LAB 3: PLL IMPLEMENTATION IN C

Jacobs University Bremen

CO27-300231 DSP & Communications Lab

Spring Semester 2020

Prof. Fangning Hu

Kelan Garcia

May 11, 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:

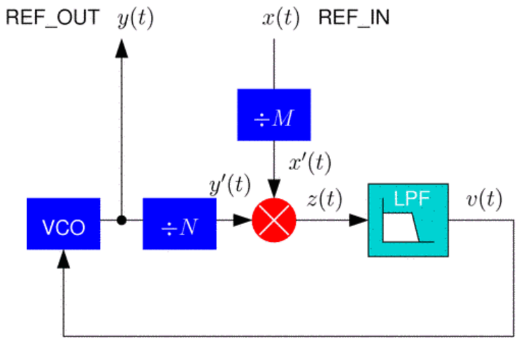
  1. blockname_init(): A function that initializes the block and returns a new state structure for the block.
  2. blockname(): A function that processes buffers of input samples to generate buffers of output samples.
  3. blockname_modify(): An optional function that allows the operation of the block 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: Review the PLL implementation which you learned in the Communiations lab[1] and describe each step.



The function of a PLL is to take an incoming sinusoid x(t) and generate an output sinusoid y(t) whose phase closely tracks the phase of the input.

The implementantion will be as follows:

(a) Create a Struct with all the values. (i.e. $[k, f, S, sintable, a_1, B_0, B_1,] \in$ Struct) This values are calculated with the initial given values k, f, S, D, w_0 via the following formulas:

$$\tau_1 = \frac{k}{\omega_0^2} \qquad a_1 = -\frac{T - 2\tau_1}{T + 2\tau_1} \quad b_1 = \frac{T - 2\tau_2}{T + 2\tau_1}$$
$$\tau_2 = \frac{2D}{\omega_0} - \frac{1}{k} \quad b_0 = \frac{T + 2\tau_2}{T + 2\tau_1}$$

(b) Inside the struct create a sinetable of size S = 1024. This is done because in this way we don't need to be calculating the sin of each phase each time, instead we just calculated once put in in a table and then we just look for the value at the table position depending on the phase in order to make it faster.

(c) Normalized the input $x[n]$ i.e. $x'[n] = \frac{x[n]}{amp+|x[n]|}$ where for the first iteration amp = 0.

(d) Multiply the normalized input $x'[n]$ with the output $y[n]$ (notice that if is first iteration then $y[0] = 0$). i.e. $z[n] = y[n] \cdot x[n]$

(e) Multiply the $z[n]$ signal with the IIR filter $F_{IIR}[n]$ i.e. $v[n] = z[n] \cdot F_{IIR}[n]$
The signal $v[n]$ is calculated with the following formula:

$$v[n] = a_1 \cdot v[n-1] + b_0 \cdot z[n] + b_1 \cdot z[n-1]$$

This is the same thing as $v[n] = z[n] \cdot F_{IIR}[n]$
Note that the values of a_1,$v[n-1]$, b_0, $z[n]$, b_1, and $z[n-1]$ are stored in the struct. The values for $v[n-1]$, $z[n]$, $z[n-1]$ at the first iteration are zero, but after the first iteration the values are updated.

(f) Calculate the accumulator that will be is used in the sine table later. The accumulator is used to obtained the position of the phase in the sinetable
Formula used for the accumulator:

$$accum[n+1] = accum[n] + f - k \cdot v[n] \cdot \frac{1}{2 \cdot \pi}$$
$$accum[n+1] = accum[n+1] - floor(accum[n+1])$$

(g) Create a sinusoidal output with the phase described by the accumulator i.e.

$$y_{out}[n] = sinetable[floor(accum[n+1] \cdot S)] \text{ Where S} = 1024$$

(h) Update the amplitude and the previous values that are stored in the struct.
For further explanation see the comments that I added in the implemented code.

Task 2: Write pll.c, pll.h and dsp_ap.c. Like in Delay and FIR project, the input data need to be processed block by block. A struct variable need to be defined to store the state. The state information need to be passed from the previous block to the current block. In pll.c, you need to allocate memory and implement the pll algorithm. In dsp_ap.c you need to initialize the necessary parameters and call the function pll

Answer:
Comments in each codeline were added in order to explain what I have done. First I provide te dsp_ap header then the .c file. Next I provide the pll header and then te .c file

```
/*****************************************************************
                             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

// Defining the value of PI, since it will be needed.
#define PI               3.141593


int dsp_init();
void dsp_process(const float inL[], const float inR[],  float
   outL[],   float outR[]);


#endif /* _dsp_ap_h_ */
```

```
/*****************************************************************
                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             */
```

```c
#include "pll.h"

          // PLL struct states declared
pll_state_def *PLL_Left;
pll_state_def *PLL_Right;

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

/*-------------------------------------------------------------
* 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
*-----------------------------------------------------------*/
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
  */

    //    Values needed for the PLL COMPONENTS

  int k = 1;            // Gain
  int D = 1;            // Damping Factor
  float f = 0.1;          // Frequency
  float w_o = 2 * PI * f * 0.1; // Corner Frequency
  int T = 1;            // Sample Period
  int S = 1024;         // # of values in the sine table
```

```

  // Initialize the Left PLL block for the left channel
  PLL_Left = pll_init(k,D,f,w_o,T,S);

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

  // Initialize the Left PLL block for the right channel
  PLL_Right = pll_init(k,D,f,w_o,T);

  //Checks if it was initialized correctly
  if(PLL_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 PLL block to the left input
   array, and stores it in the left out array    */
pll(PLL_Left, inL[], outL[]);

/*  Implements the right PLL block to the right input
   array, and stores it in the right out array    */
pll(PLL_Right, inR[], outR[]);

}
```

```
/******************************************************************
                              pll.h
           Header defines for implementing an FIR block.
 ******************************************************************/
#ifndef _pll_h_
    #define _pll_h_

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

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

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

    /*------------------- Structures -------------------*/
    typedef struct
    {
        //    Values needed for the PLL COMPONENTS
        int k;      // Gain
        float f;    // Frequency
        int S;      // # of values in the sine table


        //      Coefficients of the IIR Filter
        float a1;
        float b0;
        float b1;
```

```c

        //      Values of the functions during the PLL process
        float zp; // Multiplication between previous x_in and y_out
        float zc; // Multiplication between current x_in and y_out
        float vp; // Result of zp in the IIR filter
        float vc; // Result of zc in the IIR filter
        float y;  // Output value

        // needed variables
        float accum;//Accumulator,used for the frequency in the sin
    table.
        float amp_est;// Value used to normalizedb the current
   input
        float sine_table[1024];//Table with different values of
   sine.
    } pll_state_def;

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

    /* Initializes the PLL block */
    pll_state_def *pll_init(int k, int  D, float f, float w_o, int
   T, int S);

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


#endif /* _pll_h_ */
```

```c
/*****************************************************************
                                pll.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 "pll.h"
#include "dsp_ap.h"
```

```
#include "math.h" // needed fot floor() and abs()

pll_state_def *pll_init(int k, int D, float f, float w_0, int T, int S)
{
    /*  pll_init()
    This function initializes a pll block.
    Inputs:
    k = Gain,            D = Damping Constant,
    f = frequency,       w_0 = Corner Frequency,
    T = Sample Period,   S = # of values in the sine table
    Returns:
    0       An error ocurred
    other   A pointer to a new PLL structure */

    // Creating a struct named s
    pll_state_def *s;

    // Allocating te struct in te memory segment PLL_SEG_ID
    s = (pll_state_def *)MEM_calloc(PLL_SEG_ID, sizeof(pll_state_def), PLL_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->k = k;       // k = Gain
    s->f = f;       // f = frequency
    s->S = S;       // S = # of values in the sine table

    // All of the following values should be initialized in 0
    s->zp = 0;      // Multiplication between previous x_in and y_out
    s->zc = 0;      // Multiplication between current x_in and y_out
```

```
    s->vp = 0;       // Result of zp in the IIR filter
    s->vc = 0;       // Result of zc in the IIR filter
    s->y = 0;        // Output value
    s->accum = 0;    //Accumulator,used for the frequency in the sin
    table.

    //  This normalized with 1 at beginning
    s->amp_est = 1; // Value used to normalized the current input

    // tau1 and tau2 are helper values to calculate the time domain
    IIR filter.
    float tau1 = k / (w_0 * w_0);
    float tau2 = ((2 * D) / w_0) - 1/k;

    //this coefficients are used later to calculate the v[n]
    s->a1 = -(T - 2*tau1)/(T + 2*tau1);
    s->b0 = (T + 2*tau2)/(T + 2*tau1);
    s->b1 = (T - 2*tau2)/(T + 2*tau1);


    // Sine Table calculation  depending on the position i
    /*      It will create a table of sine starting at
        sine(2 * PI 1/1024)  until sine(2 * PI 1023/1024)   */
    for(int i = 0; i < S; i++)
    {
        s->sine_table[i] = sin(2 * PI * i/S);
    }

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

    /* Processes a buffer of samples for the PLL block */
void pll(pll_state_def *s, const float x_in[], float y_out[])
{
    float amp = 0; // Used for normalization on x_in
    for (int i = 0; i < BUFFER_SAMPLES; i++)
    {
        //Normalize the amplitude of input signal x_in[]
        amp = amp + abs(x_in[i]);   //amp added to previous amp
        x_in[i] = x_in[i] / s->amp_est; //Normalizing x_in[]

```

```
        // First two actions of the PLL block
        s->zc = s->y * x_in[i]; // z[n] =  y[n] * x[n]
        s->vc = s->a1 * s->vp + s->b0 * s->zc + s->b1 * s->zp; // v[n] = z[n] * IIR Filter

        // calculation of accumulator for the sin table
        s->accum = s->accum + s->f - (s->k * s->vc / (2 * PI)); // calculate same frequency
        s->accum = s->accum - floor(s->accum);

        // calculate output using sine table
        // objective is y_out[n + 1] = sin(2 * PI * acumm[n + 1])
        y_out[i] = s->sine_table[floor(s->accum * s->S)]; //Where S = 1024

        // update state variables
        s->y  = s->y;
        s->zp = s->zc;
        s->vp = s->vc;

    }
    //Actualize the amplitude value for next normalization.
    s->amp_est = amp/BUFFER_SAMPLES/(2/PI);
}
```

Task 3: Modify your pll.c to generate double of the input frequency and half of the input frequency.

Answer:
Comments in each codeline were added in order to explain what I have done.

I provide two pll.c files the first file shows how too produce double of the input frequency, the second pll.c file shows how to generate half of the input frequency.

```
/***************************************************************
             This pll.c generates double the input frequency
 ***************************************************************/

// 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 "pll.h"
#include "dsp_ap.h"
#include "math.h" // needed fot floor() and abs()

pll_state_def *pll_init(int k, int D, float f, float w_0, int T, int S)
{
    /*  pll_init()
    This function initializes a pll block.
    Inputs:
    k = Gain,           D = Damping Constant,
    f = frequency,      w_0 = Corner Frequency,
    T = Sample Period,  S = # of values in the sine table
    Returns:
    0       An error ocurred
    other   A pointer to a new PLL structure */

    // Creating a struct named s
    pll_state_def *s;

    // Allocating te struct in te memory segment PLL_SEG_ID
    s = (pll_state_def *)MEM_calloc(PLL_SEG_ID, sizeof(
```

```
pll_state_def), PLL_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->k = k;        // k = Gain
 s->f = f;        // f = frequency
 s->S = S;        // S = # of values in the sine table

 // All of the following values should be initialized in 0
 s->zp = 0;       // Multiplication between previous x_in and y_out
 s->zc = 0;       // Multiplication between current x_in and y_out
 s->vp = 0;       // Result of zp in the IIR filter
 s->vc = 0;       // Result of zc in the IIR filter
 s->y = 0;        // Output value
 s->accum2 = 0;   //Accumulator,used for the frequency in the sin table.

 //  This normalized with 1 at beginning
 s->amp_est = 1; // Value used to normalized the current input

 // tau1 and tau2 are helper values to calculate the time domain IIR filter.
 float tau1 = k / (w_0 * w_0);
 float tau2 = ((2 * D) / w_0) - 1/k;

 //this coefficients are used later to calculate the v[n]
 s->a1 = -(T - 2*tau1)/(T + 2*tau1);
 s->b0 = (T + 2*tau2)/(T + 2*tau1);
 s->b1 = (T - 2*tau2)/(T + 2*tau1);
```

```
    // Sine Table calculation  depending on the position i
    /*      It will create a table of sine starting at
        sine(2 * PI 1/1024)  until sine(2 * PI 1023/1024)   */
    for(int i = 0; i < S; i++)
    {
        s->sine_table[i] = sin(2 * PI * i/S);
    }

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

/* Processes a buffer of samples for the PLL block */
void pll(pll_state_def *s, const float x_in[], float y_out[])
{
    float amp = 0;  // Used for normalization on x_in
    for (int i = 0; i < BUFFER_SAMPLES; i++)
    {
        //Normalize the amplitude of input signal x_in[]
        amp = amp + abs(x_in[i]);   //amp added to previous amp
        x_in[i] = x_in[i] / s->amp_est; //Normalizing x_in[]
        // First two actions of the PLL block
        s->zc = s->y * x_in[i]; // z[n] =  y[n] * x[n]
        s->vc = s->a1 * s->vp + s->b0 * s->zc + s->b1 * s->zp; // v[n] = z[n] * IIR Filter

        // calculation of accumulator for the sin table
        s->accum2 = s->accum2 + 2 * (s->f - (s->k * s->vc / (2 * PI))); //calculate double the frequency
        s->accum2 = s->accum2 - floor(s->accum2);

        // calculate output using sine table
        // objective is y_out[n + 1] = sin(2 * PI * acumm[n + 1])
        y_out[i] = s->sine_table[floor(s->accum2 * s->S)]; //Where S = 1024

        // update state variables
        s->y  = s->y;
        s->zp = s->zc;
        s->vp = s->vc;

    }
```

```
    //Actualize the amplitude value for next normalization.
    s->amp_est = amp/BUFFER_SAMPLES/(2/PI);
}
```

```
/****************************************************************
            This pll.c generates half of the input frequency
 ****************************************************************/

// 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 "pll.h"
#include "dsp_ap.h"
#include "math.h" // needed fot floor() and abs()

pll_state_def *pll_init(int k, int D, float f, float w_0, int T,
   int S)
{
    /*  pll_init()
    This function initializes a pll block.
    Inputs:
    k = Gain,            D = Damping Constant,
    f = frequency,       w_0 = Corner Frequency,
    T = Sample Period,   S = # of values in the sine table
    Returns:
    0       An error ocurred
    other   A pointer to a new PLL structure */

    // Creating a struct named s
    pll_state_def *s;

    // Allocating te struct in te memory segment PLL_SEG_ID
    s = (pll_state_def *)MEM_calloc(PLL_SEG_ID, sizeof(
   pll_state_def), PLL_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->k = k;        // k = Gain
    s->f = f;        // f = frequency
    s->S = S;        // S = # of values in the sine table

    // All of the following values should be initialized in 0
    s->zp = 0;       // Multiplication between previous x_in and y_out
    s->zc = 0;       // Multiplication between current x_in and y_out
    s->vp = 0;       // Result of zp in the IIR filter
    s->vc = 0;       // Result of zc in the IIR filter
    s->y = 0;        // Output value
    s->accumhalf = 0;   //Accumulator,used for the frequency in the sin table.

    //  This normalized with 1 at beginning
    s->amp_est = 1; // Value used to normalized the current input

    // tau1 and tau2 are helper values to calculate the time domain IIR filter.
    float tau1 = k / (w_0 * w_0);
    float tau2 = ((2 * D) / w_0) - 1/k;

    //this coefficients are used later to calculate the v[n]
    s->a1 = -(T - 2*tau1)/(T + 2*tau1);
    s->b0 = (T + 2*tau2)/(T + 2*tau1);
    s->b1 = (T - 2*tau2)/(T + 2*tau1);


    // Sine Table calculation  depending on the position i
    /*      It will create a table of sine starting at
        sine(2 * PI 1/1024)  until sine(2 * PI 1023/1024)   */
    for(int i = 0; i < S; i++)
    {
```

```
        s->sine_table[i] = sin(2 * PI * i/S);
    }

    /* Success.  Return a pointer to the new state structure. */
    return s;
}
/* Processes a buffer of samples for the PLL block */
void pll(pll_state_def *s, const float x_in[], float y_out[])
{
    float amp = 0; // Used for normalization on x_in
    for (int i = 0; i < BUFFER_SAMPLES; i++)
    {
        //Normalize the amplitude of input signal x_in[]
        amp = amp + abs(x_in[i]);   //amp added to previous amp
        x_in[i] = x_in[i] / s->amp_est; //Normalizing x_in[]

        // First two actions of the PLL block
        s->zc = s->y * x_in[i]; // z[n] =  y[n] * x[n]
        s->vc = s->a1 * s->vp + s->b0 * s->zc + s->b1 * s->zp; // v
    [n] = z[n] * IIR Filter

        // calculation of accumulator for the sin table
        s->accumhalf = s->accumhalf + 1/2 * (s->f - (s->k * s->vc /
     (2 * PI))); //calculate half frequency
        s->accumhalf = s->accumhalf - floor(s->accumhalf);

        // calculate output using sine table
        // objective is y_out[n + 1] = sin(2 * PI * acumm[n + 1])
        y_out[i] = s->sine_table[floor(s->accumhalf * s->S)]; //
    Where S = 1024

        // update state variables
        s->y  = s->y;
        s->zp = s->zc;
        s->vp = s->vc;

    }
    //Actualize the amplitude value for next normalization.
    s->amp_est = amp/BUFFER_SAMPLES/(2/PI);
}
```

0.3 Conclusion

In conclusion for programming in C a dsp block we need to follow always the same procedure of the object oriented programming. 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. Also, I learned that the PLL is a very powerful tool that is actually very easy to code it, and thanks to the PLL we can demodulate received signals and makes us able to transmit and receive signals efficiently. I didn't meet any problems in the lab.

Bibliography

[1] Fangning Hu. Digital Phase Locked Loop (PLL): Matlab Part, 2015.