Kefil Tonouewa

11-06-2020

Dual Tone Multi Frequency Recognition

Digital Signal Processing – ECE 53800

Indiana University Purdue University Indianapolis

TABLE OF CONTENTS

Abstract

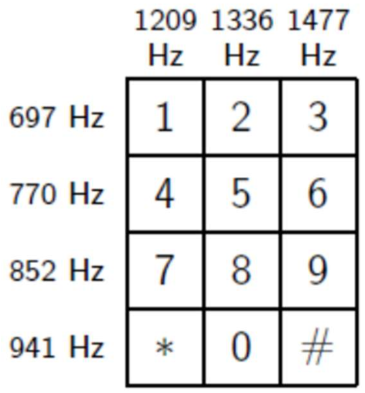
The design project that will be discussed is a Dual Tone Multi Frequency(DTMF) recognition. DTMF has been developed by the American company owned by NOKIA called Bell Labs. DTMF Represents the tones that is heard when pressing keypad buttons on a phone. The goal throughout this project is to detect the pressed key on the phone using the signal emitted by the key. In order to accomplish this goal, it was essential to use information about DTMF decoding alongside notions learned in ECE 53800 (Digital Signal Processing(DSP)). Two important notions from DSP which were used in throughout this project are Fast Fourier Transform (FFT) and sampling rate.

Dual Tone Multi Frequency Recognition

Introduction

How Does DTMF works?

Pressing a key sends a tone that represents a signal composed by two different frequencies. Every number have their own, predestined frequencies as seen in the table contained in the figure below (figure1). As it can be noticed in figure 1, the key representing “2” is composed of two frequencies which are 1336Hz and 697Hz. Therefore, when those two frequencies are detected in a signal, it can be concluded that the signal was generated by pressing “2” on the keypad.



	1209 Hz	1336 Hz	1477 Hz
697 Hz	1	2	3
770 Hz	4	5	6
852 Hz	7	8	9
941 Hz	*	0	#

Figure 1: DTMF Keypad

Hardware Equipment and Role

The following represents the Hardware used to accomplish the project:

Wolfson Audio Card: This will be used as a bridge between the signal and the microcontroller.

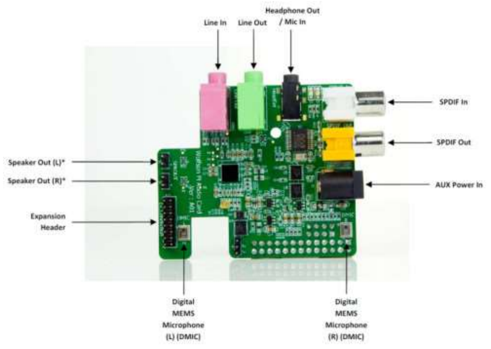


Figure 2.1: Wolfson Audio Card

Freedom Development Platform for Kinetis® K64:This will be the microcontroller used to receive and process the signal.

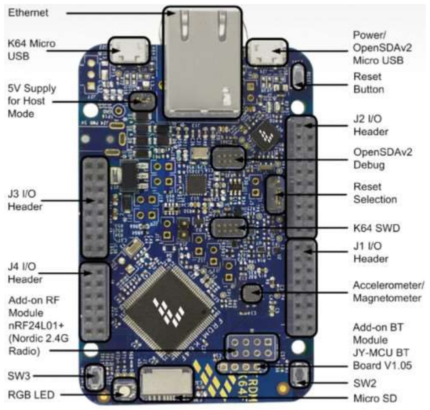


Figure 2.2: Freedom K64 Board

Seven Segment Display: This will be used to display the decoded signal as a digit/letter.

Figure2.3: Seven Segment Display

The following is an image representing the complete setup of the Hardware mentioned above:

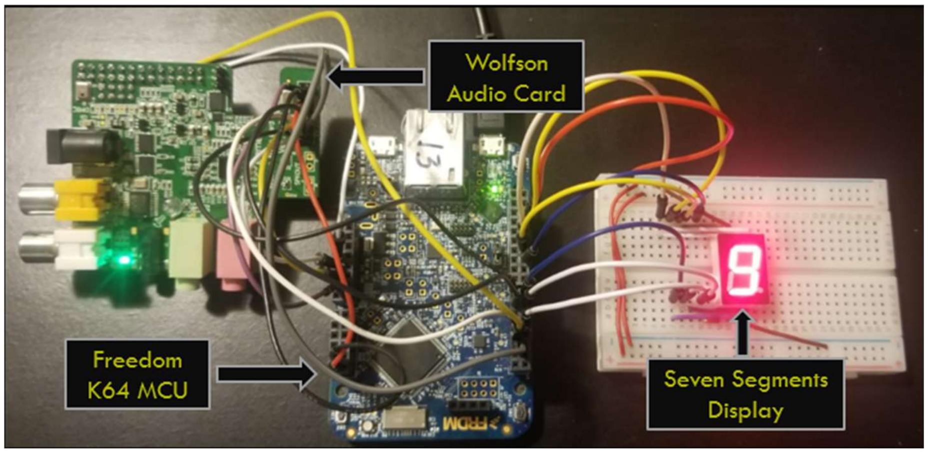
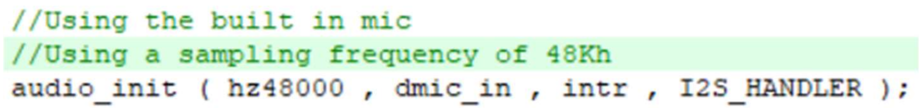


Figure 2.4: Hardware Setup

Software Description

In terms of Software used, Keil uVision 5 was utilized to implement the algorithm designed in order to program the microcontroller used to process the signal. The algorithm used to accomplish this project can be divided into steps which are the following:

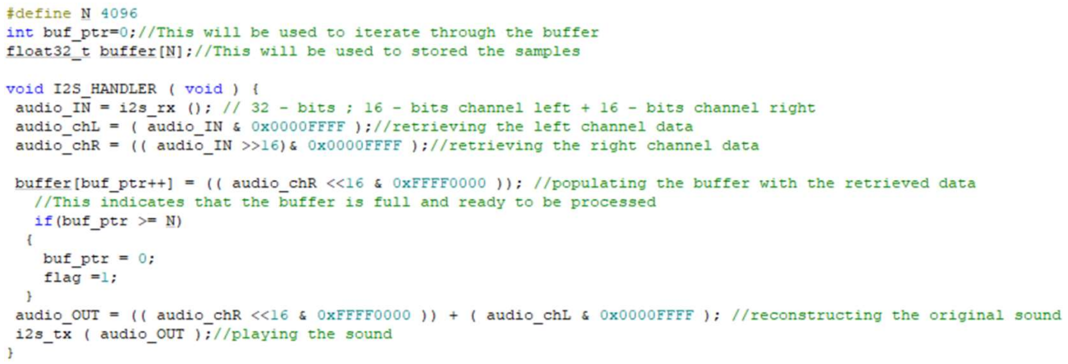
- Obtain the signal using Wolfson audio card either by using the building mic or the line in with a sampling frequency greater or equal to 2954 Hz. In this project, 48Khz was used as sampling frequency in order to accommodate for any computation error. Also, the built-in mic of the audio card was used to receive signals from the chosen device. The following represent the section of the software that took care of this portion of the algorithm:



```
//Using the built in mic
//Using a sampling frequency of 48Kh
audio_init ( hz48000 , dmic_in , intr , I2S_HANDLER );
```

Figure 3.1: Code Snippet of signal sampling

- Populate a buffer of N elements using the signal obtained. (NB: The minimum duration of a tone is 65ms) In this case, we used N = 4096 which implies that each sample will be recorded at a frequency rate of 11.72 Hz which correspond to a duration of 85.33ms. The following represents the section of the software that took care of this portion of the algorithm:



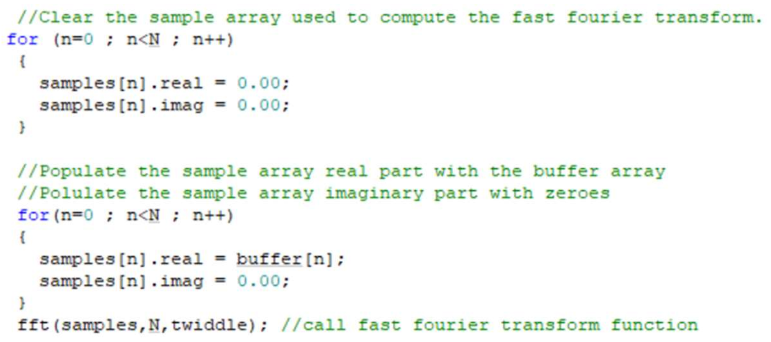
```
#define N 4096
int buf_ptr=0;//This will be used to iterate through the buffer
float32_t buffer[N];//This will be used to stored the samples

void I2S_HANDLER ( void ) {
 audio_IN = i2s_rx (); // 32 - bits ; 16 - bits channel left + 16 - bits channel right
 audio_chL = ( audio_IN & 0x0000FFFF );//retrieving the left channel data
 audio_chR = (( audio_IN >>16)& 0x0000FFFF );//retrieving the right channel data

 buffer[buf_ptr++] = (( audio_chR <<16 & 0xFFFF0000 )); //populating the buffer with the retrieved data
   //This indicates that the buffer is full and ready to be processed
   if(buf_ptr >= N)
  {
    buf_ptr = 0;
    flag =1;
  }
 audio_OUT = (( audio_chR <<16 & 0xFFFF0000 )) + ( audio_chL & 0x0000FFFF ); //reconstructing the original sound
 i2s_tx ( audio_OUT );//playing the sound
}
```

Figure 3.2: Code Snippet of buffer population

- Perform a Fast Fourier Transform on the obtained signal to get the frequencies. The following represents the section of the software used to compute FFT and how the computation of the FFT was set up:

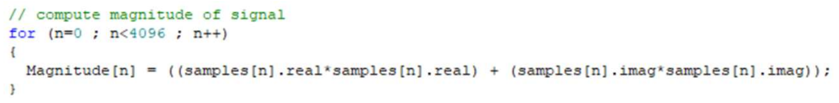


```
 //Clear the sample array used to compute the fast fourier transform.
for (n=0 ; n<N ; n++)
 {
   samples[n].real = 0.00;
   samples[n].imag = 0.00;
 }

 //Populate the sample array real part with the buffer array
 //Polulate the sample array imaginary part with zeroes
 for(n=0 ; n<N ; n++)
 {
   samples[n].real = buffer[n];
   samples[n].imag = 0.00;
 }
 fft(samples,N,twiddle); //call fast fourier transform function
```

Figure 3.3: Code Snippet of FFT computation

- Find the magnitude of samples issued from FFT. Finding this magnitude was just a matter of finding the sum between the real part of the sample squared and the imaginary part of the sample square. The following represent how the magnitude was computed in the software:



```
// compute magnitude of signal
for (n=0 ; n<4096 ; n++)
{
  Magnitude[n] = ((samples[n].real*samples[n].real) + (samples[n].imag*samples[n].imag));
}
```

Figure 3.4: Code Snippet of magnitude computation

After pressing 6 on the keypad and applying all the steps above, this is the result that is obtained:

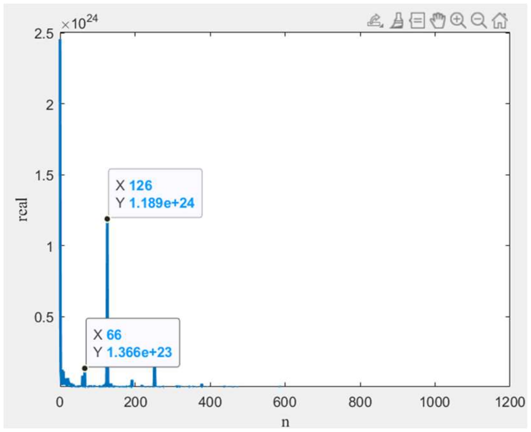


Figure 3.4: FFT spectrum of Key 6

As it can be seen, index 66 represents the lowest frequency of our signal and 126 represent the highest frequency of the signal.

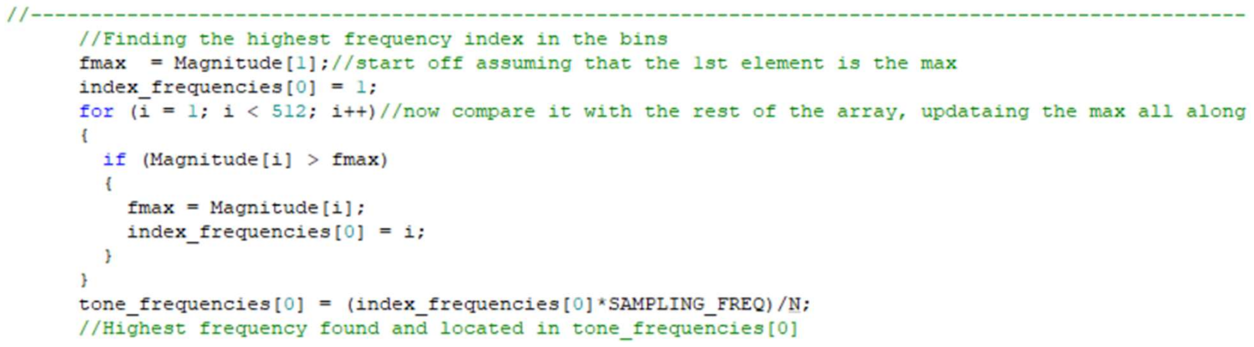
- Find the Highest frequency(FH) and the lowest frequency(FL). Using the value obtained in the previous step, we can find the highest and lowest frequency by performing the following operations.

$$FL = \frac{min_{index} \times f_{sampling}}{N} \qquad FH = \frac{ma\ \ _{index} \times f_{sampling}}{N}$$

Continuing with the example with key 6, a manual computation of FL and FH will yield the following results.

$$FL = \frac{66 \times 48000}{4096} = 773Hz \qquad FH = \frac{126 \times 48000}{4096} = 1476Hz$$

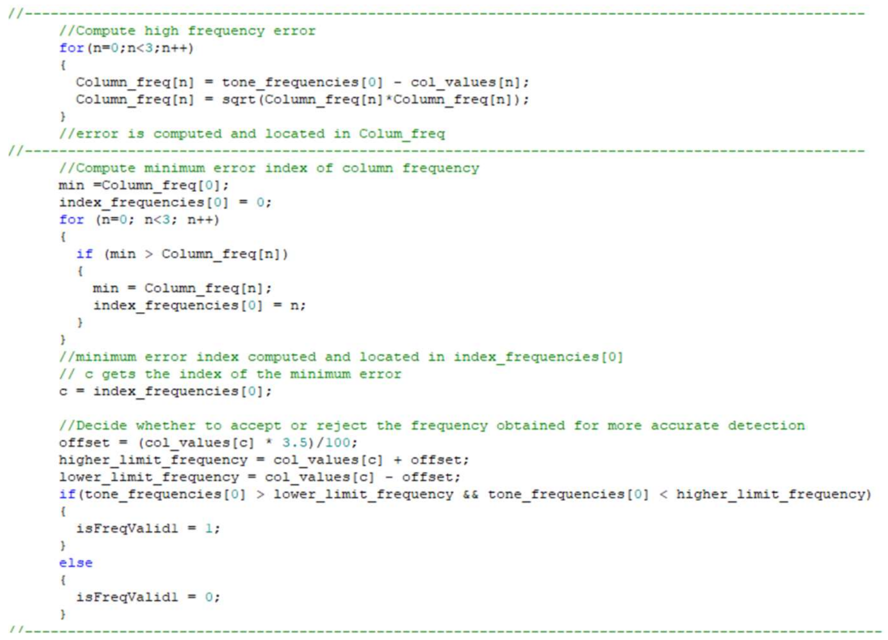
The following represent how FH:



```
//-----------------------------------------------------------------------------------------------
      //Finding the highest frequency index in the bins
      fmax  = Magnitude[1];//start off assuming that the 1st element is the max
      index_frequencies[0] = 1;
      for (i = 1; i < 512; i++)//now compare it with the rest of the array, updataing the max all along
      {
        if (Magnitude[i] > fmax)
        {
          fmax = Magnitude[i];
          index_frequencies[0] = i;
        }
      }
      tone_frequencies[0] = (index_frequencies[0]*SAMPLING_FREQ)/N;
      //Highest frequency found and located in tone_frequencies[0]
```

Figure 3.5: Code Snippet of Highest Frequency Computation

After finding the highest frequency in the signal, the presence of tone needs to be verified by checking if the frequency obtained is less than plus or minus 3.5% the expected frequency. Within the same scope, the column index corresponding to the obtained frequency is determined from the list of frequency representing the high frequencies in figure1. This operation is performed by doing the following:

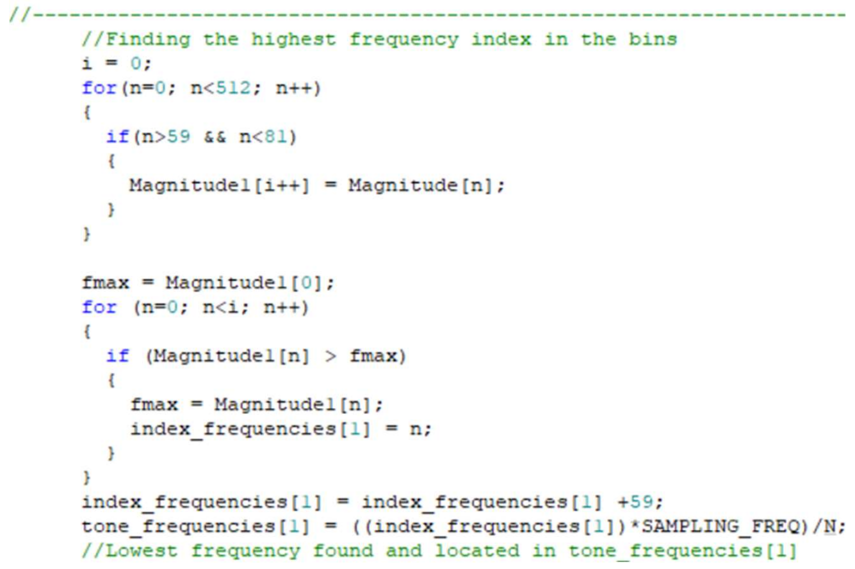


```
//---------------------------------------------------------------------------------------------
      //Compute high frequency error
      for(n=0;n<3;n++)
      {
        Column_freq[n] = tone_frequencies[0] - col_values[n];
        Column_freq[n] = sqrt(Column_freq[n]*Column_freq[n]);
      }
      //error is computed and located in Colum_freq
//---------------------------------------------------------------------------------------------
      //Compute minimum error index of column frequency
      min =Column_freq[0];
      index_frequencies[0] = 0;
      for (n=0; n<3; n++)
      {
        if (min > Column_freq[n])
        {
          min = Column_freq[n];
          index_frequencies[0] = n;
        }
      }
      //minimum error index computed and located in index_frequencies[0]
      // c gets the index of the minimum error
      c = index_frequencies[0];

      //Decide whether to accept or reject the frequency obtained for more accurate detection
      offset = (col_values[c] * 3.5)/100;
      higher_limit_frequency = col_values[c] + offset;
      lower_limit_frequency = col_values[c] - offset;
      if(tone_frequencies[0] > lower_limit_frequency && tone_frequencies[0] < higher_limit_frequency)
      {
        isFreqValid1 = 1;
      }
      else
      {
        isFreqValid1 = 0;
      }
//-----------------------------------------------------------------------------------------------
```

Figure 3.6: Code Snippet of Highest Frequency validation

The following represent how FL was determined:

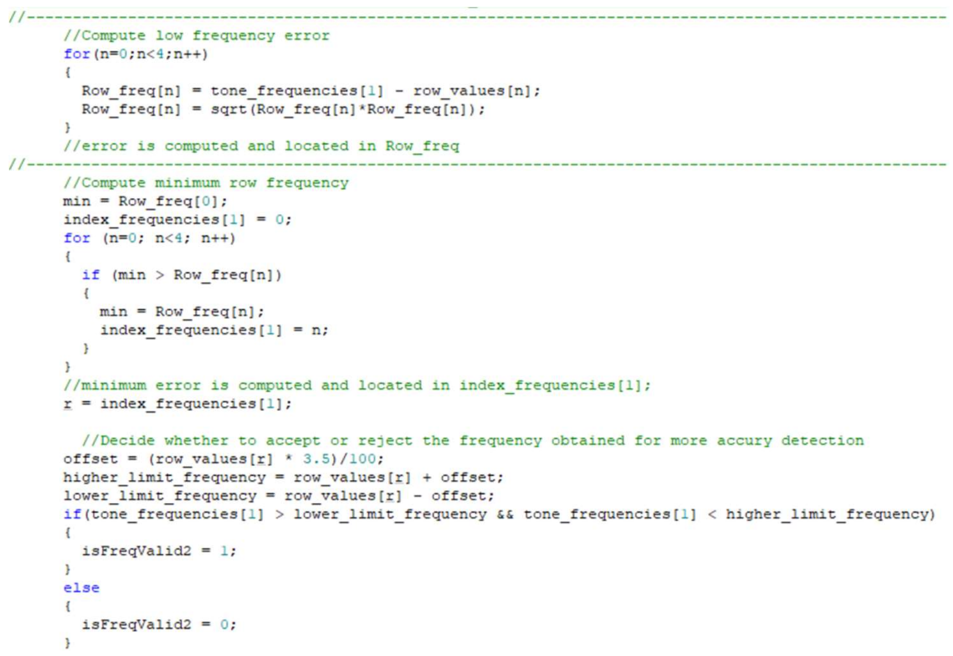


```
//------------------------------------------------------------------
      //Finding the highest frequency index in the bins
      i = 0;
      for(n=0; n<512; n++)
      {
        if(n>59 && n<81)
        {
          Magnitude1[i++] = Magnitude[n];
        }
      }

      fmax = Magnitude1[0];
      for (n=0; n<i; n++)
      {
        if (Magnitude1[n] > fmax)
        {
          fmax = Magnitude1[n];
          index_frequencies[1] = n;
        }
      }
      index_frequencies[1] = index_frequencies[1] +59;
      tone_frequencies[1] = ((index_frequencies[1])*SAMPLING_FREQ)/N;
      //Lowest frequency found and located in tone_frequencies[1]
```

Figure 3.7: Code Snippet of Lowest Frequency Computation

After finding the lowest frequency in the signal, the presence of tone needs to be verified by checking if the frequency obtained is less than plus or minus 3.5% the expected frequency. Within the same scope, the column index corresponding to the obtained frequency is determined from the list of frequency representing the low frequencies in figure1. This operation is performed by doing the following:

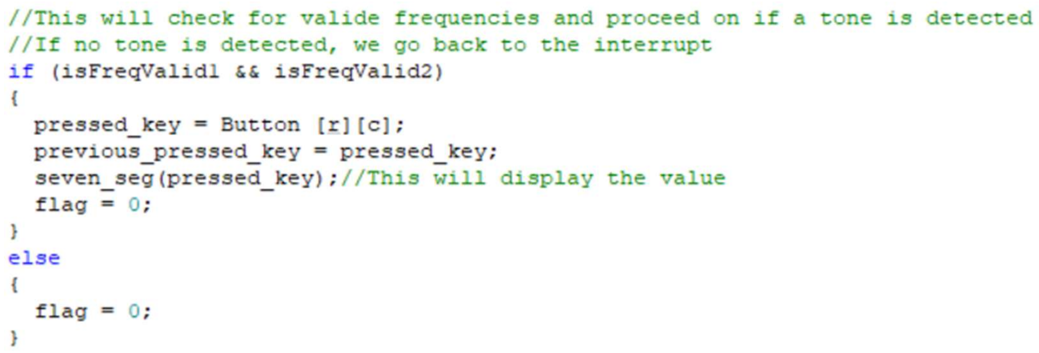


```
//-----------------------------------------------------------------------------------------------------
      //Compute low frequency error
      for(n=0;n<4;n++)
      {
        Row_freq[n] = tone_frequencies[1] - row_values[n];
        Row_freq[n] = sqrt(Row_freq[n]*Row_freq[n]);
      }
      //error is computed and located in Row_freq
//-----------------------------------------------------------------------------------------------------
      //Compute minimum row frequency
      min = Row_freq[0];
      index_frequencies[1] = 0;
      for (n=0; n<4; n++)
      {
        if (min > Row_freq[n])
        {
          min = Row_freq[n];
          index_frequencies[1] = n;
        }
      }
      //minimum error is computed and located in index_frequencies[1];
      r = index_frequencies[1];

        //Decide whether to accept or reject the frequency obtained for more accury detection
      offset = (row_values[r] * 3.5)/100;
      higher_limit_frequency = row_values[r] + offset;
      lower_limit_frequency = row_values[r] - offset;
      if(tone_frequencies[1] > lower_limit_frequency && tone_frequencies[1] < higher_limit_frequency)
      {
        isFreqValid2 = 1;
      }
      else
      {
        isFreqValid2 = 0;
      }
```

Figure 3.8: Code Snippet of Lowest Frequency validation

- Display the detected number to a seven segments display. At this point the two frequencies that compose the signal are know as well as their indexes in figure1. In addition to that the presence of tone is known. Therefore, if a tone is present, the values will be display and in the opposite case, no value will be displayed. The following represent the section of the software that cover this part of the algorithm:



```
//This will check for valide frequencies and proceed on if a tone is detected
//If no tone is detected, we go back to the interrupt
if (isFreqValid1 && isFreqValid2)
{
  pressed_key = Button [r][c];
  previous_pressed_key = pressed_key;
  seven_seg(pressed_key);//This will display the value
  flag = 0;
}
else
{
  flag = 0;
}
```

Figure 3.9: Code Snippet of Lowest Frequency validation

Flowchart

The following represents a quick summary of everything that has been discussed in the software description:

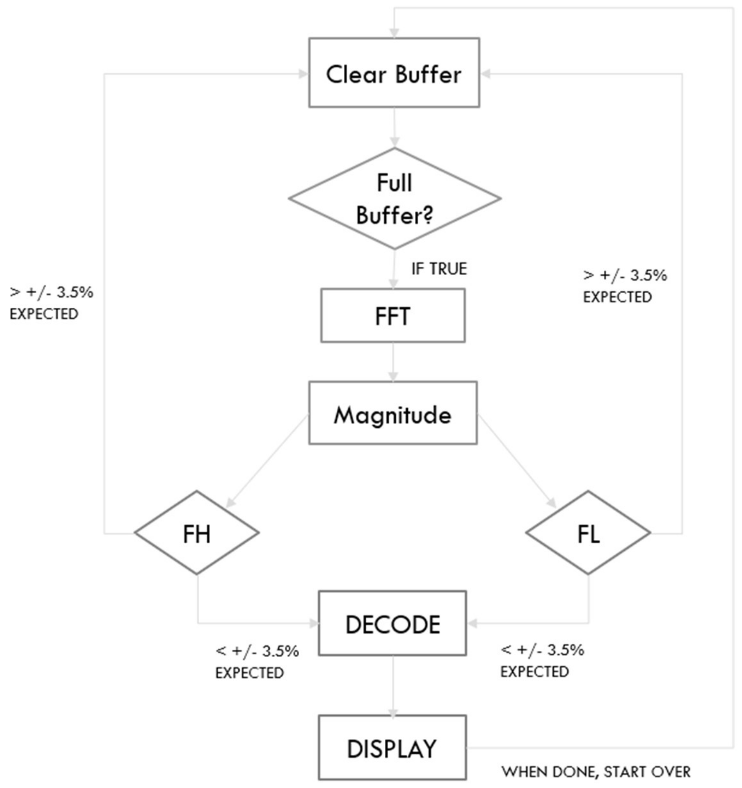


Figure 3.10: Flowchart of software description

Conclusion

The objective of this project was to put everything that students learned throughout the semester together in order to build a product that can be used in real life circumstance regarding Digital Signal Processing. In this case, the notion of Fast Fourier Transform was used to replicate a real-life example of digital signal processing. In the future, this project will be extended to interact with some home appliances and more. One thing to keep in mind is that this is not the only way to Recognize DTMF. In fact, one could design multiple low pass filters and high pass filters to solve this problem.