6.115 Final Project Report:

Theremin Number 29

Miguel Talamantez

BACKGROUND, INSPIRATION, AND INTRODUCTION

Why did I chose this project?

I’ve always loved music, instruments, but most importantly I also love, electromagnetism, conduction, and microcomputers. The obvious choice of project was a theremin. I’ve played a theremin before and each time if feels like some sort of magic. I wanted to recreate this using PSoC and show the skills I’ve learned over the semester as an aspiring embedded systems engineer.

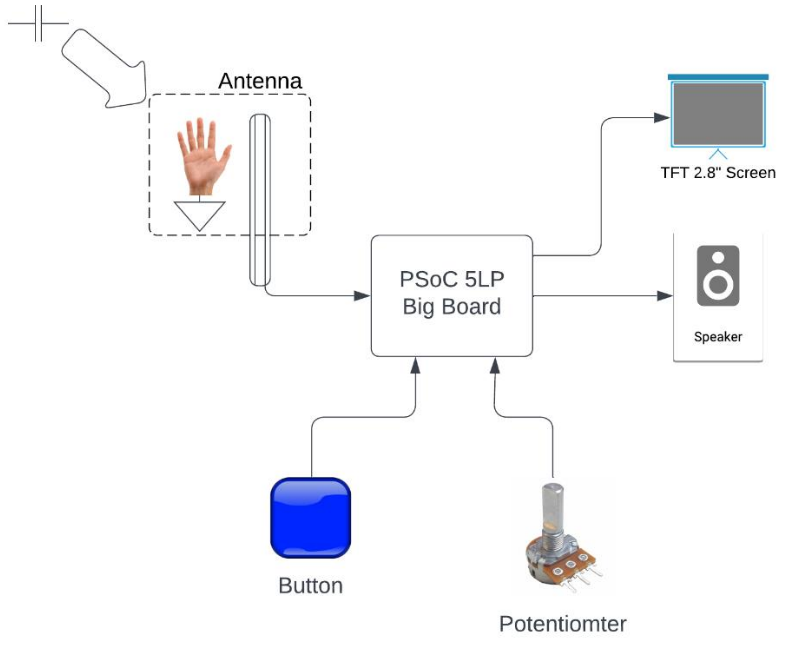
What is a theremin and how does it work?

The theremin is a musical instrument that uses a hand to vary the output frequency to achieve different notes. The original circuit uses an oscillator called a colpitts oscillator which is essentially a driven RLC circuit that uses a hand as a capacitor to change the frequency of its oscillation. Treating a hand as one plate of a capacitor and an antenna as the other plate, your body can be treated as essentially ground and change the capacitance value of the oscillator circuit. This frequency must be very high in order to observe a noticable difference in frequency. A typical theremin is around 270kHz in the colpitts circuit with an observable frequency change of around 10kHz. To make this frequency audible, the output of the colpitts circuit undergoes heterodyning with another fixed frequency which brings down the frequency to around 270Hz. This entire process generates a frequency output of varying pitch thus creating a range of notes. Another antenna is usually used to vary the volume which has similar layout as the pitch.

What is my project?

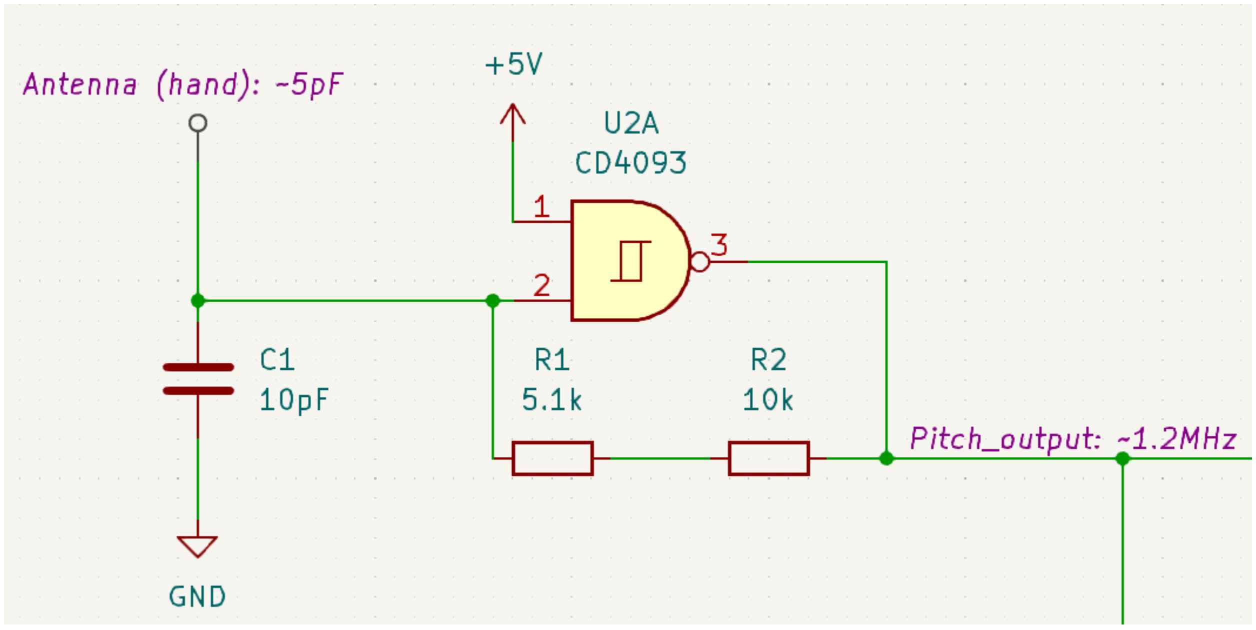
The original theremin schematic is purely analog with no need for integration of a microcomputer/embedded system, so for my project, my goal was to digitize a theremin using a psoc. Instead of sine waves, I work with square waves in order to effectively integrate PSoC into my design. Theremin Number 29 varies a frequency of 1MHz using a square wave oscillator which is then read by the PSoC, scaled down to audible frequencies, and played into a speaker. While the theremin is running there is a TFT screen that displays the frequency of the oscillator circuit, the output frequency, the musical note that is being played accordingly, and the theremin mode. There are two modes on the theremin. The first mode is the default theremin setup, and when a button is pressed, the theremin switches to a fixed frequency output mode in which a sine wave tuning note C5 is played. The purpose of this mode is to give the users ears a break from nasty square wave sounds with a perfectly in tune and smooth note.

Project Block Diagram



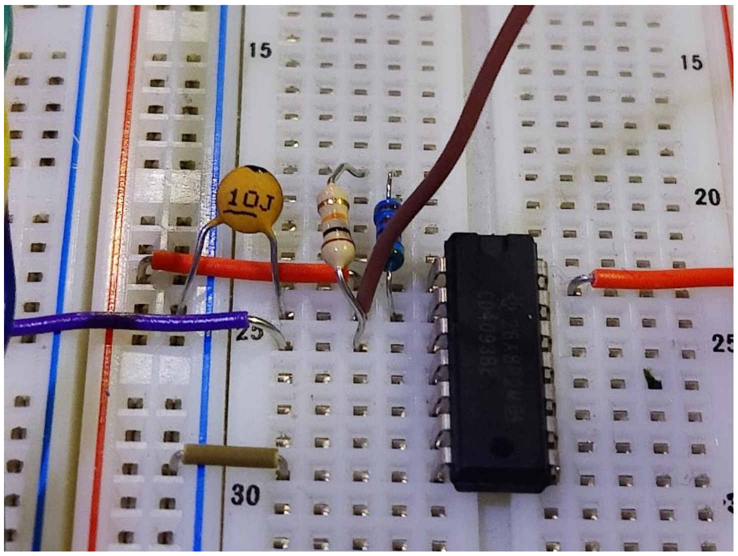
MODULE 1: DIGITAL IMITATION OF COLPITTS OSCILLATOR

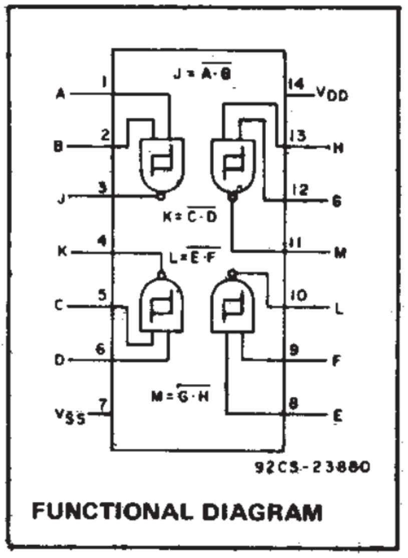
When I started the project, I did some research on how a PSoC can measure frequency. I discovered that there are various ways to measure frequency, but almost all of them prefer that the wave is a square wave. After analyzing different options of creating an oscillator, I decided to use a NAND gate and a very small capacitor to create a high frequency square wave that can be varied using a change in capacitance that's very small (a hand).

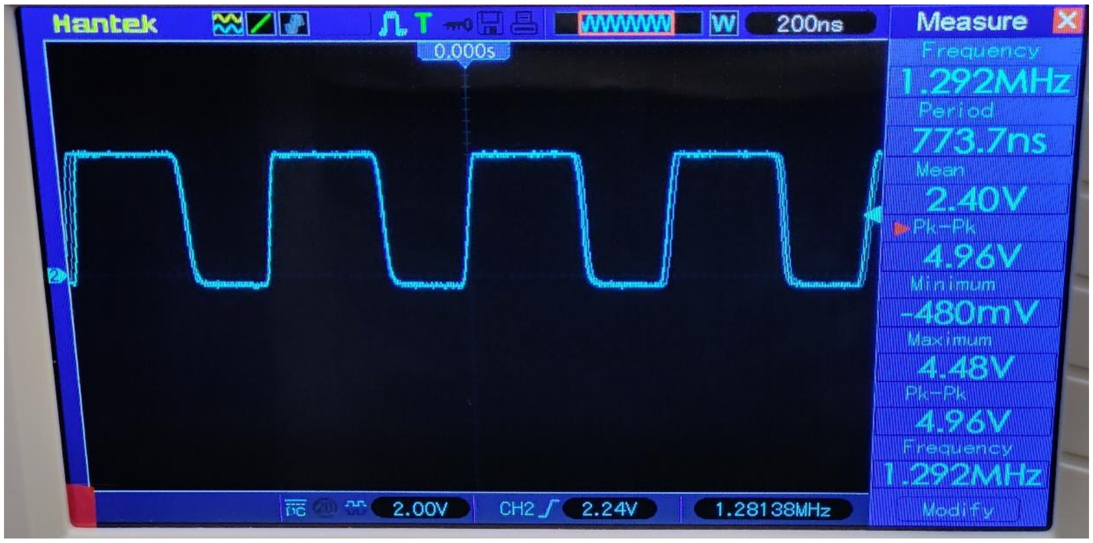


When the system first starts, terminal 1 of the NAND gate is high and terminal 2 is low, this outputs high on terminal 3 (the output). This charges the capacitor to a voltage that the terminal 2 eventually reads as high. As both terminals 1 and 2 read as high, output 3 goes low, and the cycle continues creating a square wave by constantly charging and decharging the capacitor. If you vary the capacitor value, you as a result vary the frequency output of the squarewave being generated. Because I've chosen such a small capacitor value (10pF), the output frequency is extremey high at around 1.2MHz. I chose this high frequency because it allows for a larger change in frequency as the antenna capacitance value changes. This creates a more noticable change in frequency when it's scaled down.

Although I originally wanted to do a complete colpitts circuit then later convert the sine wave into a square wave, this is my most critial module, so I decided to make it as foolproof as possible. In future iterations, I could attempt to use the original colpitts style oscillator since I already have the foundations in place.



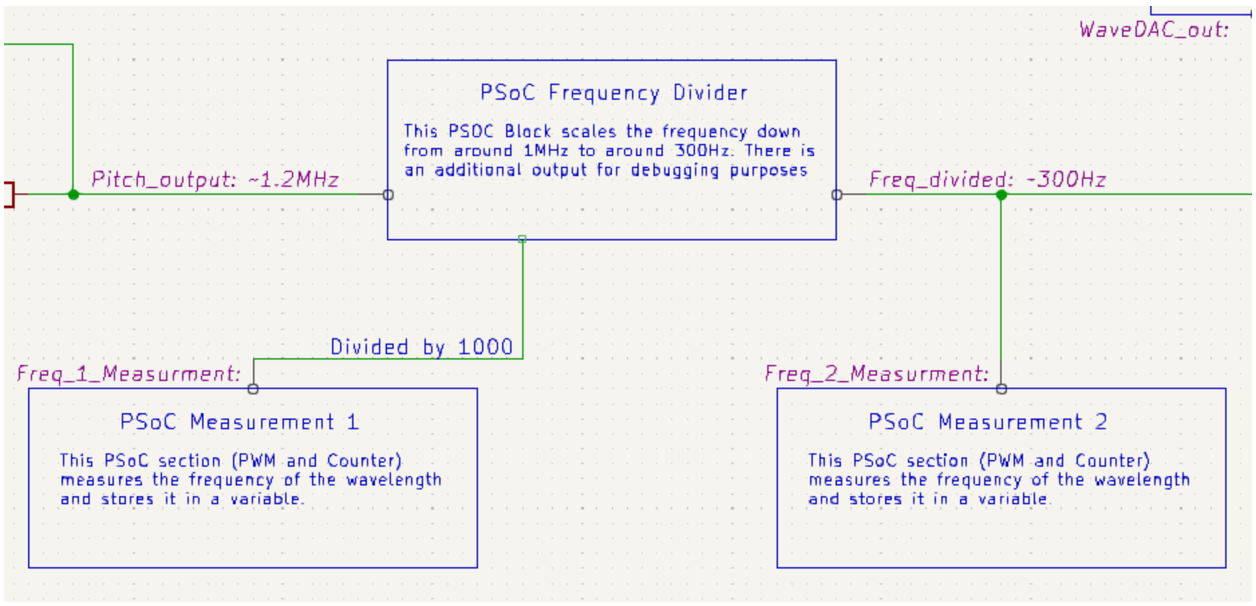

J = $\overline{A \cdot B}$
K = $\overline{C \cdot D}$
L = $\overline{E \cdot F}$
M = $\overline{G \cdot H}$
V_{DD}
V_{SS}
92CS-23880
FUNCTIONAL DIAGRAM

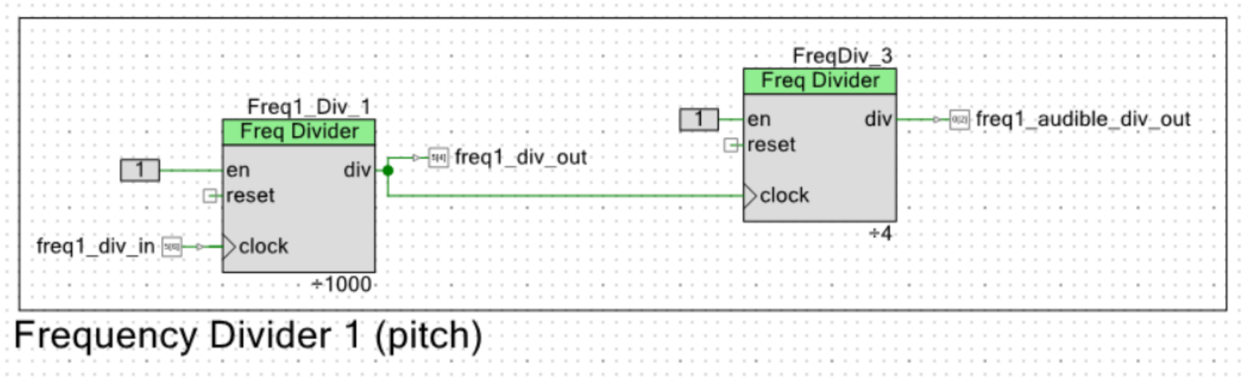


Measure
Frequency 1.292MHz
Period 773.7ns
Mean 2.40V
Pk-Pk 4.96V
Minimum -480mV
Maximum 4.48V
Pk-Pk 4.96V
Frequency 1.292MHz

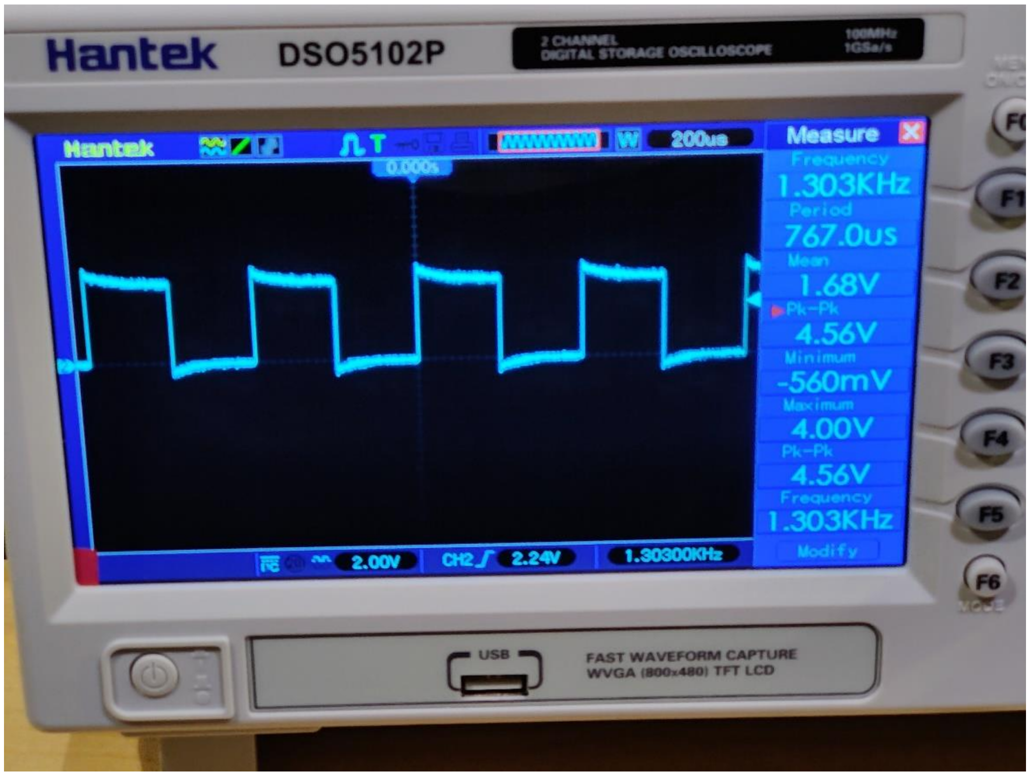
MODULE 2: PSOC FREQUENCY DIVIDER

After the varying frequency squarewave is produced by my "digital imitation of a colplitts oscillator", it must undergo frequency division by the PSoC and be measured so it can be ouput to the speaker and displayed on the TFT screen. The frequency division is done by two separate frequency dividers. The first divider divides the frequency by 1000, and the second divider divides by 4. The purose of this was for debugging so I could do math easier while reading the divided measurement. The frequency reader started becoming more innacurate when the frequency was too high and dividing by 1000 made the reading much better. There is no difference between using the two dividers and instead using one divider at value 4000 (aside from maybe some added signal loss).

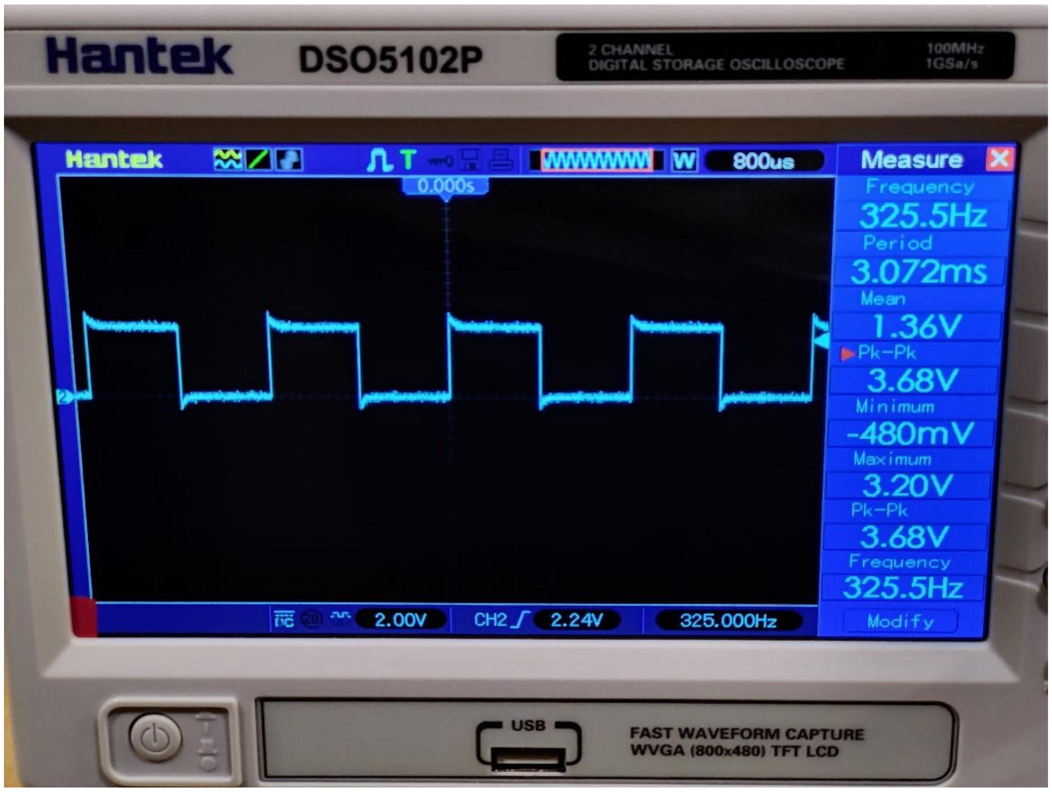

WaveDAC_out:
PSoC Frequency Divider
This PSOC Block scales the frequency down from around 1MHz to around 300Hz. There is an additional output for debugging purposes
Pitch_output: ~1.2MHz
Freq_divided: -300Hz
Divided by 1000
Freq_1_Measurment:
Freq_2_Measurment:
PSoC Measurement 1
This PSoC section (PWM and Counter) measures the frequency of the wavelength and stores it in a variable.
PSoC Measurement 2
This PSoC section (PWM and Counter) measures the frequency of the wavelength and stores it in a variable.



My original intention was to have the PSoC measure the frequency of the signal, then output a separate, unconnected ouput frequency, but I had some issues with the implementation of frequency measurement (explained later in the report) so I decided to connect them electrically to guarantee a finished product even if not completely ideal.


Hantek
DSO5102P
2 CHANNEL DIGITAL STORAGE OSCILLOSCOPE
100MHz 1GSa/s
FAST WAVEFORM CAPTURE WVGA (800x480) TFT LCD
USB

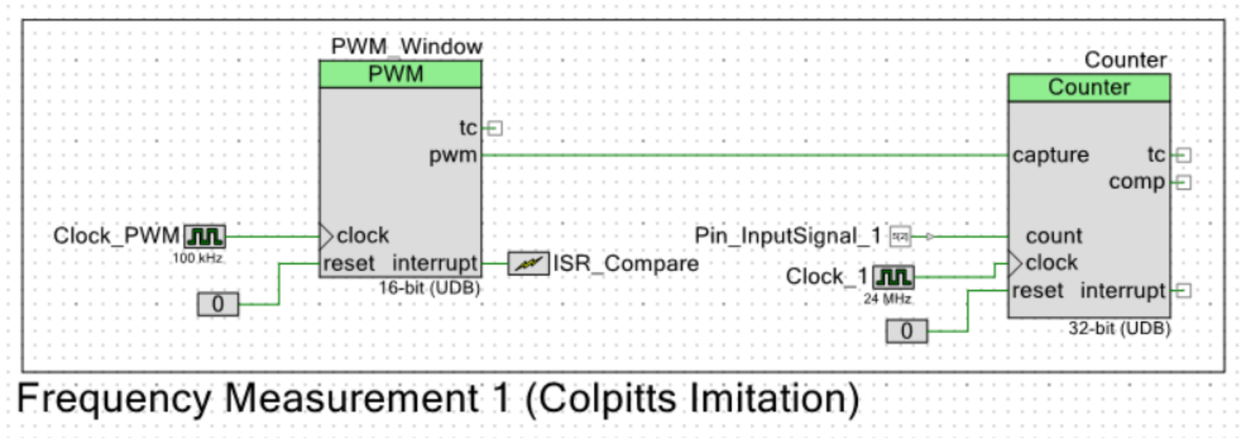
200us
0.000s
Measure
Frequency 1.303KHz
Period 767.0us
Mean 1.68V
Pk-Pk 4.56V
Minimum -560mV
Maximum 4.00V
Pk-Pk 4.56V
Frequency 1.303KHz
Modify
2.00V
CH2 2.24V
1.30300KHz

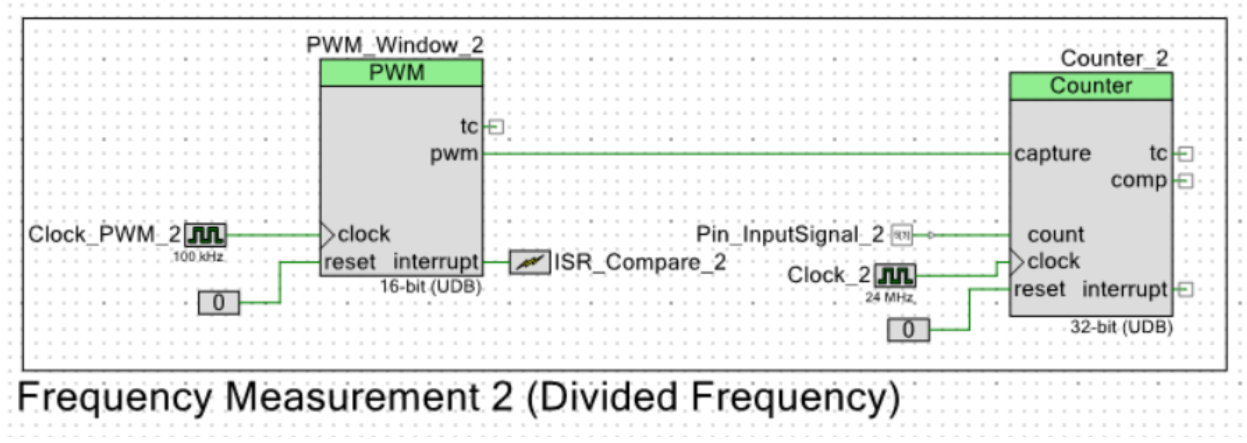

Hantek
DSO5102P
2 CHANNEL DIGITAL STORAGE OSCILLOSCOPE
100MHz 1GSa/s
FAST WAVEFORM CAPTURE WVGA (800x480) TFT LCD
USB

800us
0.000s
Measure
Frequency 325.5Hz
Period 3.072ms
Mean 1.36V
Pk-Pk 3.68V
Minimum -480mV
Maximum 3.20V
Pk-Pk 3.68V
Frequency 325.5Hz
Modify
2.00V
CH2 2.24V
325.000Hz

MODULE 3 AND 4: PSOC FREQUENCY MEASUREMENTS

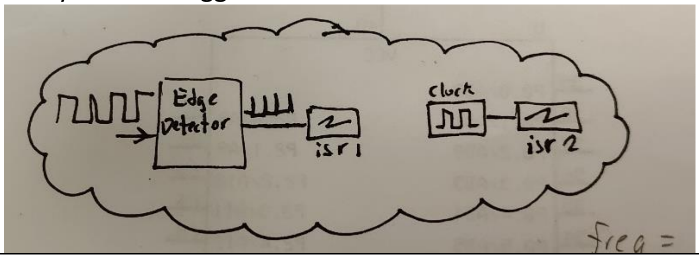
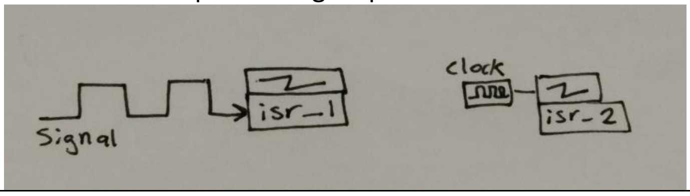
The frequency of the theremin is measured at two locations: after the squarewave generation, and after the frequency division to an audible frequency. These two frequencies are stored in variables in code to later be displayed on the TFT and LCD screen.





There are many ways to measure frequency using PSoC. I experiemented with many of them and they are described below:

Methods of Frequency Measurement (testing experimentally)

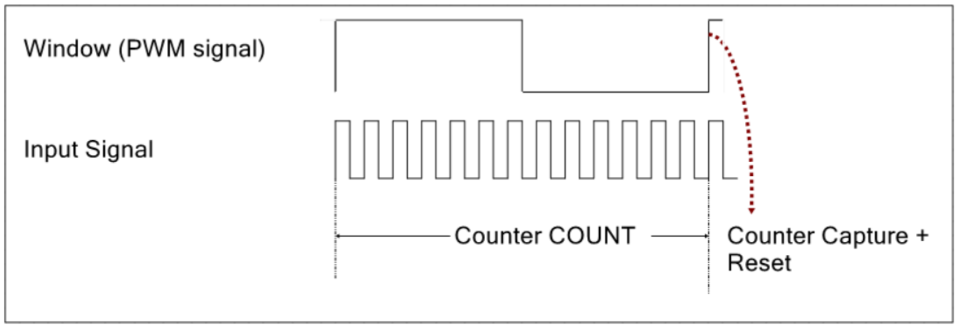
Method 1: PWM and Counter Blocks	This method is the most reliable and what I chose to use in my project for a few reasons. First of all, it relies on the hardware of the PSoC and doesn't use much code. This allows me to not take up so much CPU processing time and keep things as fast as possible.
Method 2: Signal to ISR, with additional ISR run by a fixed clock	This method uses an edge detector with the block input as the input frequency. The pulses that the edge detector produces are fed into an ISR that increments a counter every time its triggered. A separate ISR run by a slower clock then resets the counter and stores the counter value every time its triggered. 
Method 3: Signal directly to interrupt	This method is the most unreliable (trust me) and uses almost no PSoC hardware. The signal is fed directly from the input pin which triggers an interrupt when high. The interrupt implements the programmatic counter to increase and takes up a decently large amount of time in the CPU. There is also a second interrupt at a slower clock frequency triggering the counter to reset and store the value. This method is the same as method 2 but without the edge detector. It has many problems, one of them being the interrupt often gets triggered many times instead of once per one signal pulse. 

What's the purpose of the PWM block?

The implementation of the PWM block I took from a Cypress tutorial on frequency reading for the PSoC 5LP and it creates the slower frequency wavelength which triggers the counter to reset and capture its current value. This is better than using a clock and interrupt because it's simpler, faster, and more compact programmatically.

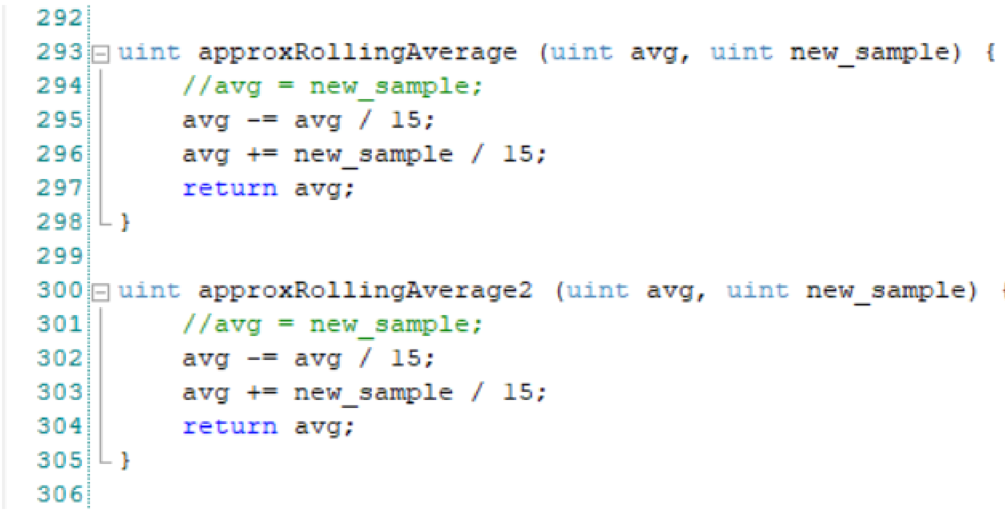
How exactly does the PWM and Counter system calculate frequency?

The basic concept for frequency reading is the same for all three methods of frequency reading shown above.



If you know how how many pulses of your input signal were received in a certain amount of time you can use the formula $Frequency = \frac{1}{T} = \frac{1}{\frac{\Delta t}{num\_pulses}} = \frac{num\_pulses}{\Delta t}$, where T is the period of input signal pulses, num_pulses is the number of pulses in the time Δt, and Δt is the amount of time. The longer your Δt is, the more pulses you will have in your time frame and will make your frequency reading more accurate.

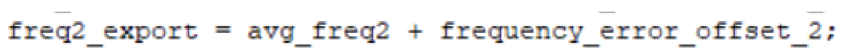
When implementing my frequency reading, my original intention was to use this frequency measurement to map to a range of lower frequency notes, but my frequency reading was very inconsistent, so I decided to use them simply as measurements of the predivided and divided frequencies. To make the frequency measurements cleaner, I implement a running average of the frequency measurements. This helps stabalize the reading at the cost of making changes to the frequency reading slightly slower.



```

uint approxRollingAverage (uint avg, uint new_sample) {
    //avg = new_sample;
    avg -= avg / 15;
    avg += new_sample / 15;
    return avg;
}

uint approxRollingAverage2 (uint avg, uint new_sample) {
    //avg = new_sample;
    avg -= avg / 15;
    avg += new_sample / 15;
    return avg;
}

```

There are several ways to calculate the rolling average. This method is an approximation of the most common method called the “exponentially weighted moving average.” The “15”s in the code indicate the weight of the function and make the new sample affect the average value either more or less.

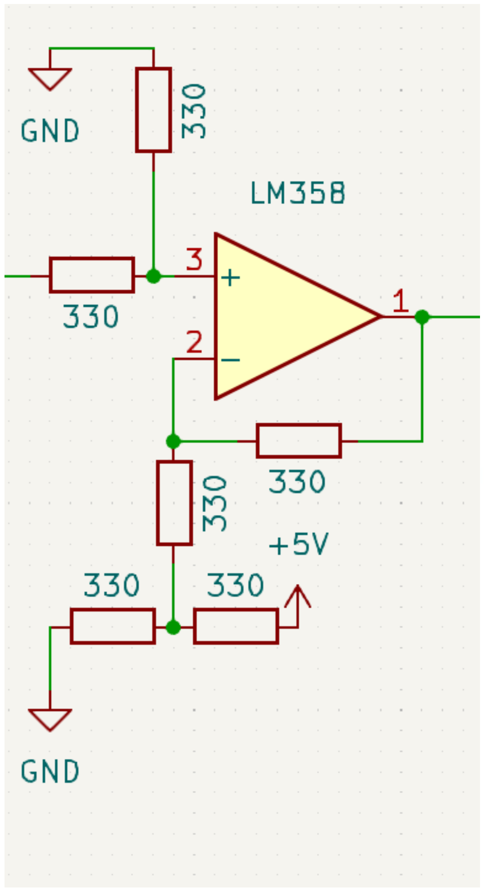
The final measurement of the frequencies are tuned and slightly adjusted with error offsets that are caused by running interrupts at such high speeds.



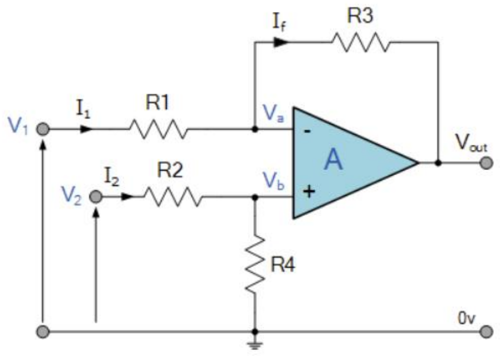
```
freq2_export = avg_freq2 + frequency_error_offset_2;
```

Module 4.5: Op-amp Offset to center wavelength

After the frequency division, there is an op-amp differential amplifier circuit that offsets the frequency by 2.5V to center it around 0V.



Differential Amplifier

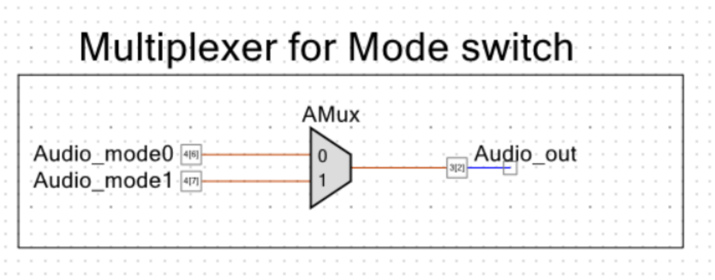


$$V_{OUT} = \frac{R_3}{R_1}\left(V_2 - V_1\right)$$

MODULE 5: SPEAKER (MODE 0 AND 1)

After the audible frequency is produced by dividing the oscillator circuit, it goes through a multiplexer, then through a non-inverting op amp with a range of -12V and 12V to increase the output signal.

Multiplexer

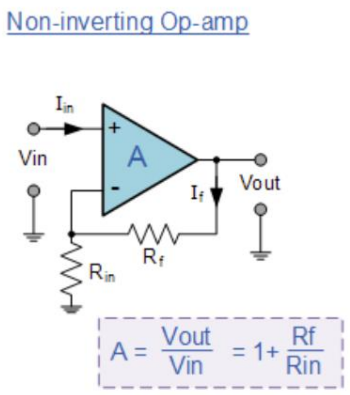


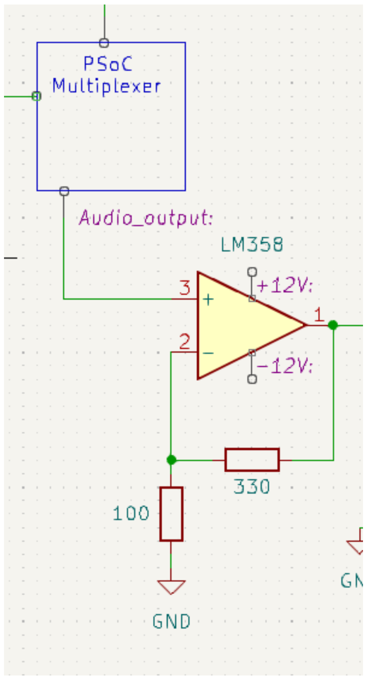
When in mode 0, the theremin runs in default "Theremin mode." This means the multiplexer will be set to channel 0 and allow the theremin audio to go through into the speaker. A variable in code called *theremin_state* determines the state of the theremin mode. This method was done with the intention of future flexibility so that this could be made into a more complex state machine with several modes.

When in mode 1, the theremin runs in "Set Frequency mode" which plays a steady fixed frequency at the note C (Frequency of 523.2Hz). The multiplexer changes the channel to channel 1.

Non-inverting Op-amp

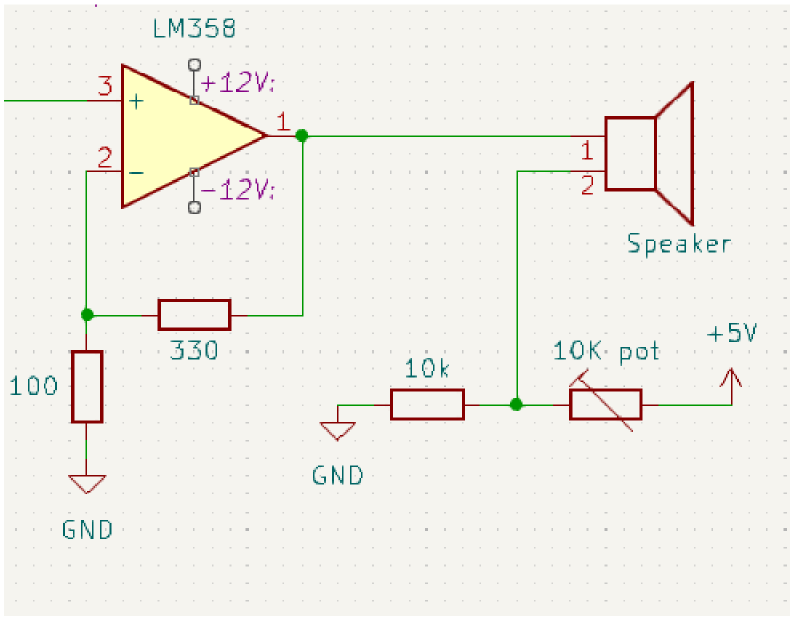
Regardless of whether the theremin is in mode 0 or 1, the audio_output goes through a non-inverting op-amp with a scale of 3 to amplify the audio to the speaker. It does this using -12V and 12V that are provided by the R31JP.



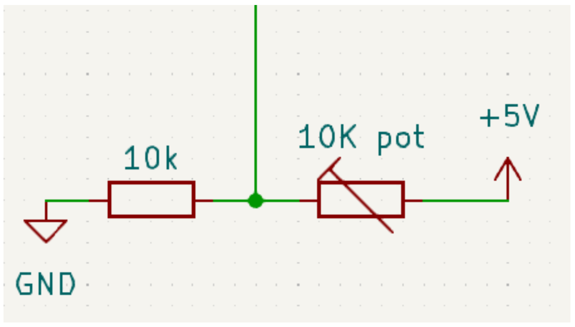


Speaker

After the audio_output has been multiplexed and amplified, it goes to the speaker.



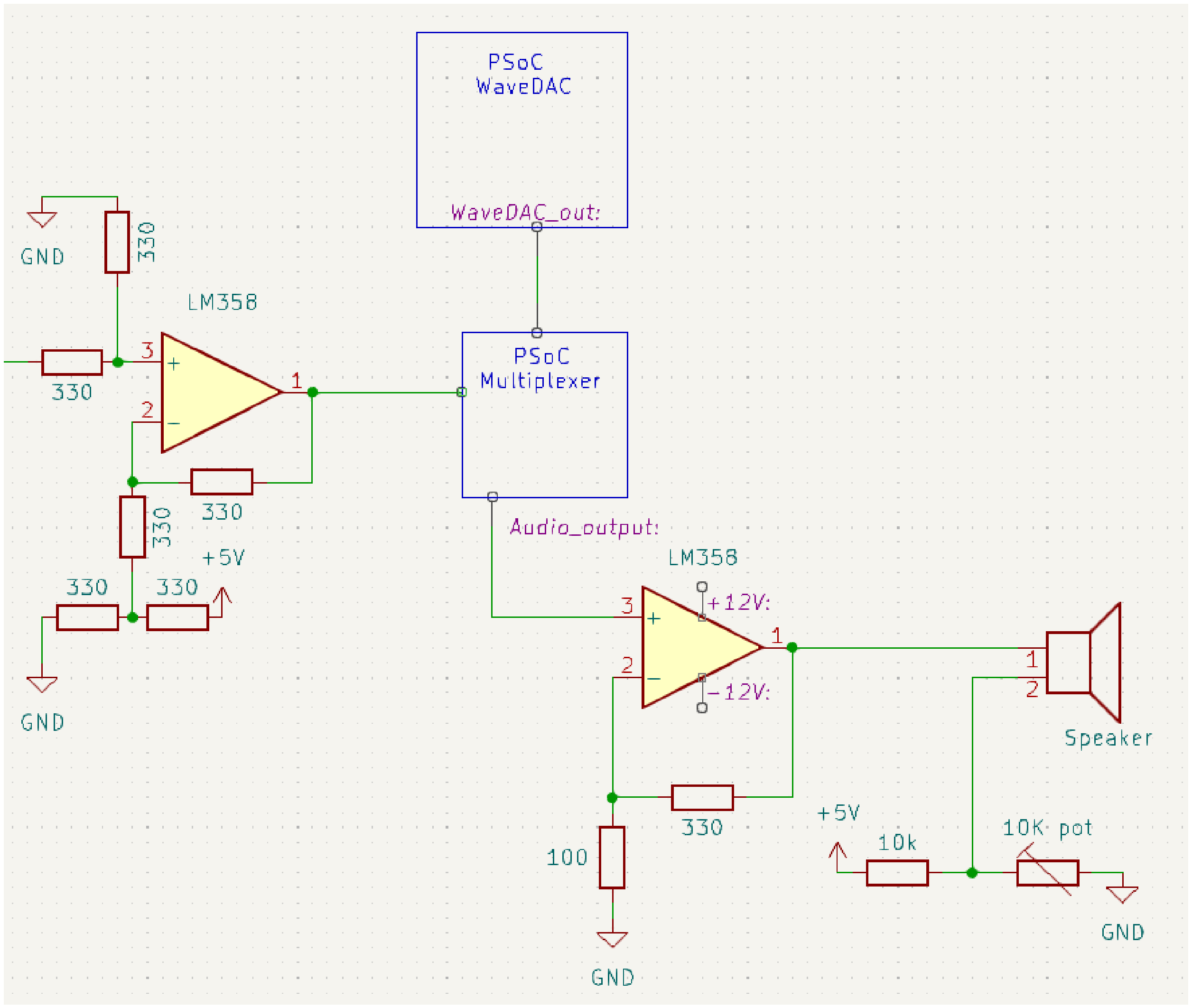
In addition to controlling pitch, I also want to be able to control the volume of my theremin. To do so, I don't connect the speaker to ground, and instead connect it to a voltage divider with a potentometer which allows me to control the voltage going to the speaker.

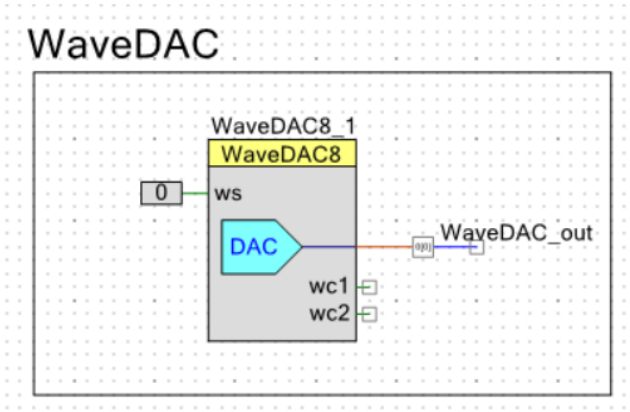


Using this method, I can varying the volume to both the mode 0 output and the mode 1 output using the same potentiometer.

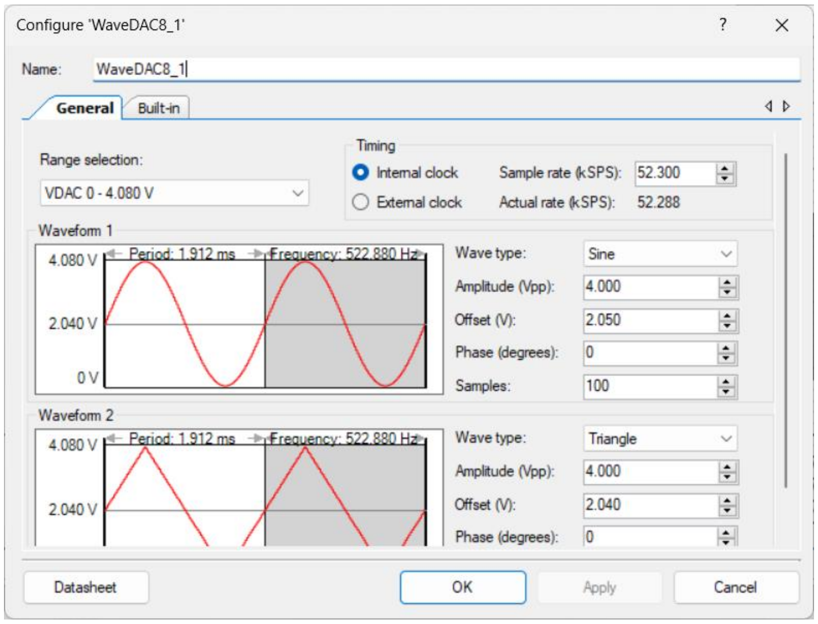
MODULE 6: MODE 1 WAVEFORM PRODUCTION

As mentioned previously, when in Mode 1, the theremin goes into “Set Frequency Mode” and plays a fixed “C” note. It does this using the WaveDAC block.





The waveDAC block produces a sinewave between 0-4.080V at 523Hz to make a crisp note of C5.


Configure 'WaveDAC8_1'

Name: WaveDAC8_1

Range selection: VDAC 0 - 4.080 V

Timing
Internal clock [✓]
External clock
Sample rate (kSPS): 52.300
Actual rate (kSPS): 52.288

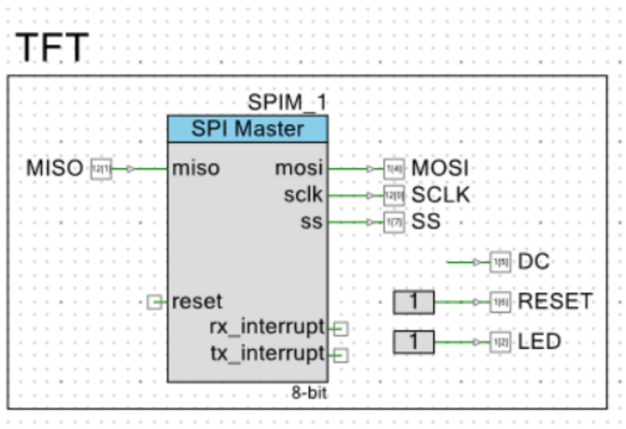
Waveform 1
Period: 1.912 ms
Frequency: 522.880 Hz
Wave type: Sine
Amplitude (Vpp): 4.000
Offset (V): 2.050
Phase (degrees): 0
Samples: 100

Waveform 2
Period: 1.912 ms
Frequency: 522.880 Hz
Wave type: Triangle
Amplitude (Vpp): 4.000
Offset (V): 2.040
Phase (degrees): 0

The original intention of mode 1 was to use buttons to change the set frequency to other perfectly tuned notes, but I couldn't find an easy way to change the frequency of the WaveDAC, so chose to keep it at C instead.

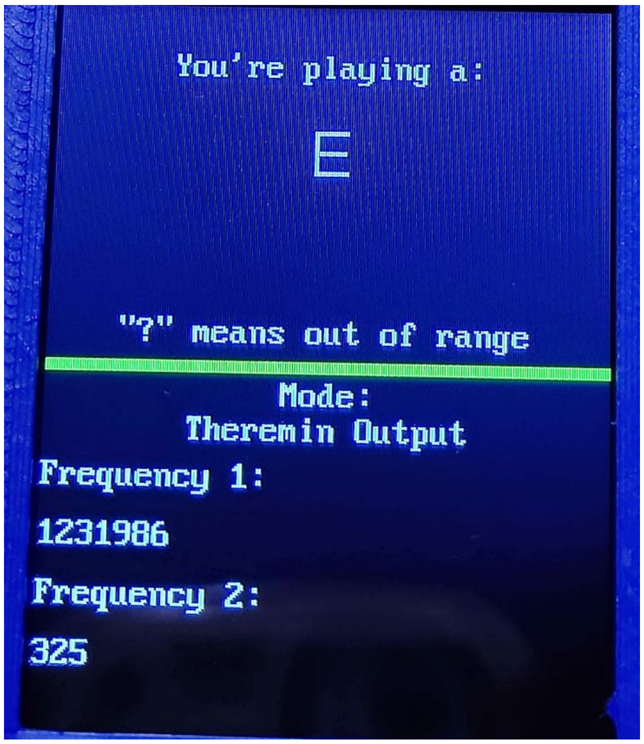
MODULE 7: OTHER PERIPHERALS (TFT, BUTTONS, AND LCD)

2.8" TFT Screen



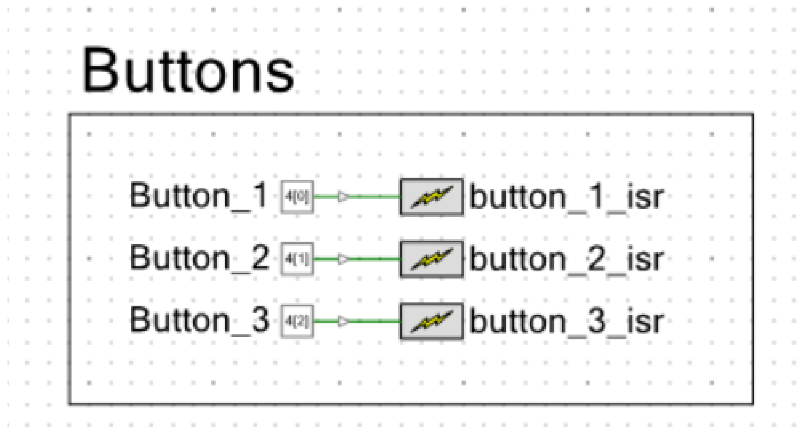
To display various stauses of the theremin, I use a TFT screen to display the frequency of the oscillator, frequency of the ouput to the speaker, the mode of the theremin, and the note being played by the theremin.

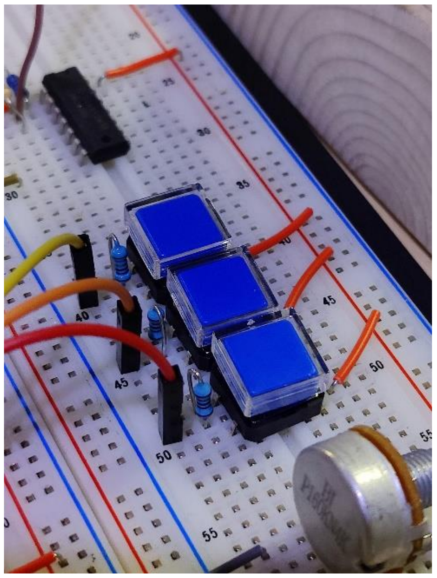
The TFT uses SPI (serial peripheral interface) to interface the screen, so I found some already made libraries that simplify the usage of the screen. Using EmWin for PSoC 5LP, I display these things all on one screen and update the values as the theremin runs. There is also a green line seperating the frequency readings and the note values for aesthetics :) .



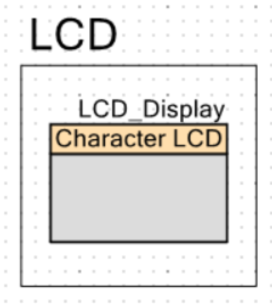
Buttons

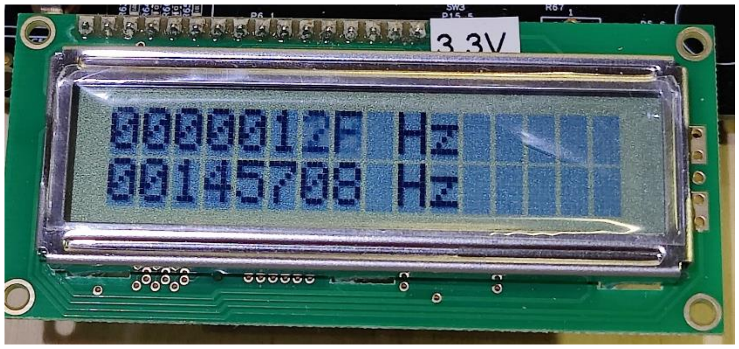
To switch from mode 0 to mode 1, button 1 must be pressed which goes high and triggers an interrupt that sets the mode to 1 and changes the multiplexer channel. When pressed, the mode on the TFT also changes to display “Set Frequency Mode” while freezing all other readings since they are not relevant.





LCD Screen

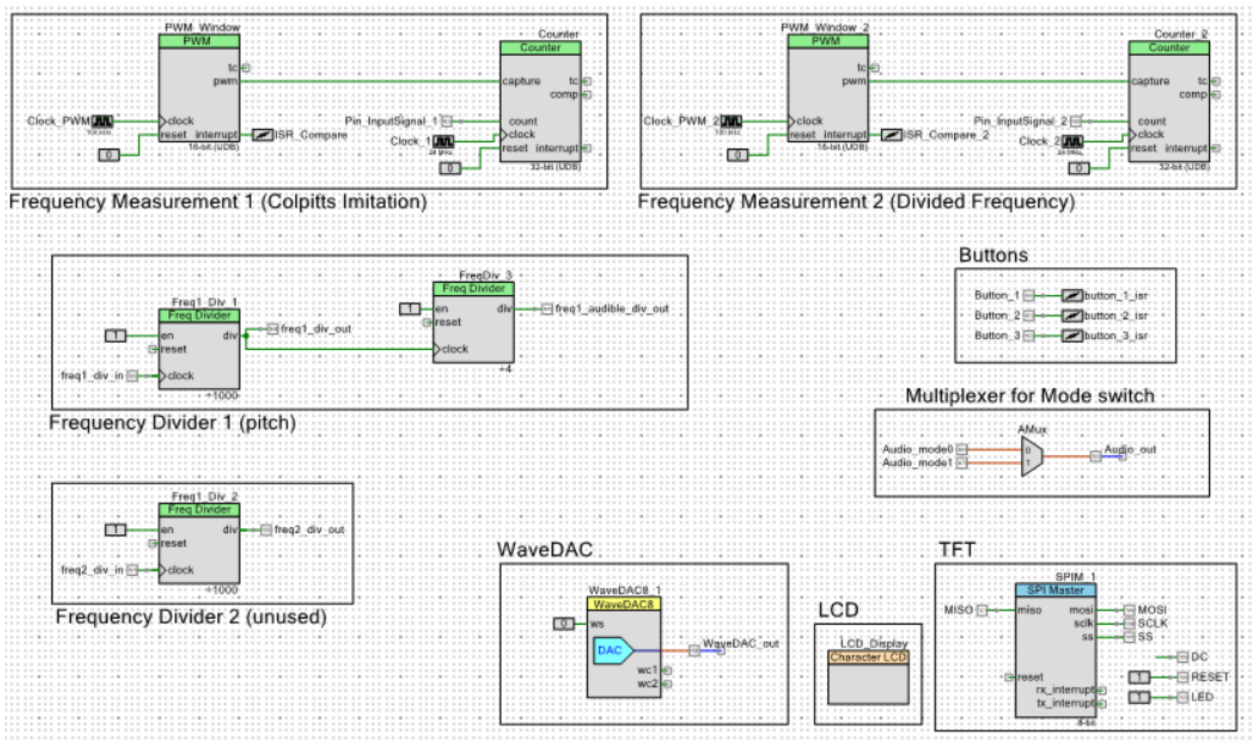


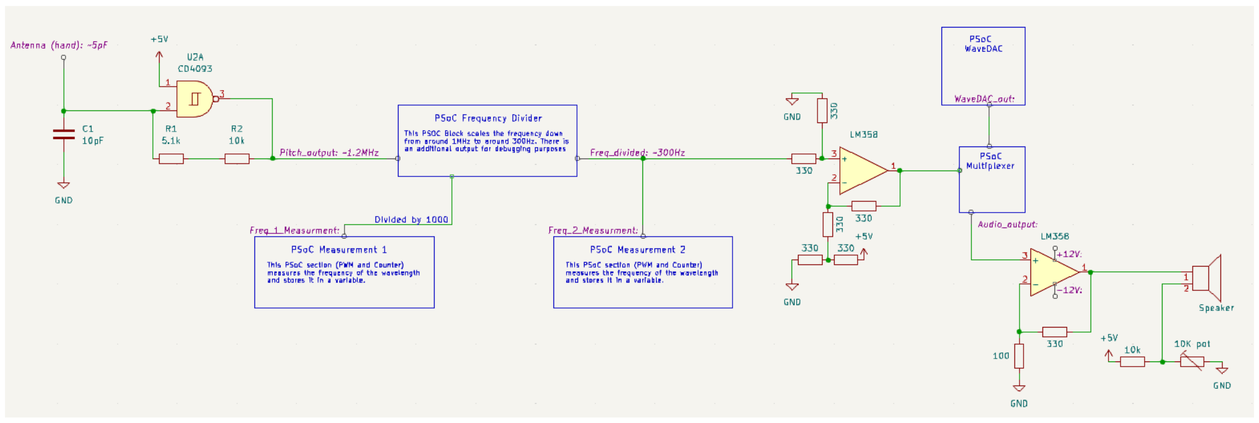


The LCD is not an important part of the final presentation since its main purpose was for debugging of the system. I personally found the LCD much easier to use for debugging than settting up a serial through tera term so I opted to use it instead.

While running in regular theremin mode, the LCD displays the frequency readings (raw and unfiltered) in Hex on the LCD screen. I guess if you like, you can look at that for fun.

SCHEMATIC APPENDIX





PSoC Frequency Divider

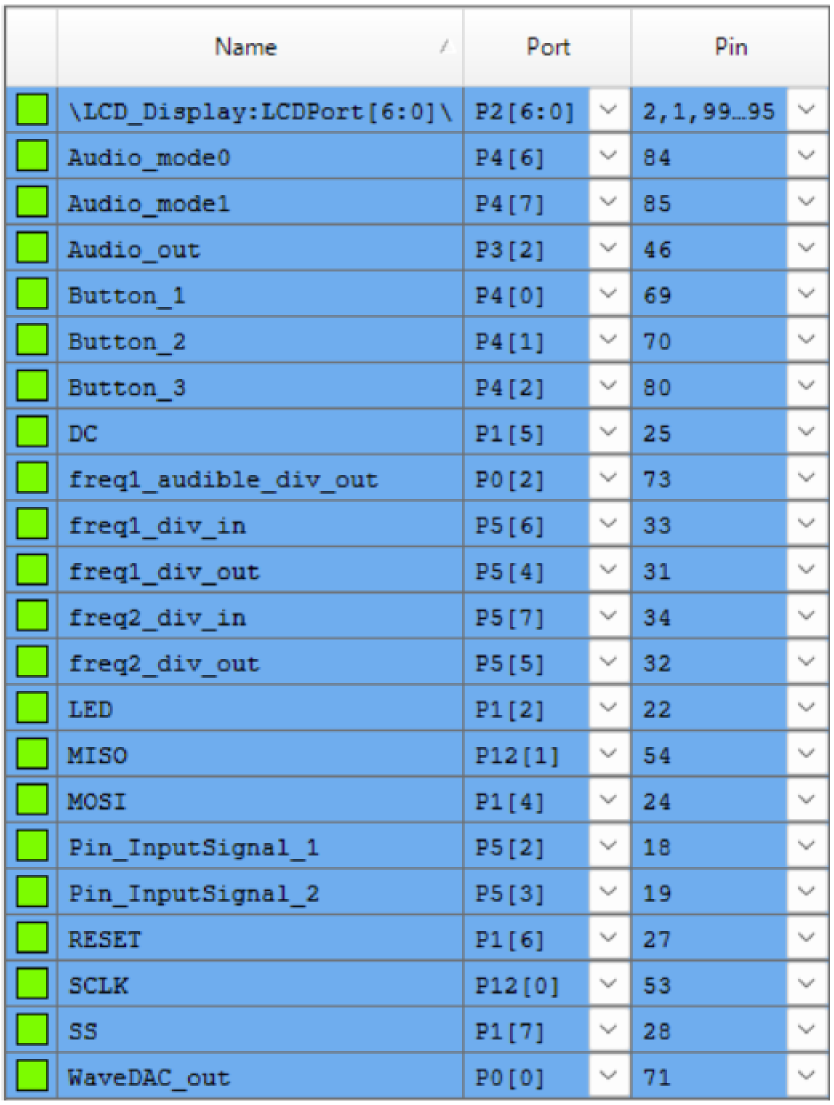
This PSOC Block scales the frequency down from around 1MHz to around 300Hz. There is an additional output for debugging purposes

PSoC Measurement 1

This PSoC section (PWM and Counter) measures the frequency of the wavelength and stores it in a variable.

PSoC Measurement 2

This PSoC section (PWM and Counter) measures the frequency of the wavelength and stores it in a variable.



	Name	Port	Pin
🟩	\LCD_Display:LCDPort[6:0]\	P2[6:0]	2,1,99…95
🟩	Audio_mode0	P4[6]	84
🟩	Audio_mode1	P4[7]	85
🟩	Audio_out	P3[2]	46
🟩	Button_1	P4[0]	69
🟩	Button_2	P4[1]	70
🟩	Button_3	P4[2]	80
🟩	DC	P1[5]	25
🟩	freq1_audible_div_out	P0[2]	73
🟩	freq1_div_in	P5[6]	33
🟩	freq1_div_out	P5[4]	31
🟩	freq2_div_in	P5[7]	34
🟩	freq2_div_out	P5[5]	32
🟩	LED	P1[2]	22
🟩	MISO	P12[1]	54
🟩	MOSI	P1[4]	24
🟩	Pin_InputSignal_1	P5[2]	18
🟩	Pin_InputSignal_2	P5[3]	19
🟩	RESET	P1[6]	27
🟩	SCLK	P12[0]	53
🟩	SS	P1[7]	28
🟩	WaveDAC_out	P0[0]	71

CODE APPENDIX (MAIN.C ON PSOC BB)

```
/* ========================================
 *
 * Copyright Miguel Talamantez, 2024
 * All Rights Reserved
 *
 * 6.115 Final Project: Theremin Project
 *   Spring 2024 - Professor S. Leeb
 *
 * ========================================
*/

//----------------------------------------
//          For debugging purposes
// Current modules integrated:
//
// -TFT (working)
// -Freq Reading 1 (working)
// -Freq Reading 2 (working)
// -LCD (working)
// -WaveDAC (working)
// -Freq Divider 1 (working)
// -Freq Divider 2 (unused)
// -Buttons (working)
// -Multiplexer (working)
//----------------------------------------


#include <project.h>
#include <stdio.h>
#include "GUI.h"
#include "tft.h"

void TFT_test(void);
void TFT_Startup(void);
void TFT_Display_Freq1(uint);
void TFT_Display_Freq2(uint);
void TFT_Note_C(void);
void TFT_Draw_HLine(void);
void DisplayNoteForFrequency(uint32_t);
void TFT_unstable_freq(void);
void TFT_clean_numbers(void);
uint approxRollingAverage(uint, uint);
uint approxRollingAverage2(uint, uint);
    uint32 theremin_state = 0;
        //State 0: Theremin Mode
        //State 1: Set Frequency Mode
uint16 set_note_wavedac = 0;
    // 0 is C(261Hz)
    // 1 is C#(277Hz)
    // 2 is D(293Hz)
    // 3 is D#(311Hz)
    // 4 is E(329Hz)
    // 5 is F(349Hz)
    // 6 is F#(369Hz)
    // 7 is G(391Hz)
    // 8 is G#(415Hz)
    // 9 is A(440Hz)
    // 10 is A#(466Hz)
    // 11 is B(493Hz)

//--------------Variable Init---------------
extern uint8 compare_occured;
extern uint8 compare_occured2;
#define PWM_FREQ 100 //kHz Clock freq of PWM_Window(1 and 2). If PWM clock is
changed, this MUST be updated.
#define NO_OF_MSEC 1000 //ms

//--------------Interrupt Init---------------
    CY_ISR_PROTO(button_1_isr);
    CY_ISR(button_1_isr){
    theremin_state = 1;
    AMux_Select(1);
    GUI_DispStringAt("Set Frequency Output", 40, 180);
    button_1_isr_ClearPending();
    }

    CY_ISR_PROTO(button_2_isr);
    CY_ISR(button_2_isr){
    set_note_wavedac = set_note_wavedac + 1;
    button_2_isr_ClearPending();
    }

int main()
{
//  Miscl Initializations + Variables
    CyGlobalIntEnable;                  // Enable global interrupts
    uint32 input_freq = 0;
    uint32 input_freq2 = 0;
    uint32 avg_freq1 = 0;
    uint32 avg_freq2 = 0;
      static uint16 PWM_windowPeriod = 0; //period of PWM_Window
      static uint32 counter_countVal; //stores the count value after capture
      static uint16 PWM_windowPeriod2 = 0; //period of PWM_Window
      static uint32 counter_countVal2; //stores the count value after capture
    uint32 frequency_error_offset_2 = 25; //Hz
    uint32 freq2_export = 0;


//  LCD Initializations
      LCD_Display_Start();
//  AMux Intializations
    AMux_Start();
    AMux_Select(0);
//  TFT Initializations
    SPIM_1_Start();
    TFT_Startup();
//  WaveDAC Initializations
    WaveDAC8_1_Start();
//  Frequency Reading Initializations
    ISR_Compare_Start();
    ISR_Compare_2_Start();
```

```
      PWM_Window_Start();
    PWM_Window_2_Start();
      Counter_Start();
    Counter_2_Start();
      Clock_PWM_Start();
    Clock_PWM_2_Start();
    PWM_windowPeriod = PWM_Window_ReadPeriod();
      PWM_windowPeriod = PWM_windowPeriod/PWM_FREQ;
    PWM_windowPeriod2 = PWM_Window_2_ReadPeriod();
      PWM_windowPeriod2 = PWM_windowPeriod2/PWM_FREQ;

//  Clear and prepare screen before start
    GUI_Clear();
    GUI_SetFont(&GUI_Font8x16); //GUI_FontD24x32
    GUI_DispStringAt("You're playing a:", 50, 20);
    GUI_DispStringAt("Mode: ", 100, 165);
    GUI_DispStringAt("                        ", 40, 180);
    GUI_DispStringAt("Theremin Output", 60, 180);
    TFT_Draw_HLine();
    TFT_unstable_freq();
    CyDelay(1000);

//  Interrupt Initializations
    button_1_isr_StartEx(button_1_isr);
    button_2_isr_StartEx(button_2_isr);
    AMux_Select(0);     //Set multiplexer to channel 0
    theremin_state = 0;     //Set theremin to mode 0


    //START LOOP
    for(;;) {

        if (theremin_state == 0){
            GUI_DispStringAt("                                ", 0, 180);
            GUI_DispStringAt("Theremin Output", 60, 180);
            if (compare_occured == 1) //If interrupt1 has occured: Read Freq
and display on LCD
            {
                  counter_countVal = Counter_ReadCapture(); //read the counter
value
                  input_freq = ((uint32)(NO_OF_MSEC *
(uint32)counter_countVal) / (uint32)PWM_windowPeriod); //calculate the freq
                  input_freq = input_freq*1000; //scale
                avg_freq1 = approxRollingAverage(avg_freq1, input_freq);
                LCD_Display_Position(1, 0);
                  LCD_Display_PrintInt16(HI16(input_freq));
                  LCD_Display_Position(1, 4);
                  LCD_Display_PrintInt16(LO16(input_freq));
                  LCD_Display_Position(1, 9);
                  LCD_Display_PrintString("Hz");
                TFT_Display_Freq1(avg_freq1);
                  compare_occured = 0; //Clear interrupt flag
            }

            if (compare_occured2 == 1) //If interrupt2 has occured: Read Freq2
and display on LCD
            {
                    counter_countVal2 = Counter_2_ReadCapture();
                    input_freq2 = ((uint32)(NO_OF_MSEC *
(uint32)counter_countVal2) / (uint32)PWM_windowPeriod2);
                  avg_freq2 = approxRollingAverage2(avg_freq2, input_freq2);
                  freq2_export = avg_freq2 + frequency_error_offset_2;
                  LCD_Display_Position(0, 0);
                    LCD_Display_PrintInt16(HI16(input_freq2));
                    LCD_Display_Position(0, 4);
                    LCD_Display_PrintInt16(LO16(input_freq2));
                    LCD_Display_Position(0, 9);
                    LCD_Display_PrintString("Hz");
                  TFT_Display_Freq2(freq2_export);
                  DisplayNoteForFrequency(freq2_export); //Display note being
played
                    compare_occured2 = 0; //Clear interrupt flag
             }
         }
         if (theremin_state == 1){
             GUI_DispStringAt("                       ", 0, 180);
             GUI_DispStringAt("Set Frequency Output", 40, 180);
             GUI_SetFont(&GUI_Font32_ASCII);
             GUI_DispStringAt("        ", 110, 50);
             GUI_SetFont(&GUI_Font8x16);
             GUI_DispStringAt("Perfect C :)", 70, 50);

         }
     }
}

//--------------Subroutines---------------

void TFT_test()
{
    GUI_Init();                             // initilize graphics library
    GUI_Clear();
    GUI_SetFont(&GUI_Font8x16);
    GUI_DispStringAt("Hi, my name is Theremin.", 0, 0);
}

void TFT_Startup() {
    GUI_Init();                             // initilize graphics library
    GUI_Clear();
    GUI_SetFont(&GUI_Font8x16);
    GUI_DispStringAt("6.115 Final Project", 50, 50);
    GUI_DispStringAt("Miguel Talamantez", 60, 70);
    GUI_DispStringAt("March, 2024", 80, 90);
    GUI_DispStringAt("Theremin Number 29", 50, 150);
}

void TFT_Display_Freq1(uint frequency)
{
    GUI_DispStringAt("Frequency 1:", 0, 200);
    GUI_DispStringAt("                            ", 0, 225);
    char tft_inputfreq1[100];
    sprintf(tft_inputfreq1, "%d", frequency);
    GUI_DispStringAt(tft_inputfreq1, 0, 225);
}
```

```
void TFT_Display_Freq2(uint frequency)
{
    GUI_DispStringAt("Frequency 2:", 0, 250);
    GUI_DispStringAt("                                   ", 0, 275);
    char tft_inputfreq2[100];
    sprintf(tft_inputfreq2, "%d", frequency);
    GUI_DispStringAt(tft_inputfreq2, 0, 275);
}

void TFT_Note_C(void){
    // Draw 'C'
    GUI_SetFont(&GUI_Font32_ASCII); // Set font size
    GUI_DispCharAt('C', 110, 50); // Display 'C' at coordinates (70, 90)
    GUI_SetFont(&GUI_Font8x16);
}

void TFT_Draw_HLine(void){
    // Set the line color and thickness
    GUI_SetColor(GUI_GREEN); // Set line color to green
    GUI_SetPenSize(5); // Set line thickness to 5 pixel
    int startY = GUI_GetScreenSizeY() / 2; // Start Y-coordinate (halfway
down the screen)
    int endY = startY; // End Y-coordinate (same as start Y-coordinate)
    int startX = 0; // Start X-coordinate (left edge of the screen)
    int endX = GUI_GetScreenSizeX(); // End X-coordinate (right edge of the
screen
    GUI_DrawLine(startX, startY, endX, endY);
    GUI_SetColor(GUI_WHITE); // Set line color to green
}

void TFT_unstable_freq(void){
    //GUI_DispStringAt("\"u\" means unstable reading", 20, 120);
    GUI_DispStringAt("\"?\" means out of range ",30, 138);
}

void TFT_clean_numbers(void){
    GUI_SetFont(&GUI_Font32_ASCII);
    GUI_DispStringAt("       ", 110, 50);
    GUI_SetFont(&GUI_Font8x16);
    GUI_DispStringAt("                                   ", 0, 275);
    GUI_DispStringAt("                                   ", 0, 225);
}

void DisplayNoteForFrequency(uint32_t frequency) {
    char note;
    if (frequency >= 150 && frequency < 170) {
        note = 'E';
    } else if (frequency >= 170 && frequency < 190) {
        note = 'F';
    } else if (frequency >= 190 && frequency < 210) {
        note = 'G';
    } else if (frequency >= 210 && frequency < 230) {
        note = 'A';
    } else if (frequency >= 230 && frequency < 260) {
        note = 'B';
    } else if (frequency >= 260 && frequency < 290) {
        note = 'C';
    } else if (frequency >= 290 && frequency < 310) {
        note = 'D';
    } else if (frequency >= 310 && frequency < 330) {
        note = 'E';
    }
    //else if (frequency >= 330) {
        //note = 'u';
    //}
    else {
        note = '?';
    }
    GUI_SetFont(&GUI_Font32_ASCII);
    GUI_DispStringAt("      ", 110, 50);
    GUI_DispCharAt(note, 110, 50);
    GUI_SetFont(&GUI_Font8x16);
}


uint approxRollingAverage (uint avg, uint new_sample) {
    //avg = new_sample;
    avg -= avg / 15;
    avg += new_sample / 15;
    return avg;
}

uint approxRollingAverage2 (uint avg, uint new_sample) {
    //avg = new_sample;
    avg -= avg / 15;
    avg += new_sample / 15;
    return avg;
}
```