);
        on tile[1]:  [[distribute]]i2s_handler(i_i2s, i_i2c[0], c_audio);

        on tile[0]: {
            configure_clock_src_divide(pdmclk, p_mclk, MASTER_TO_PDM_CLOCK_DIVIDER);
            configure_port_clock_output(p_pdm_clk, pdmclk);
            configure_in_port(p_pdm_mics, pdmclk);
            start_clock(pdmclk);

            streaming chan c_4x_pdm_mic[DECIMATOR_COUNT];
            streaming chan c_ds_output[DECIMATOR_COUNT];

            interface led_button_if lb[1];

            par {
                button_and_led_server(lb, 1, leds, p_buttons);
                mic_array_pdm_rx(p_pdm_mics, c_4x_pdm_mic[0], c_4x_pdm_mic[1]);
                mic_array_decimate_to_pcm_4ch(c_4x_pdm_mic[0], c_ds_output[0]);
                mic_array_decimate_to_pcm_4ch(c_4x_pdm_mic[1], c_ds_output[1]);
                lores_DAS_fixed(c_ds_output, lb[0], c_audio);
            }
        }
```

```
    }
    return 0;
}
```