EE 321 Final Project Report

Theremin with Frequency Analyzer

Authors: D. Dodge, B. Willis

20-March-2018

Authorship: This lab was conducted by Daniel Dodge and Brian Willis.

Primary author of Methods and Results: Daniel Dodge

Primary author of Abstract, Introduction, Discussion, and Conclusion: Brian Willis

Abstract

This lab report contains the results of creating a Theremin and frequency analyzer. Our Theremin is played by a user's hand positions relative to a radio antenna, and can play notes in the 8th and 9th octaves. Our frequency analyzer can measure the frequency of an inputted waveform from 10 Hz to 20 kHz, and has a worst-case resolution of ±300 Hz. The measured frequency along with a calculated note and octave is displayed on a Liquid Crystal Display (LCD). In this report, we discuss our design process of an analog Theremin as well as a digital waveform analyzer with the struggles associated with each.

Equipment and Parts

- Tektronix MSO 2024 Mixed Signal Oscilloscope
- Agilent 33210A 10MHz Function / Arbitrary Waveform Generator (2)
- Topward Electric Instruments Co., LTD. Dual - Tracking DC Power Supply
- LM13700N Operational Transconductance Amplifier (OTA)
- 2N4401 NPN Bipolar Junction Transistor (BJT) (2)
- TS4990 Audio Amplifier
- LME49721 Operational Amplifier
- 5% capacitors
- 1% resistors
- 20% inductors
- Telescoping radio antenna
- 8 Ω speaker
- Freescale K65 Tower Board

1 Introduction

This inquiry-based project required us to create a signal-conditioning circuit that detects some quantity in the environment and then sample the conditioned signal on the Freescale K65 Tower Board's Analog to Digital Converter (ADC) such that it could be quantitatively displayed on the K65 board's LCD [1]. We decided to design and construct a Theremin with a frequency analyzer.

A Theremin is an electronic instrument that is controlled without physical contact by the user, but by electronic sensors reading some other kind of interference such as infrared light, solar light, or temperature. Our more conventional Theremin design detects electromagnetic interference from a user's hand positions relative to an antenna connected to the instrument. We created a finely tuned variable oscillator that can be adjusted by the natural capacitance of the human hand, especially when picked up

by the conductive antenna. The variable oscillator's output frequency is brought down to the audible frequency range (10 Hz - 20 kHz), then noise filtered and amplified before being output to a speaker. The frequency analyzer samples the waveform generated by the Theremin using ARM Digital Signal Processing (DSP) math functions. It displays not only the frequency of the Theremin's output, but its corresponding note and octave as well.

2 Methods

2.1 - Theremin Circuit Evolution

Our original circuit design for the Theremin project proposal is presented in Figure 2.1. In our proposed design, the Pitch Antenna is used as external manipulation for the Pitch Oscillator, which is tuned such that the minor capacitance of the human hand is enough to create a noticeable change in the oscillator's frequency. The Reference Oscillator is then mixed with the Pitch Oscillator to bring the resulting waveform down to the frequency band that humans can hear. The Low Pass Filter (LPF) removes any remaining high frequency noise from the signal before being amplified and output to a speaker. Additionally, the output from the audio amplifier is at the appropriate level such that the K65 Tower Board can analyze the incoming waveform for its frequency content, which is then displayed on the LCD.

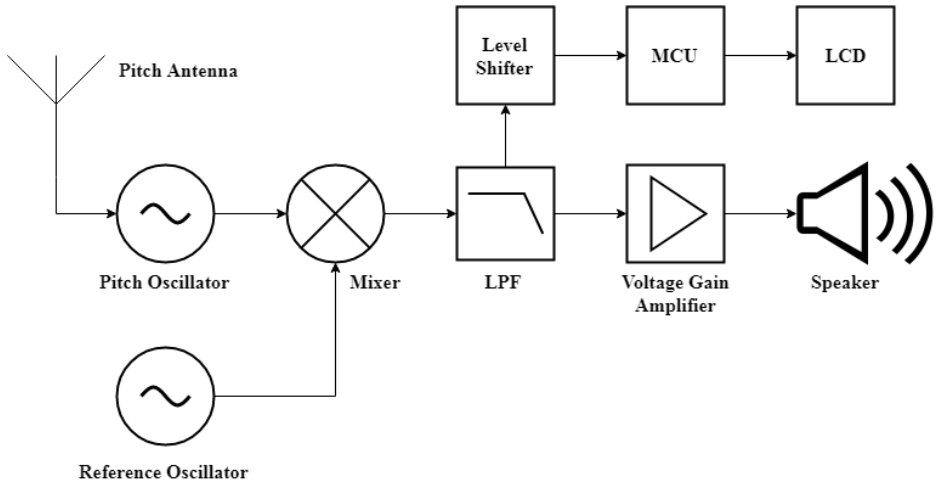

Figure 2.1 - Proposed Theremin Circuit

We noticed the effects of some non-idealities as the circuit design process continued and came up with a new Theremin circuit design, displayed in Figure 2.2. Each module of this new design is discussed below; it's important to note that we designed and tested each module individually before connecting modules together.

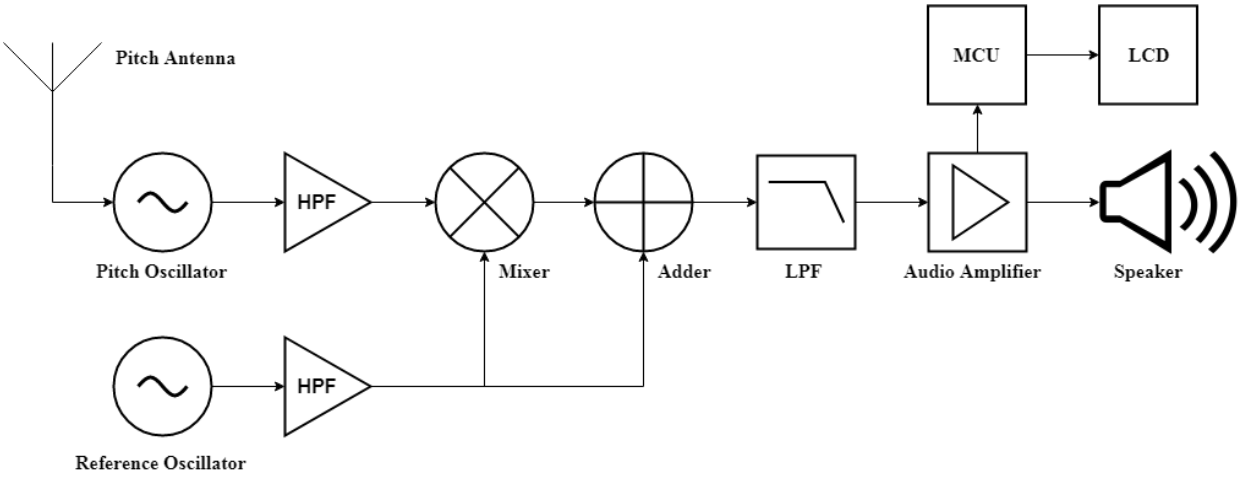

Figure 2.2 - Final Theremin Circuit

2.2 - Theremin Mixer

The mixer in Figure 2.3 is the first module that we designed and confirmed to be working. To achieve this we first simulated the design in Multisim and noted an expected waveform at the output around R3 for two input frequencies of 600 kHz and 590 kHz, see Figure 2.4. This design was used in another lab for Professor Klein, see [3] for in an in depth walkthrough on how to configure this mixer. In order to confirm the physical circuit we used two of the Agilent Function Generators.

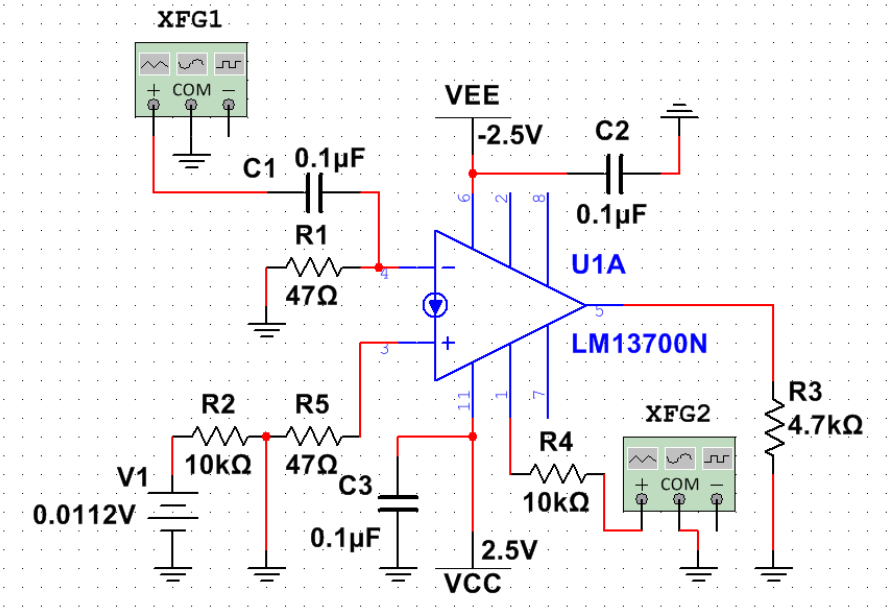

Figure 2.3 - Mixer where XFG1/2 Simulate Oscillators

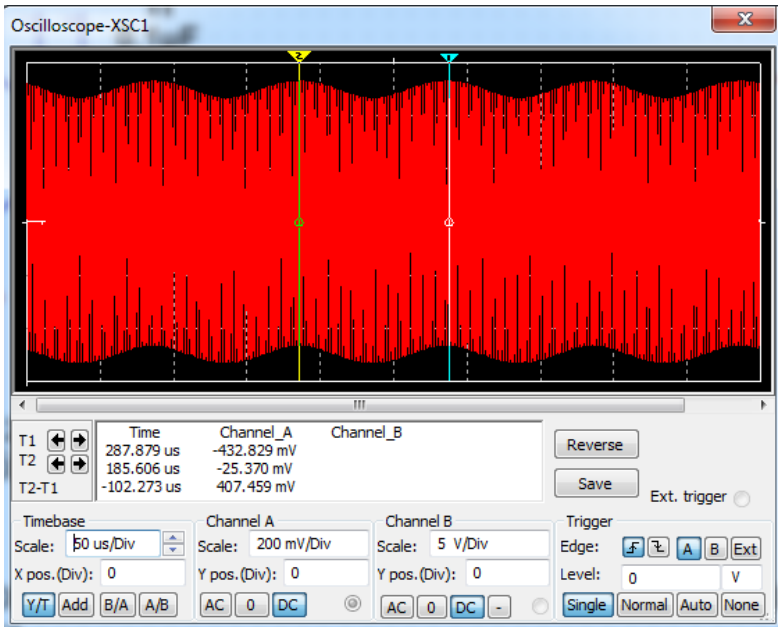

Figure 2.4 - Expected Mixer Output when V_{ref} = 600 kHz, V_{var} = 590 kHz

2.3 - Theremin Variable Oscillator

Both the variable and reference oscillators have the same base design, where the only differences are the reference oscillator has a tuning capacitor (0-60 pF) in parallel with C2 and the variable oscillator has an antenna connected to the top of C1. These oscillators were designed to have a base frequency of roughly 582 kHz. We can see this in the design below (Figure 2.5), by looking at the Colpitts LC tank (C1, C2, and L1) and Equation (1) which represents the resonant frequency. We subsequently see an expected frequency in Figure 2.6, which is roughly 571 kHz.

$$F_0 = \frac{1}{2\pi\sqrt{L\frac{C1C2}{C1+C2}}} \tag{1}$$

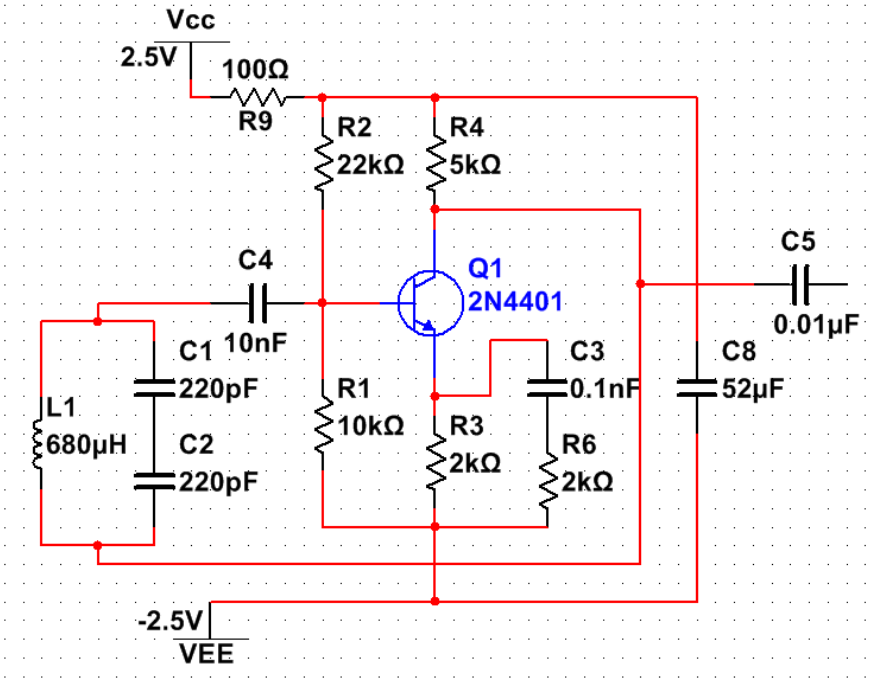

Figure 2.5 - Oscillator Design for Expected 571 kHz Output at C5

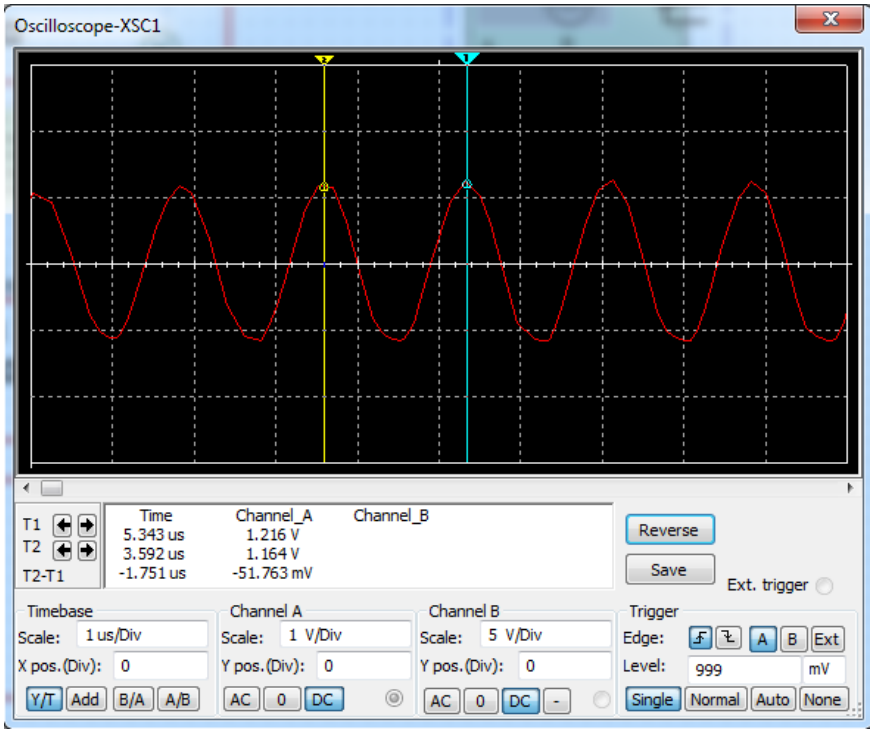

Figure 2.6 - Expected Oscillator Base Frequency of 571 kHz

The most important aspect of the Colpitts oscillator design is that the LC tank be placed in the feedback loop of the BJT or Op-Amp configuration used. The rest of the components in this circuit are placed to

help bias the NPN BJT where R1 and R2 provide a resistor divider to provide a voltage at the base. R3 provides a current for the emitter, while R6 and C3 provide a bypass branch. Finally, C5 is a DC decoupling capacitor for when the output signal goes into the mixer.

2.4 - Theremin Low Pass Filter

After a sine wave gets mixed, its frequency components shift up and down. The purpose of this Low Pass Butterworth filter is to attenuate all frequencies above 10 kHz so that when we mix our two oscillators the frequency result at the end of the filter is in the audible range. This design, Figure 2.7, was built during Lab 1 of EE 321 [4].

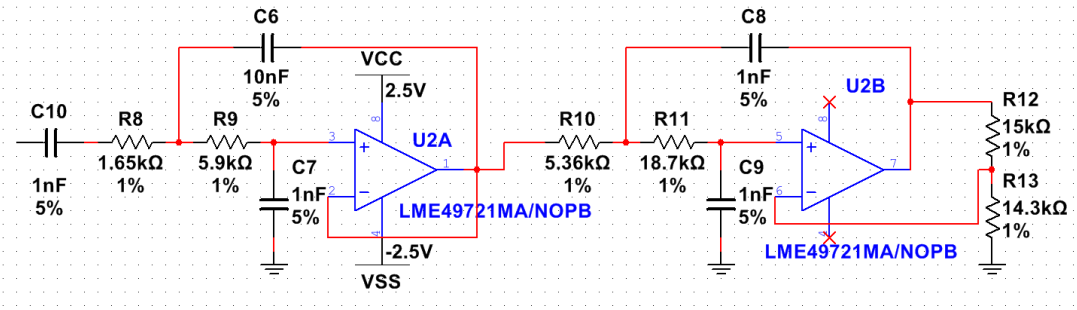

Figure 2.7 - 4th Order Low Pass Butterworth Filter with F_{cutoff} = 10 kHz

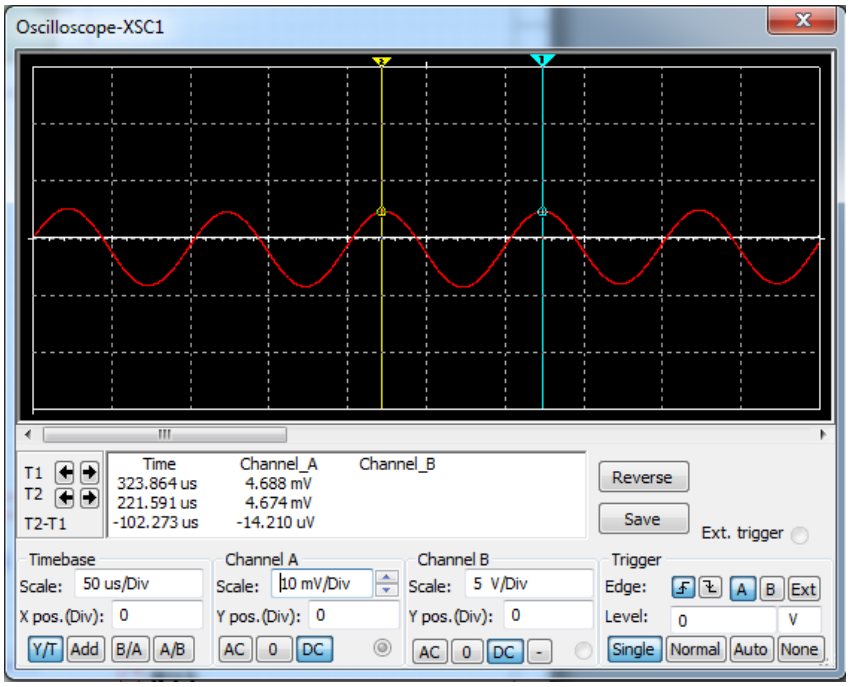

Figure 2.8 - Expected LPF Output when V_{ref} = 600 kHz, V_{var} =590 kHz

2.4 - Frequency Analyzer

The frequency analyzer takes the output from the Theremin circuit and measures its frequency content. Its code is presented in Appendices A and B, and its system flow diagram is presented in Figure 2.9. The ADC takes a predetermined number of samples of the incoming waveform, and then the program uses the ARM DSP Fast Fourier Transform (FFT) functions to find the frequency with the largest magnitude in the signal, which represents its fundamental frequency. The program repeats this step a predetermined number of times, then takes the average of all the found frequencies to attempt to correct for the amount of swing in each measurement. After averaging the measured frequencies, the program adjusts for the ADC's gain and offset errors. Figure 2.10 illustrates offset and gain errors for a Digital to Analog Converter (DAC).

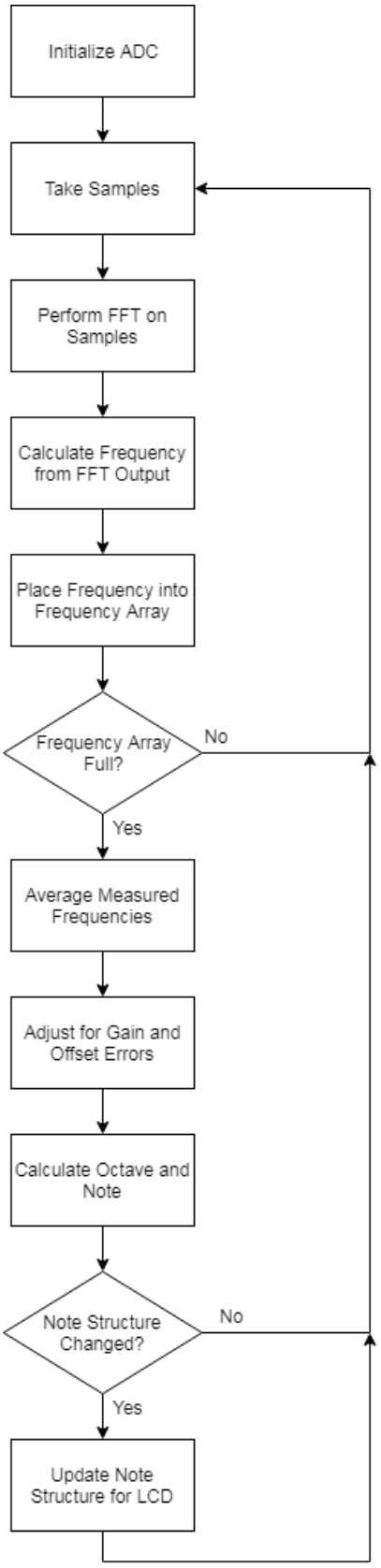

Figure 2.9 - Frequency Analyzer System Flow Diagram

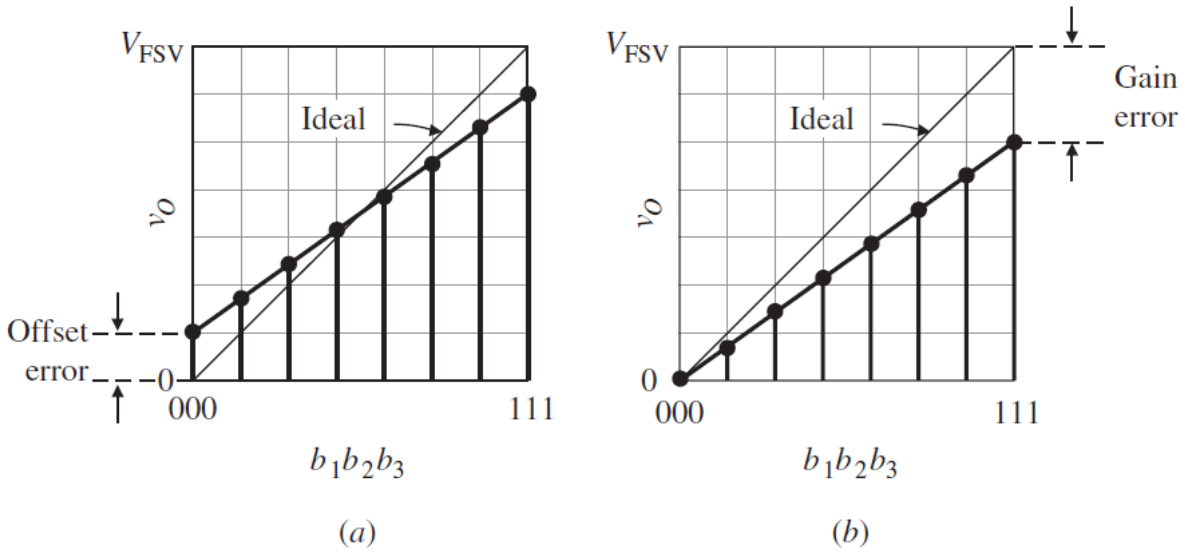

Figure 2.10 - Graphical Representation of DAC Offset (a) and Gain (b) Errors [2]

The program adjusts for the ADC errors in the same way as it would for DAC errors, exemplified in Equation (2). However, since the program's ADC outputs were not linear as in Figure 2.10, the corrections helped some frequencies become more accurate but hindered others.

$$ADC_{adjusted} = (ADC_{out} - Offset\ Error) * \frac{F_{FSV}}{(F_{FSV} - Offset\ Error + Gain\ Error)} \quad (2)$$

Where $ADC_{adjsuted}$ is error-corrected ADC output, ADC_{out} is the ADC output, and F_{FSV} is the Full Scale Voltage displayed in Figure 2.10. However, the offset and gain errors in this program's case are frequencies, not direct voltages from the ADC. Similarly, the F_{FSV} is 20 kHz, not 3.3 V. However, since the math works out the same, we believed this was an appropriate way to correct for errors.

Originally, we attempted to create a frequency analyzer without incorporating FFT math functions, which was not only difficult and time consuming but proved to be very inaccurate for very high and very low frequency signals. The original frequency analyzer sampled the signal and attempted to detect the number of peaks in a defined number of milliseconds in order to detect its frequency. However, due to noise in the signal with very low frequencies it would detect an extraordinary number of peaks, throwing off the calculations. Very high frequencies needed a sample rate of over twice that of standard audio sampling (44.1 kHz) in order to achieve any sort of accurate results. Implementing the ARM FFT functions proved to be the correct method of analyzing a signal's frequency content.

3 Results

3.1 - Oscillators

We found that the reference oscillator can range anywhere from 558 kHz to 580 kHz based on the tuning capacitor. The pitch oscillator rests at about 589.9 kHz and can drop to around 586.5 kHz depending on stray capacitance in the area. In Figure 3.1 we see the output of our reference oscillator, which has a frequency very close to the resting frequency of the pitch oscillator.

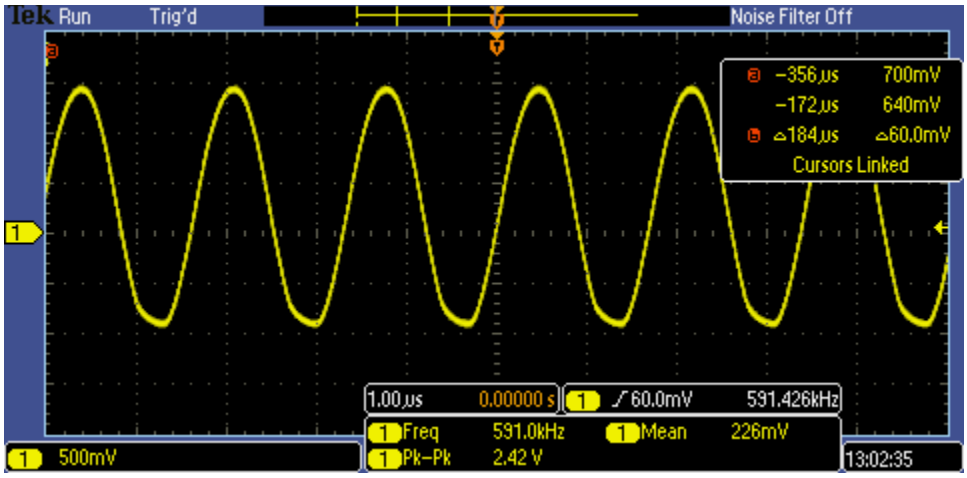

Figure 3.1 - Oscillator Output

3.2 - Mixer

In our original design, the output of the mixer was pure noise. To alleviate this, we added the High Pass Filters (HPF) to the inputs of the mixer to act as buffers that would give the input signals a high impedance. This helped the output the mixer by limiting current draw from the oscillators. In Figure 3.2 we can see the output of the mixer when the variable oscillator is at its resting frequency.

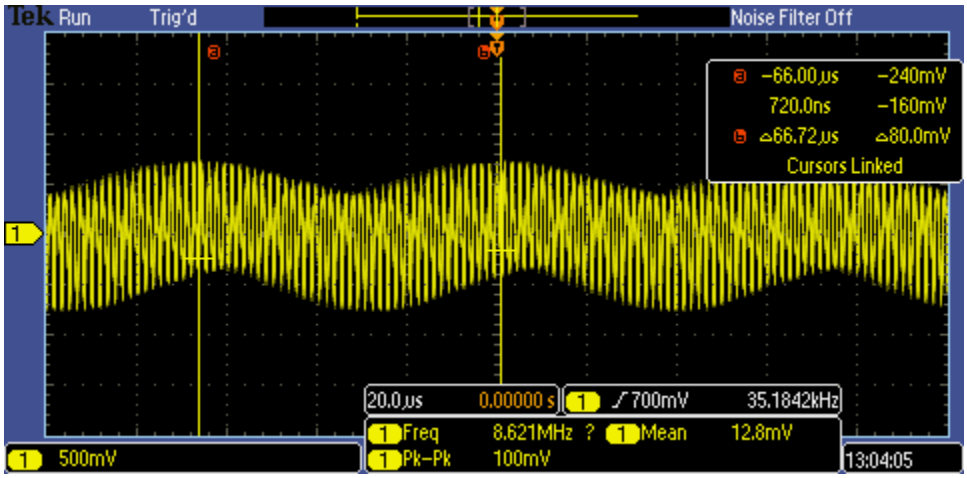

Figure 3.2 - Mixer Output 1

Figure 3.3 shows how the wave output changes when the pitch oscillator produces a different frequency.

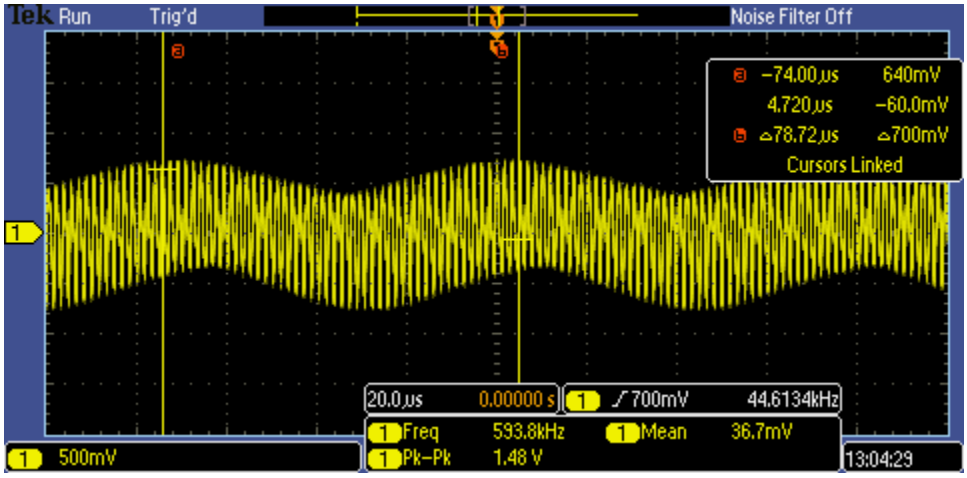

Figure 3.3 - Mixer Output 2

3.3 - Low Pass Filters

The output of the low pass filters worked as expected; the signal out of the filter was still low however so we had to add some gain into the audio amplifier module. Figure 3.4 shows the output of the 4th Order Low Pass Butterworth Filter.

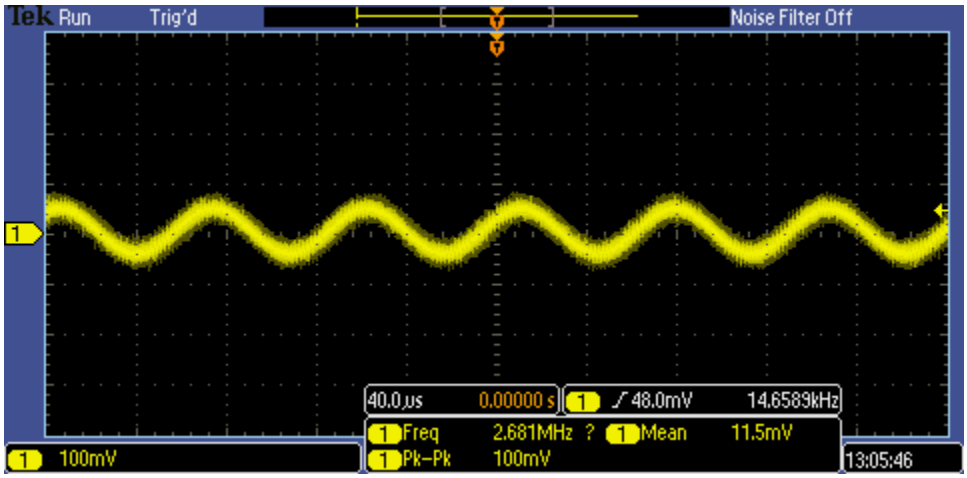

Figure 3.4 - LPF Output

3.4 - TS4990 Audio Amplifier

The audio amplifier provides a differential output signal which allows us to connect the speaker directly to the two outputs of the amplifier. Because this is a differential output we can also connect the MCU to the positive output (V_{out1}) of the amplifier and connect the MCU ground to common ground on the board. This is safe because the positive output voltage of the amplifier is at most 1.7 V and the whole wave is

above 0 V. Figure 3.5 shows the output of the audio amplifier when the pitch oscillator is at resting frequency, and Figure 3.6 shows the output when the pitch changes.

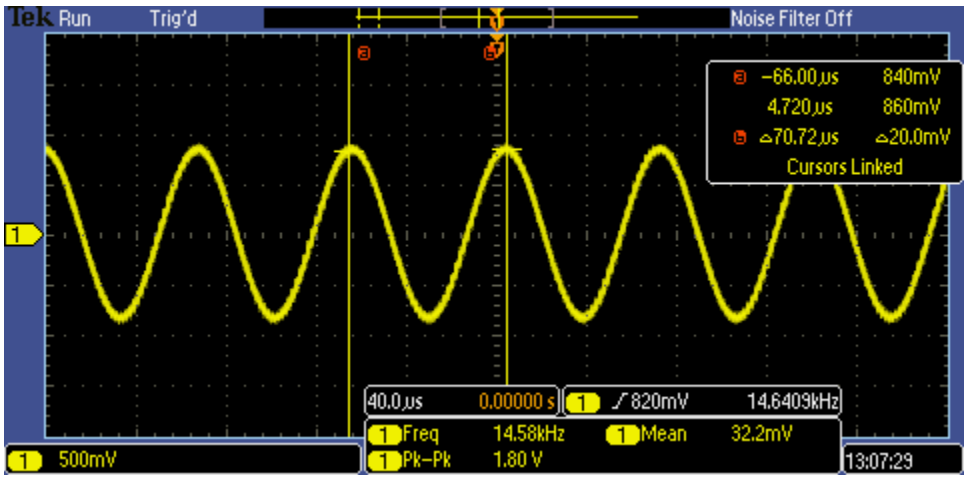

Figure 3.5 - Audio Amplifier Output 1

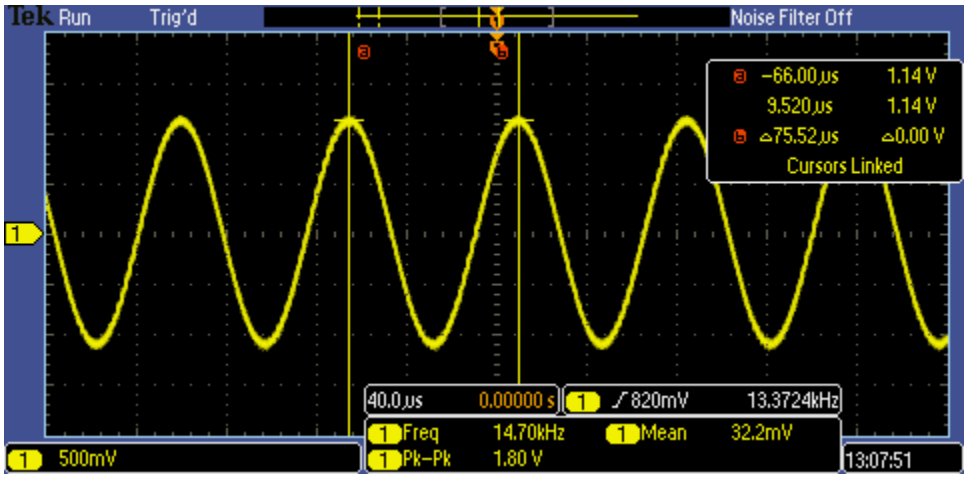

Figure 3.6 - Audio Amplifier Output 2, Lower Frequency

3.5 - Sampler

The frequency analyzer results in satisfactory outputs. Using a function generator as a tester, we compared the sampler's outputs to the generator's output and found that it was accurate up to about ±300 Hz. At low frequencies, this would be absolutely terrible for calculating the note and octave of the waveform as notes can differ by as little as 1 Hz and octaves by 15 Hz. However, our Theremin luckily operates in the 8th and 9th octaves, which have hundreds of Hz between notes and thousands of Hz between octaves.

Ideally, the resolution of the frequency analyzer is 43 Hz, as presented in Equation (3). However, non-idealities resulted in much larger errors than 43 Hz, which initially resulted in the creation of the averaging operation discussed in Section 2.5.

$$Resolution = \frac{Sampling\ Rate}{Re\{Samples\}} = \frac{44.1\ kHz}{1024} \approx 43\ Hz \quad (3)$$

Where Re{Samples} represents the real portions of the samples (imaginary portions are all 0).

4 Discussion

4.1 - Oscillators

The variable oscillator incorporates a Colpitts Oscillator with very small capacitors in its LC tank such that the minor capacitance of the antenna can adjust the frequency of the output waveform.
The reference oscillator is the exact same circuit as the variable oscillator, except for a tuning variable capacitor mentioned in Section 2.3. When the two oscillators' outputs are mixed therefore, the resulting waveform is brought relatively close to baseband (< 20 kHz) where humans' ears can pick up signals.
We expected signals closer to baseband (< 10 kHz) but struggled to adjust the reference oscillator to bring the mixer's frequency down further. Theoretically, a higher reference frequency would bring the output signal's frequency down, but more work is required to achieve this.

4.2 – Mixer

The mixer takes the variable oscillator's output, in the hundreds of kHz, and downshifts it to the audible spectrum (< 20 kHz). The adder is present in Figure 2.2 as we're using an LM13700N OTA IC, which was readily available and easy to use but provides a DC offset to the signal. The output of the mixer was not exactly as expected, as mentioned in Section 4.1, but we suspect the issue lies with the reference oscillator.

4.3 – Filters

The two oscillators were given HPFs to remove any DC signals and provide a high impedance output to the mixer. The Theremin's LPF filtered the mixer's carrier wave and any remaining high frequency signals and provided a high impedance output to the audio amplifier. The audio amplifier then boosted the signal to a max V_{pp} of 3.3 V and eliminated the DC portion of the signal. This allowed the speaker to receive a clean input and the ADC to receive an input of appropriate voltage magnitude.

The output of the filters were as expected when observing them on an oscilloscope, but the speaker seemed to have a significant amount of noise, which suggests we did not fully inspect the output of the audio amplifier.

4.2 - Sampler

The ADC samples the incoming waveform at a rate of 44.1 kHz, slightly above twice the maximum frequency the system is designed to detect, which is 20 kHz. This meets the Nyquist sampling rate requirement and results in a relatively clean and accurate ADC output. The offset and gain errors in the output from the ADC were as expected, as they're a common issue with converters. Luckily, they're correctable as discussed in Section 2.5.

We did not originally suspect the amount of swing in the frequency measurements mentioned in Section 4.5. We know that the problem is contained within the frequency analyzer however, as all tests were done with precise high impedance function generators. We made several attempts at improving the accuracy of the frequency measurements, such as increasing the sampling rate past 44.1 kHz, increasing the sample sizes, averaging the frequency samples, and adjusting for offset and gain errors (all discussed in Section 2.5). However, more work is required to adjust these parameters to result in a more consistent and accurate frequency analyzer.

5 Conclusion

Overall, we chose a fun but frustrating project to complete. We're very satisfied by our results, but if we had more time we would clean up the remaining noise coming from the speaker as well as alter our reference oscillator such that the Theremin's output waveforms were lower in the audible frequency band for a more pleasant listening experience. We would also further improve the frequency analyzer, especially the offset and gain correction factors which could use a more professional approach to finding. Creating a spreadsheet of all measured frequencies from a function generator for example would help immensely. However, the pitch oscillator was the most tedious and painstaking section to create, as it had to be finely tuned to not only pick up the minor capacitance from the user's hand but result in a low enough frequency such that the frequency shift bandwidth could be discerned by the human ear. We learned a lot about precise signal conditioning, non-ideal systems, and sampling tactics in the real world.

References

[1] A.G. Klein and T.D. Morton, “Final Project: Signal Conditioning and Analog-to-Digital Conversion,” Western Washington University, Winter 2017

[2] S. Franco, “Design with Operational Amplifiers and Analog Integrated Circuits, Fourth Edition,” San Francisco State University, 2015.

[3] A.G. Klein, “AM Transmission and Reception,” Western Washington University, Winter 2018

[4] T.D. Morton, “Active Filter Design,” Western Washington University, Winter 2018

Appendices

Appendix A. C Code Listing for ADC.c

```
/*****************************************************************
* ADC.c - Module for controlling ADC peripheral
* Uses DMA for sample output and PIT0 for sample trigger, based off
* Todd Morton's ADC demo for the K65 Tower Board.
*
* 03/20/2018 Brian Willis
*****************************************************************/
#include "MCUType.h"
#include "app_cfg.h"
#include "os.h"
#include "K65TWR_GPIO.h"
#include "ADC.h"

#define SAMPLE_RATE 44100       //Rate in Hz that ADC samples at
#define AVERAGING_PER 100       //Time in ms between frequency calculations

#define SAMPLES 2048            //1024 real parts and 1024 imaginary parts. Creates
                                //frequency resolution of 44100/1024 = 43Hz
#define FFT_SIZE SAMPLES/2      //FFT size is number of real samples

#define FREQ_AVG_SIZE 20        //Number of frequency calculations to take an average
                                //of for note output (1 = no averaging)

//Offset and gain errors from frequency calculations (found experimentally)
#define OFFSET_ERR 0                //Measured frequency at ~0Hz accurate
#define GAIN_ERR (30 + OFFSET_ERR)  //Measured frequency at 20kHz is 30Hz too high

FP32 Input[SAMPLES];
FP32 Output[FFT_SIZE];

static void ADCTask(void *p_arg);

//Private resources
static OS_TCB adcTaskTCB;                               //Allocate ADC Task control
                                                        //block
static CPU_STK adcTaskStk[APP_CFG_ADC_TASK_STK_SIZE];   //Allocate ADC Task stack
                                                        //space
static OS_SEM NoteChgFlag;
```

```
static INT32U adcFreq[FREQ_AVG_SIZE] = {0};             //Frequencies calculated from
                                                        //ADC's readings
static NOTE noteOut;

/*************************************************************************************
* ADCInit() - Initializes the ADC peripheral
* Uses ADC0 triggered by PIT1 to take samples of incoming waveform.
*************************************************************************************/
void ADCInit(){
    OS_ERR os_err;

    //Enable PIT1
    SIM_SCGC6 |= SIM_SCGC6_PIT_MASK;        //Start SCGC6 clock for PIT
    PIT_MCR &= ~PIT_MCR_MDIS_MASK;          //Enable PIT via MCR
    PIT_LDVAL1 = 60000000/SAMPLE_RATE;      //Set PIT1 sample rate
                                           //(Bus Clock = 60MHz)
    PIT_TCTRL1 |= PIT_TCTRL_TEN_MASK;       //Enable timer
    PIT_TCTRL1 |= PIT_TCTRL_TIE_MASK;       //Enable timer interrupt

    //Enable ADC0
    SIM_SCGC6 |= SIM_SCGC6_ADC0(1);         //Start SCGC6 clock for ADC0
    ADC0_CFG1 |= ADC_CFG1_ADIV(3);          //Divide bus clock by 8
    ADC0_CFG1 |= ADC_CFG1_MODE(3);          //Set to 16 bit samples
    ADC0_CFG1 |= ADC_CFG1_ADLSMP(1);        //Set to long samples
    ADC0_SC2 |= ADC_SC2_ADTRG(1);           //Set ADC0 for hardware trigger
    ADC0_SC3 |= ADC_SC3_AVGE(1);            //Enable hardware averager
    ADC0_SC3 |= ADC_SC3_AVGS(3);            //Set hardware averager to 32 samples
    SIM_SOPT7 |= SIM_SOPT7_ADC0TRGSEL(5);   //Set PIT1 as hardware trigger
    SIM_SOPT7 |= SIM_SOPT7_ADC0ALTTRGEN(1); //Set ADC0 to have alternate trigger

    PIT_TFLG1 |= PIT_TFLG_TIF_MASK;         //Reset PIT1 interrupt flag
    NVIC_ClearPendingIRQ(PIT1_IRQn);        //Clear PIT1 pending interrupts

    ADC0_SC1A = ADC_SC1_ADCH(3);            //Set input to DADP3

    do{
        ADC0_SC3 = ADC_SC3_CAL(1);                  //Begin calibration
        while((ADC0_SC3 & ADC_SC3_CAL(1)) == 1){}   //Wait for calibration
    } while((ADC0_SC3 & ADC_SC3_CALF(1)) == 1);     //Repeat if failed

    noteOut.note = "X";
    noteOut.oct = 255;
    noteOut.freq = 0;

    OSTaskCreate(&adcTaskTCB,                       //Create ADC Task
                 "ADC Task",
                 ADCTask,
                 (void *) 0,
                 APP_CFG_ADC_TASK_PRIO,
                 &adcTaskStk[0],
                 (APP_CFG_ADC_TASK_STK_SIZE / 10u),
                 APP_CFG_ADC_TASK_STK_SIZE,
                 0,
                 0,
```

```
                (void *) 0,
                (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
                &os_err);
    while(os_err != OS_ERR_NONE){}                  //Error Trap

    OSSemCreate(&NoteChgFlag, "Note Change Flag Semaphore", 0, &os_err);
    while(os_err != OS_ERR_NONE){}                  //Error Trap
}


/*********************************************************************************
 * ADCTask() - Controls the ADC peripheral
 * Uses ARM DSP functions to analyze incoming waveform and takes a running average of
 * results to achieve more accurate results.
*********************************************************************************/
static void ADCTask(void *p_arg){
    OS_ERR os_err;
    (void)p_arg;
    NOTE note_prev = noteOut;
    INT8U conv_cnt = 0;
    INT8U n = 0;
    INT16U dummy_freq = 0;

    arm_cfft_radix4_instance_f32 S;     //ARM CFFT module
    FP32 maxValue;                      //Max FFT value is stored here
    INT32U maxIndex;                    //Index in Output array where max value is

    while(1){
        for (INT16U i = 0; i < SAMPLES; i = i + 2) {
            while((ADC0_SC1A & ADC_SC1_COCO_MASK) == 0){}
            Input[(INT16U)i] = (FP32)ADC0_RA;   //Real part
            Input[(INT16U)(i + 1)] = 0;         //Imaginary part
        }

        //Initialize the CFFT/CIFFT module, intFlag = 0, doBitReverse = 1
        arm_cfft_radix4_init_f32(&S, FFT_SIZE, 0, 1);
        //Process the data through the CFFT/CIFFT module
        arm_cfft_radix4_f32(&S, Input);
        //Process the data through the Complex Magniture Module for calculating the
         //magnitude at each bin
        arm_cmplx_mag_f32(Input, Output, FFT_SIZE);

        //Zero out results that contain useless information
        Output[0] = 0;
        for(int j = FFT_SIZE/2; j < FFT_SIZE; j++){
            Output[j] = 0;
        }

        //Finds max magnitude in output spectrum with corresponding index
        arm_max_f32(Output, FFT_SIZE, &maxValue, &maxIndex);

        //Calculate frequency from location of max magnitude
        adcFreq[conv_cnt] = (SAMPLE_RATE*10000/(FFT_SIZE*10000/maxIndex));
        conv_cnt++;
```

```
//Take average of frequency samples
if(conv_cnt == FREQ_AVG_SIZE){
    noteOut.freq = 0;
    for(INT8U j = 0; j < FREQ_AVG_SIZE; j++){
        noteOut.freq = noteOut.freq + adcFreq[j];
    }
    noteOut.freq = noteOut.freq/FREQ_AVG_SIZE;
    conv_cnt = 0;

    //Adjust measured frequency for offset and gain errors
    noteOut.freq = noteOut.freq - OFFSET_ERR;
    noteOut.freq = (noteOut.freq*20000)/(20000 + GAIN_ERR - OFFSET_ERR);

    //Find note and octave of measured frequency
    //Find n in freq/2^n <= 31.87, where largest n = octave
    //0th octave ends at 31.87Hz
    //Multiply by 100Hz for 2 decimal points of precision
    while(((noteOut.freq*100)/(1<<n)) > 3187){
        n++;
    }
    noteOut.oct = n;
    n = 0;

    //Find note from frequency downshifted to 0th octave
    //Multiply by 100Hz for 2 decimal points of precision
    dummy_freq = (noteOut.freq*100)/(1<<noteOut.oct);
    if(dummy_freq < 1683){
        noteOut.note = "C";
    } else if((dummy_freq >= 1638) && (dummy_freq < 1783)){
        noteOut.note = "C#";
    } else if((dummy_freq >= 1783) && (dummy_freq < 1890)){
        noteOut.note = "D";
    } else if((dummy_freq >= 1890) && (dummy_freq < 2002)){
        noteOut.note = "D#";
    } else if((dummy_freq >= 2002) && (dummy_freq < 2121)){
        noteOut.note = "E";
    } else if((dummy_freq >= 2121) && (dummy_freq < 2247)){
        noteOut.note = "F";
    } else if((dummy_freq >= 2247) && (dummy_freq < 2381)){
        noteOut.note = "F#";
    } else if((dummy_freq >= 2381) && (dummy_freq < 2523)){
        noteOut.note = "G";
    } else if((dummy_freq >= 2523) && (dummy_freq < 2673)){
        noteOut.note = "G#";
    } else if((dummy_freq >= 2673) && (dummy_freq < 2832)){
        noteOut.note = "A";
    } else if((dummy_freq >= 2673) && (dummy_freq < 3000)){
        noteOut.note = "A#";
    } else if((dummy_freq >= 3000) && (dummy_freq <= 3178)){
        noteOut.note = "B";
    } else{}
} else{}

//If note structure updated, set flag
if((*noteOut.note != *note_prev.note)
```

```
            || (noteOut.oct != note_prev.oct)
            || (noteOut.freq != note_prev.freq)){
            OSSemPost(&NoteChgFlag, OS_OPT_POST_1, &os_err);
            while(os_err != OS_ERR_NONE){}          //Error Trap
        } else{}
        note_prev = noteOut;
    }
}

/*****************************************************************************
 * NotePend() - Sets main module's note to ADC module's note when note updates
*****************************************************************************/
void NotePend(NOTE *new_note){
    OS_ERR os_err;

    //Wait for note to be updated
    OSSemPend(&NoteChgFlag, 0, OS_OPT_PEND_BLOCKING, (CPU_TS *)0, &os_err);
    while(os_err != OS_ERR_NONE){}      //Error Trap

    *new_note = noteOut;                //Update note for LCD screen
}
```

Appendix B. C Code Listing for ADC.h

```
/*****************************************************************
* ADC.h - Header file for ADC module
*
* 03/20/2018 Brian Willis
*****************************************************************/
#ifndef ADC_H_
#define ADC_H_

//Note structure
typedef struct{
    INT8C *note;
    INT8U oct;
    INT32U freq;
} NOTE;

void ADCInit(void);
void NotePend(NOTE *new_note);

#endif
```