LAB 2: FIR ON DSP BOARD DSK6713

Jacobs University Bremen

CO27-300231 DSP & Communications Lab

Spring Semester 2020

Prof. Fangning Hu

Kelan Garcia

May 4, 2020

Mailbox Number: XC-316

Contents

0.1 Introduction

Objective

The objective of this lab is to learn how to program a real-time dsp FIR block in C. Following all dsp coding and oriented programming practices.

Background

- Structured Programming of DSP Blocks in C

  An individual DSP block can be coded by writing the following functions:

  1. blockname_init(): A function that initializes the block and returns a new state structure for the block.
  2. blockname(): A function that processes buffers of input samples to generate buffers of output samples.
  3. blockname_modify(): An optional function that allows the operation of the block to be modified at runtime.

- File Organization For each block, you write it in two files:

  1. blockname.h: A "header" file that contains global definitions for the block. This file can then be "included" in other C files that need to use the block.
  2. blockname.c: The actual code that implements the block.

0.2 Tasks

Task 1: Read the chapter FIR Filter Design (Window Method) at this webpage[2] and understand how to design an FIR filter by window method. Describe the procedure to design an FIR filter in the lab report.

In order to design an FIR filter first we design the filter in the frequency domain depending on the desire cuttoff frequencies as shown in Figure 1.

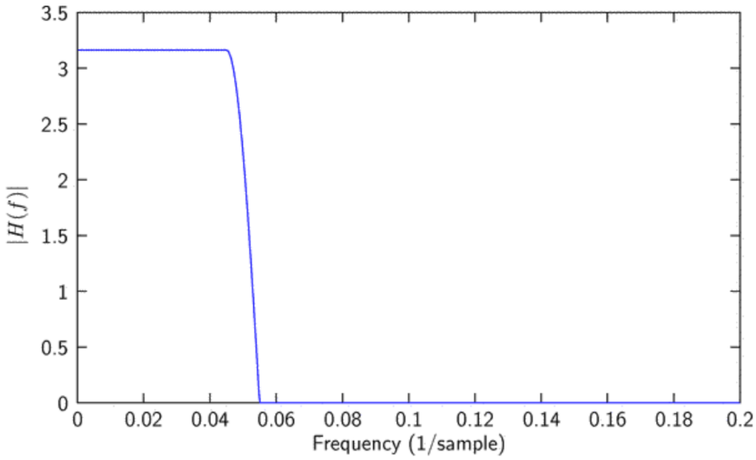

Figure 1: Desire FIR FIlter $H(f)$ with cuttoff at 0.05[4]

Next, the signal $H(f)$ in Figure 1 is converted in time domain with IDFT (i.e., $h_f[n]$) and multiplied by a rectangular window function $w_r[n]$ of lenght M resulting in a sinc function in the time domain (Figure 2).

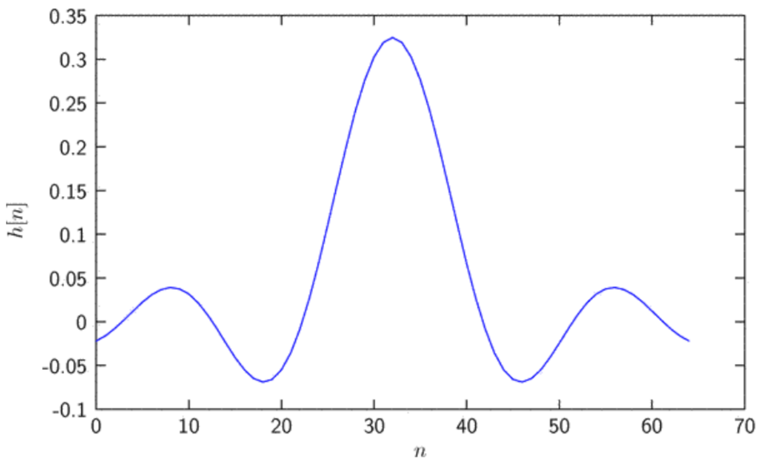

Figure 2: Response of a rectangular window (i.e.,$h[n] = h_f[n] \times w_r[n]$)

The signal in Figure 2 can be the final FIR filter, but rectangular windows have high side lobes. In order to avoid this, the Hamming window (i.e.,$w_h\left[n\right]$) is used insted of a rectangular window.

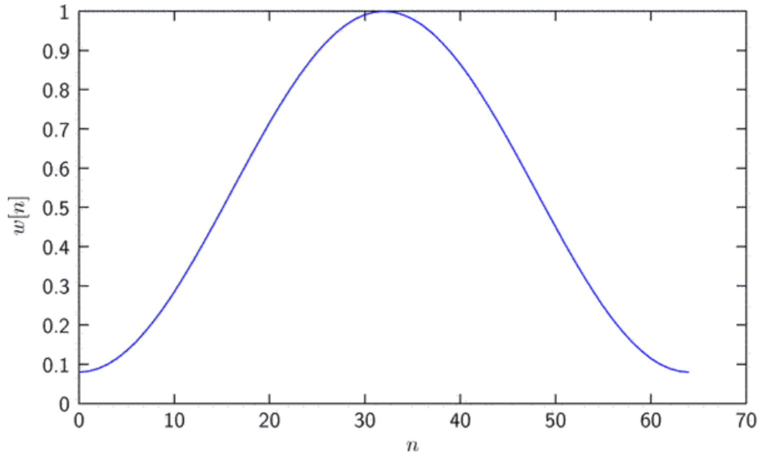

Figure 3: Hamming window (i.e.,$w_h\left[n\right]$)

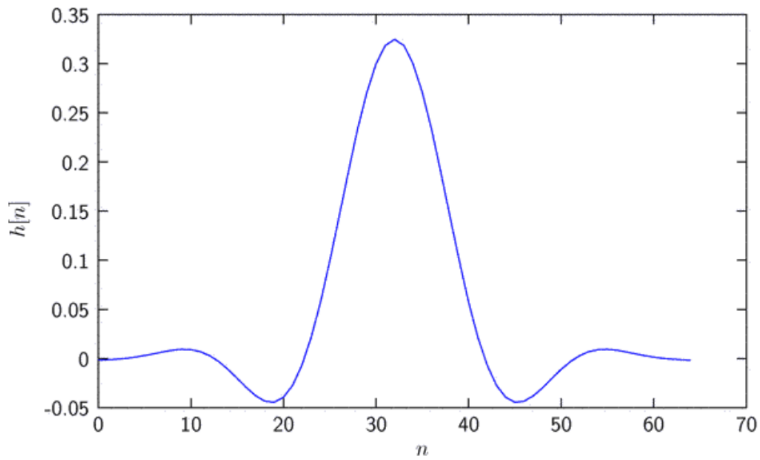

Figure 4: Response of a Hamming window (i.e.,$h\left[n\right] = h_f\left[n\right] \times w_h\left[n\right]$)

Task 2: Download dsp_lab.zip extract it and read the file "win_method.m", explain what "Hp" denotes in "win_method.m" and which lines of the code try to calculate the time domain filter's coefficients h_k by IDFT.

Note that the formula for IDFT is:

$$h_k = \frac{1}{N}\sum_{n=0}^{N-1} H_n \cdot e^{\frac{j2\pi kn}{N}}, n, k \in Z$$

```
function [hw, f, Ha, Hi, win] = win_method(H_func, p, f_max, fs, M, wtype);

% Creates a filter using the window method.
%
% Inputs:
% H_func    User-supplied function that gives H(f, p)
% p   Options to pass to the user function
% f_max   Maximum integration frequency
% fs    Sample rate
% M   Number of filter taps (-1)
% wtype   Window to use: 0=rect, 1=hamming
%
% Outputs:
% h   Time domain filter taps
% f   Frequency samples used
% Ha    Actual response of filter
% Hi    Ideal response of filter
% win   Window applied

% Response is real (H is symmetric)
op_real = 1;
op_lin_phase = 1;

% Get the frequency samples
K = 1000;
f = ([0:K-1]+0.5)/K*f_max;
w = 2*pi*f;

% Time samples
m = [0:M].';
om = ones(size(m));

```

```
% Evaluate the user function at the sample points
eval(sprintf('Hp = %s(f, p);', H_func));

% Get negative frequencies if necessary
if ~op_real,
  sprintf('Hm = %s(f, p);', H_func);
else
  Hm = conj(Hp);
end

% Get discrete frequencies
wd = w/fs;
dw = 2*pi*f_max/K/fs;

% Put in the delay (lin phase) to make causal.
if (op_lin_phase),
  lp = exp(-j*wd*M/2);
else
  lp = ones(size(Hp));
end

% Do integration
hp = 1/(2*pi)*sum(((om*(lp.*Hp)).*exp(j*m*wd)).')*dw;
hm = 1/(2*pi)*sum(((om*(conj(lp).*Hm)).*exp(-j*m*wd)).')*dw;

h = (hp + hm).';

% Do window
if wtype == 0,
  win = ones(1,M+1).';
elseif (wtype == 1),
  % Hamming window
  win = 0.54 - 0.46*cos(2*pi*m/M);
else
  error('Invalid window type');
end

hw = h.*win;

% Compute the response back in the frequency domain
Hi = Hp;

```

```
75 Ha = zeros(size(f));
76 for ii=1:length(f),
77   Ha(ii) = sum(hw.*exp(j*2*pi*f(ii)/fs*m));
78 end
```

"Hp" denotes the values of the desire filter. In the 34^{th} codeline "Hp" is created depeding on the variable H_func. This variable decides if is a rectangular or a raise cosine filter.

Next, in codelines 36-41 we get the negative frequencies if necessary depending if we want the function at frequency domain to be full real or imaginary. Then, in codelines 47-52 we implement the linear phase to make casuality and store it in the vaiable l_p.

Finally, in codelines 54-58 we do the integration. This part is when we implement the IDFT formula, we use "Hp" in order to simplify the terms in the formula, but at the end the time domain function h is giving by code line 58:

$$h = (h_p + h_m).'; \ (58)\ \text{codeline}$$

This calculate the time domain filter's coefficients h_k by IDFT where:

$$h_p = \frac{1}{2\pi}\left(\sum l_p \cdot H_p \cdot e^{\frac{j2\pi kn}{T}}\right) dw \ (55)\ \text{codeline}$$

$$h_m = \frac{1}{2\pi}\left(\sum l_p^* \cdot H_m \cdot e^{-\frac{j2\pi kn}{T}}\right) dw \ (56)\ \text{codeline}$$

and,

$$H_m = \text{conj}(H_p); \iff op\_real \neq 0$$

$$l_p^* = \text{conj}(l_p);$$

In codelines 60 - 68 we create the Rectangular or Hamming window depending in the wtype value. In codeline 70, we multiply the filter time domain with the window function and at the end in codelines 72 - 78, we compute the response back in the frequency domain.

Task 3: Read "Getting Filter Coefficients into C".[3] Read the file "make_fir.m" and explain how the filter time domain coefficients to be produced in the file "make_fir.m" and which file these coefficients will be stored after running the file?

The file "Make_file.m" is the following:

```
1 %                              make_fir.m
2 % Generates the coefficients for an RC filter.
3
4 % Generate filter coefficients
5 p.beta = 0.5;
6 p.fs = 0.333; % Make stop freq at 0.25 = 2000Hz
7 p.root = 0; % 0=rc 1=root rc
8 M = 128;
9 [h f H Hi] = win_method('rc_filt', p, 0.5, 1, M, 0);
10
11 % For this lab, make filter have unit gain in passband
12 scale = abs(H(1));
13 h = h/scale;
14 H = H/scale;
15 Hi = Hi/scale;
16
17 % Write the data out to a file that can be used in C code
18 write_real_array('rc1_taps', 'rc1_taps', 'float', h);
19
20 % Write the filter response to a file so it's actual
21 % response can be tested
22 save rc1_taps.mat h Hi H f
```

First, codelines 5-7 creates a struct "p" with the variables that are needed in order to design the filter. Later, a variable "M" is created, which is the size of the filter. Next, in codeline 9, the function "win_method()" is called. This function input parameters are: Name of the function that creates the filter, in this case is 'rc_filt', the p struct, max frequency = 0.5, sample rate = 1, M = Number of filter taps, Variable that decides if the window use is rectangluar or hamming.

The parameters that we received from the function are: h = Actual Time domain filter. H = Actual Frequency domain filter. Hi = Ideal Frequency domain filter. f = frequency samples. The explanation on how the "win_method()" function calculates the time domain filter is explained in Task 2. Then, in codelines 12-15 the output functions are being scaled.

Finally, by using the write_real_array() function, two output files are created. In this case rc1_taps.h, and rc1_taps.c. These files are going to contain the values of the time domain filter function "h". The "rc1_taps.h" file is the header of the c script file, and this will only contain #define codelines, and the "rc1_taps.c" file contains a float array named "float rc1_taps[rc1_taps_len]" with the values of the time domain filter. At the end, in codeline 22, we are going to save the variables h,Hi,H,f in a file name rc1_taps.mat.

Task 4: Copy the file "make_fir.m" into the same directory where you extract dsp_lab.zip. Run the matlab file "make_fir.m" and provide the resulting filter coefficients in your report or in a seperate file.

"rc1_taps.h"

It defines the length of the array and defines the array as an extern float variable

```
/* File automatically generated by write_real_array.m. */

#define rc1_taps_LEN 129

extern float rc1_taps[rc1_taps_LEN];
```

"rc1_taps.c"

Stores the filter coefficients inside the array rc1_taps.

```
/* File automatically generated by write_real_array.m. */

#define rc1_taps_LEN 129

float rc1_taps[rc1_taps_LEN] = {
-4.29059e-06,
-2.50174e-08,
-5.7139e-06,
-9.30198e-06,
8.36137e-07,
1.06655e-05,
5.82568e-06,
2.76694e-08,
7.69509e-06,
```

```
1.27289e-05,
-1.0331e-06,
-1.47046e-05,
-8.17011e-06,
-3.09507e-08,
-1.07297e-05,
-1.80581e-05,
1.30886e-06,
2.10772e-05,
1.19302e-05,
3.51169e-08,
1.56302e-05,
2.68136e-05,
-1.71189e-06,
-3.17492e-05,
-1.83474e-05,
-4.05843e-08,
-2.40986e-05,
-4.22556e-05,
2.33475e-06,
5.10674e-05,
3.02313e-05,
4.80818e-08,
4.01146e-05,
7.22072e-05,
-3.37287e-06,
-8.99477e-05,
-5.48521e-05,
-5.90149e-08,
-7.44859e-05,
-0.000138671,
5.30083e-06,
0.000181184,
0.000114988,
7.65071e-08,
0.000163752,
0.000319973,
-9.54251e-06,
-0.000454532,
-0.000306926,
-1.09306e-07,
-0.000484863,
```

```
-0.00103095,
2.22688e-05,
0.0017534,
0.00134552,
1.9679e-07,
0.00286003,
0.00767583,
-0.000111408,
-0.0269819,
-0.0442377,
0.000261932,
0.124226,
0.268427,
0.333002,
0.268427,
0.124226,
0.000261932,
-0.0442377,
-0.0269819,
-0.000111408,
0.00767583,
0.00286003,
1.9679e-07,
0.00134552,
0.0017534,
2.22688e-05,
-0.00103095,
-0.000484863,
-1.09306e-07,
-0.000306926,
-0.000454532,
-9.54251e-06,
0.000319973,
0.000163752,
7.65071e-08,
0.000114988,
0.000181184,
5.30083e-06,
-0.000138671,
-7.44859e-05,
-5.90149e-08,
-5.48521e-05,
```

```
-8.99477e-05,
-3.37287e-06,
7.22072e-05,
4.01146e-05,
4.80818e-08,
3.02313e-05,
5.10674e-05,
2.33475e-06,
-4.22556e-05,
-2.40986e-05,
-4.05843e-08,
-1.83474e-05,
-3.17492e-05,
-1.71189e-06,
2.68136e-05,
1.56302e-05,
3.51169e-08,
1.19302e-05,
2.10772e-05,
1.30886e-06,
-1.80581e-05,
-1.07297e-05,
-3.09507e-08,
-8.17011e-06,
-1.47046e-05,
-1.0331e-06,
1.27289e-05,
7.69509e-06,
2.76694e-08,
5.82568e-06,
1.06655e-05,
8.36137e-07,
-9.30198e-06,
-5.7139e-06,
-2.50174e-08,
-4.29059e-06};
```

Task 5: Explain why we don't need to just use $ptr - 1$ in codeline 9?

We don't need to use $ptr - 1$ or $tail - 1$, because we need to implement it in a circular way since is a buffer. Instead $ptr = (ptr + L - 1)\,\text{bitmask\&}\,(L - 1)$ and $ptr = (tail + L - 1)\,\text{bitmask\&}\,(L - 1)$ (where L = buffer size) should be used.

Algorithm 1 Pseudocode of Convolution in a Circular Buffer

procedure CONVOLUTION(x_buffer, h)
  for `n = 0 to M - 1` **do**
    x_buffer($tail$) $\leftarrow n^{th}sample$
    Increment *tail.*
    $ptr \leftarrow tail$ *- 1.*
    $sum \leftarrow 0.0$
    for `i = 0 to N - 1` **do**
      $sum \leftarrow sum + x$_buffer(ptr)* *h(i)*
      $ptr \leftarrow ptr$ *- 1*
    $y(n) \leftarrow sum.$

These formulas are based on updating the tail in a circular buffer using bitwise operators (i.e., "$tail = (tail + 1)\,\&$ *buffer_ Cmask*" where *buffer_ Cmask* $= L - 1$), but since this time we are not updating foward $(+1)$ instead we are updating it backward (-1) we could encounter a problem. $(Tail - 1)$ & $(L - 1)$ could work, but if tail is at position 0, and in here L $= 512$ then the expresion will be (-1) & (511) this -1 could give us errors. So, in order to solve this, we need a way to not reach negatives values (-1). Therefore, if we add the size of the buffer due to the circular form it will end at the same position as started (i.e., 0 & $511 = 0 \rightarrow (0 + 512)$ & $511 = 0$) and since we want to reduce one position then we subtract one (i.e., $0 + 512 - 1$ & $511 = 511$) and that's why they work. Therefore the convolution should be:

Algorithm 2 Pseudocode of Convolution in a Circular Buffer

procedure CONVOLUTION(x_buffer, h)
  $L \leftarrow$ Length of x_buffer
  for `n = 0 to M - 1` **do**
    x_buffer($tail$) $\leftarrow n^{th}sample$
    buffer_Cmask $= L - 1$
    $tail \leftarrow (tail + 1)$ bitmask& *buffer_Cmask*
    $ptr \leftarrow (tail + L - 1)$ bitmask& *buffer_Cmask*
    $sum \leftarrow 0.0$
    for `i = 0 to N - 1` **do**
      $sum \leftarrow sum + x$_buffer(ptr)* *h(i)*
      $ptr \leftarrow (ptr + L - 1)$ bitmask& *buffer_Cmask*
    $y(n) \leftarrow sum.$

Task 6: Read the file „fir.h“[3] and understand the code. Comment the following part:

```
    typedef struct {
        float buffer[FIR_BUFFER_SIZE];
        float len;
        float *h;
        unsigned int t;
    }fir_state_def;
```

The part above is a small part of the "fir.h" file. This is just the part that declares the struct fir_state_def;.

It declares that the struct fir_state_def will contain: a buffer with the size that was define at the top of the "fir.h" file that is "FIR_BUFFER_SIZE" = 512. A variable named "len" which will contain the size of the filter. Then, it contains the variable "*h", this will be a pointer to the coefficients that are store in a c file that was created with the matlab code. Finally, the variable "t" will hold the position of the tail. Later in the fir.h file the functions that are used in fir.c file were declared, which are:

fir_state_def *fir_init(int len, float *h);
void fir(fir_state_def *s, const float x_in[], float y_out[]);

Task 7: Similiar to delay.c[1], write your own fir.c where you need write two functions:

fir_state_def *fir_init(int len, float *h);
void fir(fir_state_def *s, const float x_in[], float y_out[]);

The fir.h is the header of the fir.c block code, in here we define all the needed variables, structures, and function prototypes.

```
/*****************************************************************
                              fir.h
           Header defines for implementing an FIR block.
 *****************************************************************/
#ifndef _fir_h_
    #define _fir_h_

    /*-------------------- Defines -----------------------*/

    /* Size of buffer (samples).  Maximum filter length. */
    #define FIR_BUFFER_SIZE       512

    /* Mask.  Used to implment circular buffer */
```

```c
    #define FIR_BUFFER_CMASK      (FIR_BUFFER_SIZE-1)

    // Which memory segment the data should get stored in
    //#define FIR_SEG_ID 0 // IDRAM - fastest, but smallest
    #define FIR_SEG_ID   1 // SRAM  - a bit slower, but bigger

    /* Allows alignment of buffer on specified boundary. */
    #define FIR_BUFFER_ALIGN            128

    /*------------------- Structures -------------------*/
    typedef struct
    {
        float buffer[FIR_BUFFER_SIZE];
        float len;
        float *h;
        unsigned int t;
    } fir_state_def;

    /*---------------- Function Prototypes ----------------*/

    /* Initializes the fir block */
    fir_state_def *fir_init(int len, float *h);

    /* Processes a buffer of samples for the fir block */
    void fir(fir_state_def *s, const float x_in[], float y_out[]);

#endif /* _fir_h_ */
```

The fir.c is the script who has the init() and fir() block functions.

```c
/*****************************************************************
                                fir.c
 *****************************************************************/

// Libraries that were used in delay.c
#include <std.h>
#include <sys.h>
#include <dev.h>
#include <sio.h>


// Include the headers of the files needed
#include "fir.h"
```

```
#include "dsp_ap.h"



fir_state_def *fir_init(int len, float *h){
    /*  fir_init()
        This function initializes a fir block.
        Inputs:
        len = length of the filter
        *h = pointer to coefficients
        Returns:
        0       An error ocurred
        other   A pointer to a new delay structure */

    // Creating a struct named s
    fir_state_def *s;

    // Allocating te struct in te memory segment FIR_SEG_ID
    s = (fir_state_def *)MEM_calloc(FIR_SEG_ID, sizeof(
   fir_state_def), FIR_BUFFER_ALIGN);

    //Checking if the struct was created correctly and sending a
   message of error if not
    if(s == NULL){
        SYS_error("Unable to create an input delay floating-point
   buffer.", SYS_EUSER, 0);
        return(0);
    }

    // Setting all variables inside the struct, so it can be
   returned in one variable.
    s->len = len; //Length of the filter
    s->h = h;  //coefficients
    s->t = 0; // tail

    /* Success.  Return a pointer to the new state structure. */
    return(s);
}

void fir(fir_state_def *s, const float x_in[], float y_out[]){
    /*  fir_init()
        This Function was explained in task 5
```

```
        it initializes a fir block.
        Inputs:
        len = length of the filter
        *h = pointer to coefficients
        Returns:
        0       An error ocurred
        other   A pointer to a new delay structure */

    /* This Function was explained in task 5
         This loop walks between y_out array */
    for(int i = 0; i < BUFFER_SAMPLES; i++)
    {
        x_in[s->t] = s->h[i]; //Coefficients into buffer

        //updating tail in a circular way
        s->t = (s->t + 1) & FIR_BUFFER_CMASK;

        //updating ptr in a circular way
        int ptr = (s->t + FIR_BUFFER_SIZE - 1) & FIR_BUFFER_CMASK;
        float sum = 0;// sum variable needed

        /* This loop does the convolution*/
        for (int j = 0; j < s->len; j++)
        {
            sum = sum + s->h[j] * x_in[ptr]; //convolution of h&x
            //updating ptr in a circular way
            ptr = (ptr + FIR_BUFFER_SIZE - 1) & FIR_BUFFER_CMASK;
        }
        y_out[i] = sum; //FINAL RESULT in y_out[n]
    }
}
```

In the fir_init() function we create a struct and allocated in the memory and named it "s" and it pointer is "*s". Then we checked for errors. Finally, we insert all initial values to the struct "s" and return the address of the struct.

In the fir() function, we do convolution in a circular way (because we are working with buffers) between the x_in array and the coefficients stored in the received struct "s". The result is stored in the y_out array.

For more details see the comments that I added in the code.

Task 8: Similiar to dsp_ap.c[1] in your delay project, write your dsp_ap.c. You need to initialize your filter and implement your filter there.

The dsp_ap.h is the header of the dsp_ap.c block code, in here we define all the needed variables, structures, and function prototypes.

```c
/****************************************************************
                          dsp_ap.h
  Contains global definitions for your DSP application.
  Here you can control the size and number of
  audio buffers (if needed), the sample rate, and
  the audio source.
 ****************************************************************/

#ifndef _dsp_ap_h_
    #define _dsp_ap_h_

    #include "math.h"
    #include "aic23.h"
    #include "dsk_registers.h"

    /* DSP_SOURCE
    * -----------
    * The following lines control whether Line_In or Mic_In is
    * the source of the audio samples to the DSP board.  Use Mic_In
    * if you want to use the headset, or Line_In if you want to use
    * the PC to generate signals.  Just uncomment one of the lines
    * below.
    */
    //#define  DSP_SOURCE   AIC23_REG4_LINEIN
    #define  DSP_SOURCE   AIC23_REG4_MIC

    /* DSP_SAMPLE_RATE
    * ----------------
    * The following lines control the sample rate of the DSP board.
    * Just uncomment one of the lines below to get sample rates from
    * 8000 Hz up to 96kHz.
    */
    #define DSP_SAMPLE_RATE     AIC23_REG8_8KHZ
    //#define DSP_SAMPLE_RATE     AIC23_REG8_32KHZ
    //#define DSP_SAMPLE_RATE     AIC23_REG8_48KHZ
    //#define DSP_SAMPLE_RATE     AIC23_REG8_96KHZ
```

```

/*******************************************************/
/* You can probably leave the stuff below this line alone. */
/*******************************************************/

// Number of samples in hardware buffers.  Must be a multiple
 of 32.
#define BUFFER_SAMPLES    128

// Number of buffers to allocate.  Need at least 2.
#define NUM_BUFFERS     2

// Scale used for FP<->Int conversions
#define SCALE           16384

int dsp_init();
void dsp_process(const float inL[], const float inR[],  float
 outL[],   float outR[]);
#endif /* _dsp_ap_h_ */
```

The dsp_ap.c is the script who has the dsp_init() and dsp_process() block functions.

```
/*************************************************************
                dsp_ap.c
 *************************************************************/

    /*    We include the header dsp_ap. because
        is the header of this file.       */
#include "dsp_ap.h"

    /*    We include the header fir.h in order
        to implement the fir block            */
#include "fir.h"

    /*    We include the header rc1_taps.h in order
        to have acces to the rc1_taps array    */
#include "rc1_taps.h"

          // FIR struct states declared
fir_state_def *FIR_Left;
fir_state_def *FIR_Right;

```

```
        // Global Declarations.  Add as needed.
float mybuffer[BUFFER_SAMPLES];

int dsp_init(){

  /*
   dsp_init
    This function will be called when the board first starts.
    In here you should allocate buffers or global things
      you will need later during actual processing.
   Inputs:
    None
   Outputs:
    0 Success
    1 Error
  */

  // Initialize the Left FIR block for the left channel
  FIR_Left = fir_init(rc1_taps_LEN, rc1_taps);

  //Checks if it was initialized correctly
    if (FIR_Left == 0){
        /* Error */
        return(1);
    }

  // Initialize the Right FIR block for the right channel
  FIR_Right = fir_init(rc1_taps_LEN, rc1_taps);

  //Checks if it was initialized correctly
    if (FIR_Right == 0){
        /* Error */
        return(1);
    }

    /* Success */
    return(0)
}

void dsp_process(
  const float inL[],
  const float inR[],
```

```
  float outL[],
  float outR[]){

  /*
  * dsp_process
  * This function is what actually processes input samples
  * and generates output samples .
  * Inputs :
  * inL ,inR Array of left and Right input samples .
  * outL , outR Array of left and Right output samples .
  *
  * Outputs :
  * 0 Success
  * 1 Error
  */

  /*  Implements the left FIR block to the left input
    array, and stores it in the left out array    */
  fir(FIR_Left, inL[], outL[]);

  /*  Implements the right FIR block to the right input
    array, and stores it in the right out array    */
  fir(FIR_Right, inR[], outR[]);
}
```

In the dsp_init() function we initialize two FIR blocks, one for the left input channel and another one for the right channel. Also, it checks for error every time a block was initialized.

In the dsp_process() function implement the FIR blocks to the left and right channel.

For more details see the comments that I added in the code.

0.3 Conclusion

In conclusion for programming in C a dsp block we need to follow always the same procedure of the circular buffing and the object oriented programming. The only thing that changes is the dsp operation algorithm, but then is basicly the same structure for every implemented dsp operation. Also, I notice that programming a dsp block in C is really similar to do it in Matlab. The difference is that the code runs faster in C but is more complicated to program it than in Maltab, but after this lab I saw that in reality is not that dificult to code it, is basically the same structure than the Matlab code and also every time we call a function in C we do it in a more efficient way than in matlab because we use pointers to the struct instead of passing the state to every function. I didn't meet any problems in the lab.

Bibliography

[1] Kelan Garcia. DSP Lab Report 1: Delay in C, 2019.

[2] Fangning Hu. Delay and FIR (Matlab Part), 2015.

[3] Fangning Hu. Protected: Delay and FIR on DSP Board (DSK6713), 2015.

[4] Fangning Hu. DSP FIR in C, 2020.