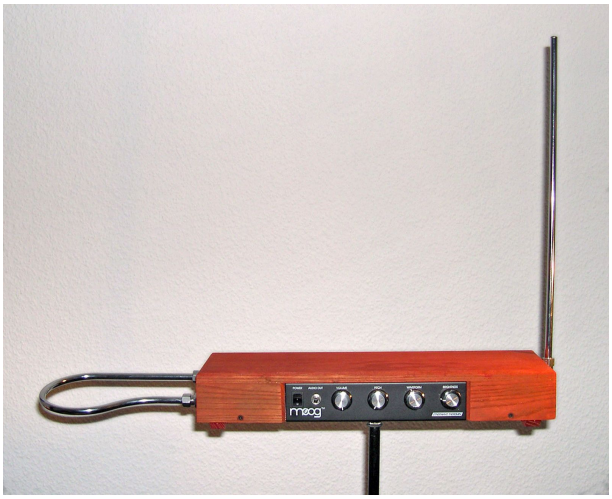
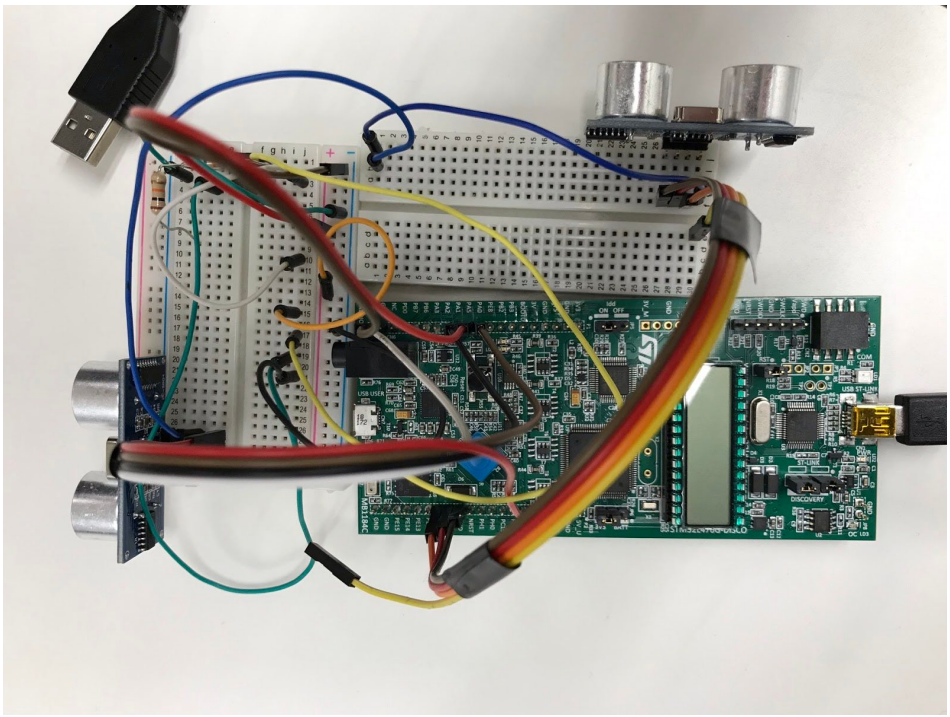
Final Project Report - Digital Theremin

Nik Jensen
A02250195
Orion Watson
A02346146
McClain Jorgensen
A02270953





Introduction

The Digital Theremin is a modification of the electronic musical instrument called a Theremin. The Theremin is controlled without physical touch by the performer and has two main antennas that sense the relative position of the user's hands. The original Theremin allows the user to control the soundwave with one antenna and the amplitude with the other. The Digital Theremin differs from the regular Theremin, as our device has been translated to a digital design, is cheaper, smaller, and more portable. Our device uses an electronic distance sensor to determine specific notes to be outputted through a speaker, as well as another distance sensor to adjust the amplitude of our soundwave. This device uses the STM32L476G Discovery Board, two HC-SR04 Ultrasonic Distance Sensors, and a speaker desired by the user.

Scope

This document describes the design and verification of a Digital Theremin. This device will give an overview of how the device functions, the details of the device's operation, the testing/results of this device, and challenges during design. This document will not cover the specifications, budget, and management plan of the Digital Theremin.

This device is a musical instrument that can manipulate note frequencies without physical contact of the device. The Digital Theremin uses two sensors, one to change the note of the instrument, and another sensor to adjust the amplitude of the output waveform.

Design Overview

Requirements

The Digital Theremin is required to output desired frequencies to a speaker, given the input from the user. Two HC-SR04 Ultrasonic Distance Sensors and a STM32L476G Discovery Board are required for the Digital Theremin design. This device will require the ability to locate the user's hand or held object to modulate a desired soundwave frequency and amplitude based on the distances from the device. Lastly, this device is required to output to a speaker using a Digital-to-Analog Converter (DAC).

Dependencies

The dependencies of the Digital Theremin is the user's input, and a power supply. In order for the device to be used properly, an input is required by the user. This input is either created by the user's hand, or preferrably, a flat object to place in front of the distance sensors in order to determine amplitude and note frequency. For a power supply, the ultrasonic distance sensor

requires a 5V power supply to each device. The device also requires an external 5V power supply to power our microcontroller. This external power supply can be implemented through the USB ST-LINK connection or the External 5V_I connection. This design cannot be powered through the 3V battery insert as the design requires a 5V supply to the distance sensors.

Theory of operation

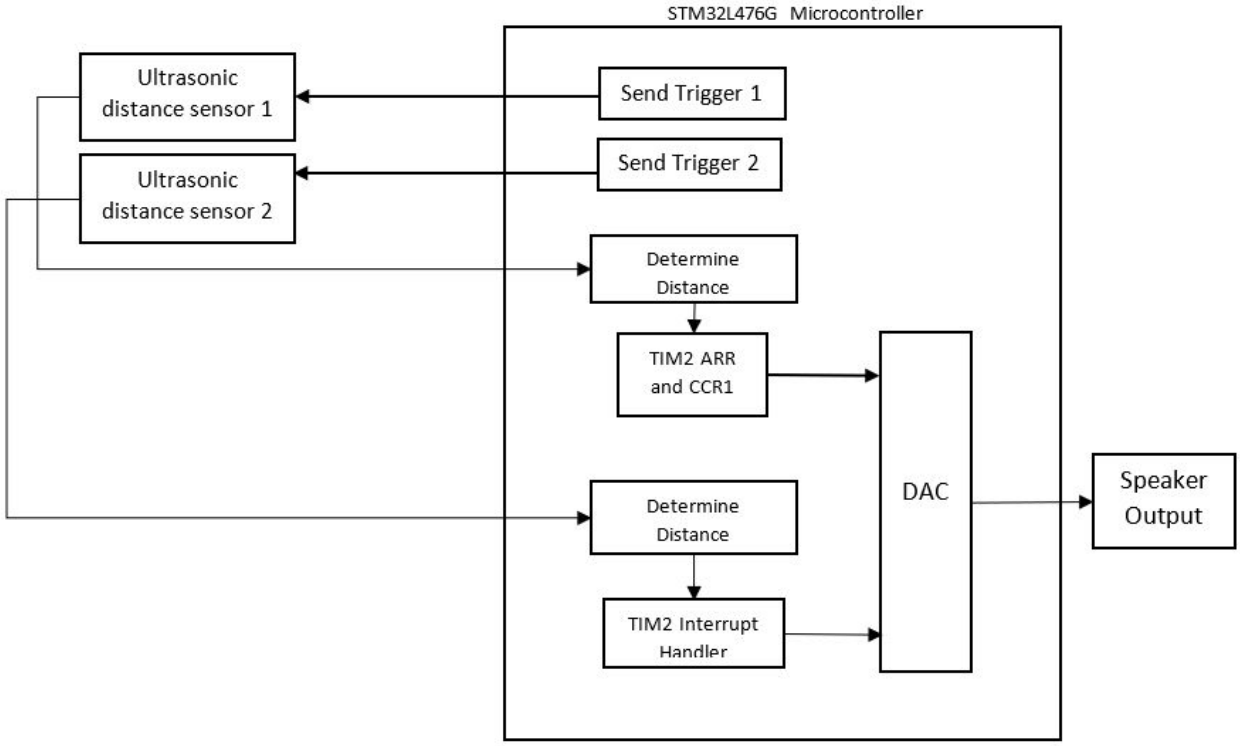


Figure: Function Block Diagram

Using the provided figure above, this section will describe each functional block of the design and how they interact with each other and the device. In terms of operation, the device begins by setting two output pins to high for 10 microseconds. The connected pins of the ultrasonic distance sensors receive these inputs and emit a frequency that is to be echoed back by an object. After an implemented 20 microsecond delay, the echoed frequency is then received by the microcontroller in the Determine Distance function block for both sensors.

The Determine Distance block then loops while the receiving echo is set as high for the input pin. While looping, the function determines how long the pin was high and then converts the time to a distance in centimeters. To determine this distance, the function uses the equation below.

$$Distance\ (cm)\ =\ (Time\ (in\ microseconds)\ *\ Speed\ of\ Sound)\ /\ 2$$

This distance is then used to determine what frequency is to be played by the speaker for sensor 1 or used to adjust the amplitude of our frequency in sensor 2. For our function block, TIM2 ARR and CCR1, from sensor 1, the Auto Reload Register (ARR) and Compare and Capture Register

1 (CCR1) values are adjusted to be used in our DAC soundwave, as these registers determine the frequency of our desired sound. For the function block from sensor 2, TIM2 Interrupt Handler, our determined distance reduces the amplitude of our sound. This is done by dividing the value's found from a created sine wave by a value determined from sensor 2.

Lastly, using the TIM2 interrupt, this interrupt is looped consistently while stepping through values of an equated sine wave. The ARR, CCR1, and amplitude are adjusted by our sensors, manipulating the frequency at which our TIM2 steps, thus resulting in a frequency change in our speaker output. The output of our DAC is simultaneously sent to our speaker to create a consistent waveform for our sound.

Design Alternatives

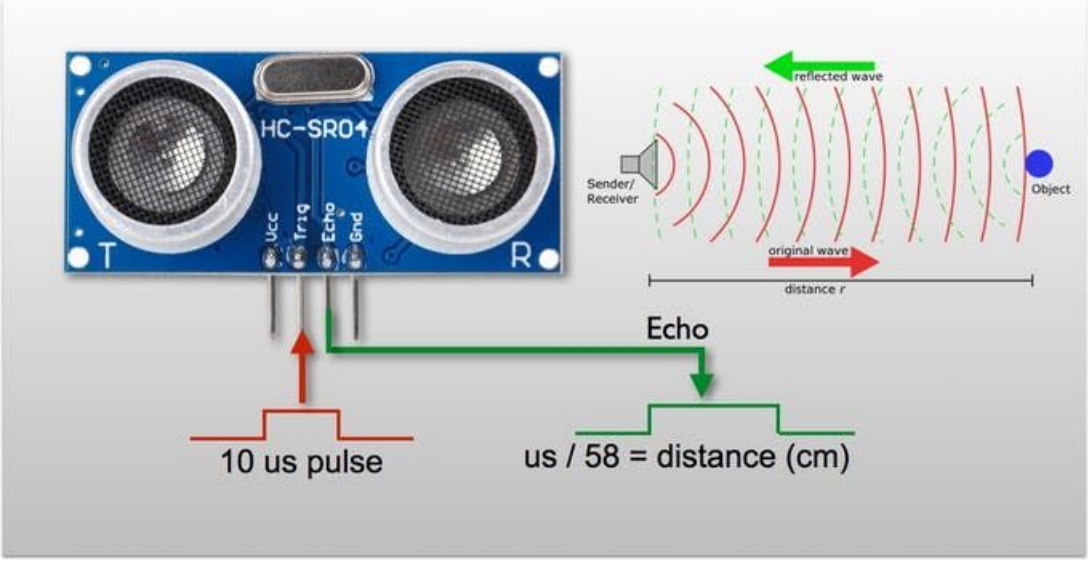
Other approaches to this design involved using several distance sensors to adjust octaves, timbre, and other needed sound outputs. This approach was not taken as this was deemed too difficult due to our experience levels, our time frame, a shortage of connector wires, and the addition of potential issues involving frequency interference between the sensors (each sensor already has to be activated on its own, not at the same time as the others, to avoid picking up eachother's ultrasonic chirps). We originally planned on a continuous spectrum of pitch as that was most similar to an analog theremin but decided on discrete values so that the theremin would always be in tune and would be easier to play. We originally built the theremin with the piezo buzzer but then upgraded to the earbud and speaker for better volume and sound quality.

Another alternate design consideration was about the sound system, instead of driving the speaker directly we could use the audio DAC (designated U13 in the microcontroller user manual, the actual part number is CS43L22) to provide the signal to the audio jack which would allow us to play the audio on powered speakers. This audio DAC could be accessed via the Serial Audio Interface (SAI) or through I^2C protocols documented in the data sheet for the CS43L22. However, due to time constraints we decided that what we had was good enough, but if this design was to be altered in order to boost its potential volume output using this method would be a straightforward way of accomplishing that.

Design Details

Distance Sensors

This device requires two distance sensors as it is the simplest way to implement the touchless operation of this device which is the nature of the instrument. The distance sensors we used in this project are ultrasonic and require a 10 microsecond pulse signal to trigger a series of 8 ultrasonic pulses at 40kHz and the echo signal will turn on afterward and stay on for a time that is proportional to the distance it traveled, 5 volts is supplied to Vcc.



HC-SR04 signal diagram

In order to send the trigger signal we simply have to turn on the right pin for 10 microseconds. In this case our system clock is running at 16MHz and the first sensor is connected to pin PA0.

```
void sendtrigger1() {
        //needs to have PA0 set as high for 10us
        //10us has a period of 100,000Hz
    //so our delay = 16MHz/100kHz = 160
        for(int i = 0; i <= 160; i++) {
        GPIOA->ODR |= 0x00000001;
        }
        GPIOA -> ODR &= 0xFFFFFFFE;
}
```

We wrote a delay function and determined that 400 was a suitable number to separate the trigger from the reply. These three functions are in **sensor.c**.

```
void delay(__IO uint32_t nCount){
        while(nCount--);
}
```

```
float receivetrigger1(){
        //while receiving, count
        //once done receiving, calculate distance from time the input was high
        float count = 0;
        float time = 0;

        //GPIOE->ODR |= 0x00000400; //For testing on scope

        while((GPIOA->IDR & 0x00000002) != 0){
        count += 1;
        if(count >= 20000){ //if count counts for too long, break
                break;
        }
```

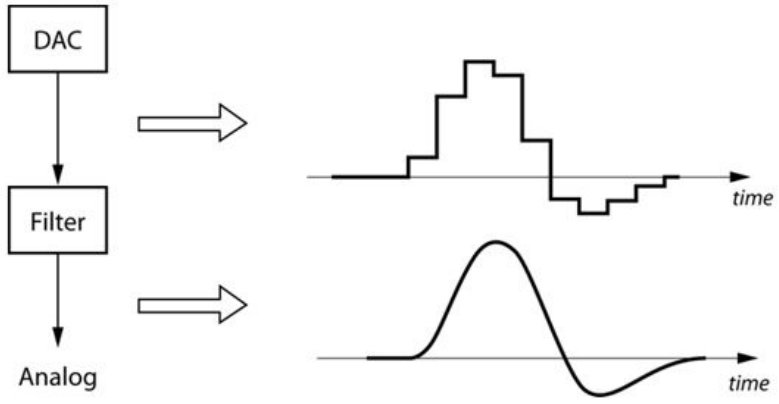
```
        }

        //GPIOE -> ODR &= 0xFFFFFBFF;//For testing on scope

        time = count * (6.25 * (10e-8));

        return time;
}
```

In **main.c**, we send the trigger signal, wait for the sensor and receive the echo signal, this is then repeated for the second sensor. After that we translate the **time** variable into a note number for the first sensor and a volume (amplitude) divisor, this is done using switch-case statements to convert a range of values into discrete values (hence a digital system). The note number also uses a switch-case statement in order to change the ARR and CCR of TIM2, thus adjusting the frequency of the final output, a digitized sine wave.

```
sendtrigger1();          //outputs 10us signal from sensor1
delay(400);              //delay around 10us (number determined on scope)
time = receivetrigger1(); //receives signal and translates into how long it took the signal to come back
```

Digital-to-Analog Converter (DAC)

The DAC is a system that converts a digital signal to an analog signal. The Digital Theremin depends upon the use of the DAC to output a soundwave as our speaker will require an analog frequency. For the implementation of our design, the DAC requires the use of an interrupt. An interrupt is a signal to the microcontroller that is emitted by hardware to let the microcontroller know that the interrupt handler needs to be used. The interrupt is important for our device as we need it to drive our analog output. For greater detail, the DAC system requires the use of a TIM interrupt (or timer interrupt), hence the use of TIM2 in our **DAC.c** programming file. While the DAC system does create an analog output, it does not create a real analog waveform, but a digital signal that has been manipulated to act similar to an analog waveform. This problem can be further improved with the implementation of a filter as shown below.

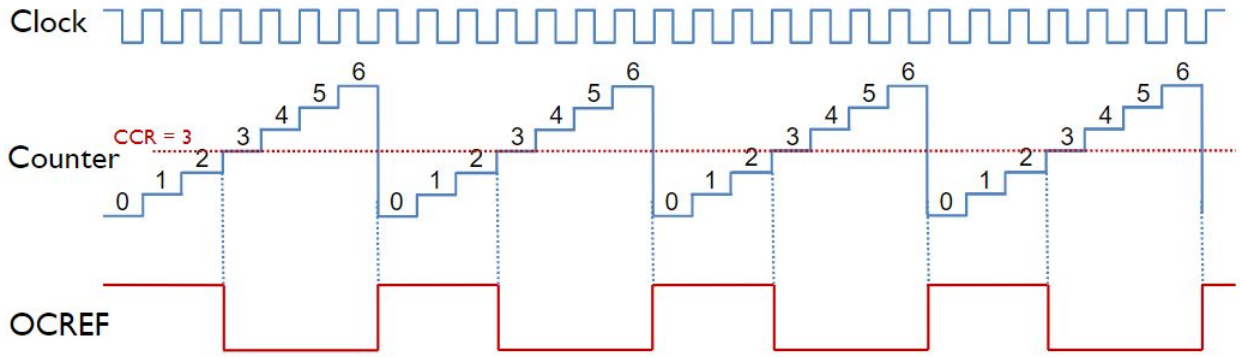


DAC imperfections and possible solutions

Referring to our code, **DAC.c**, we need to implement multiple functions. We started our functions with the initialization of our DAC's channel. This "channel" is where our analog signal can be outputted from. The initialization, **DAC_Channel2_Init**, consists mainly of enabling clocks, triggers, and channels while also calling the function **Create_Sine_Table** to create a simple sine wave for use later. This function can be seen below Next was the initialization of our TIM2 function, **TIM2_Init**, and creating our TIM2 Handler function, **TIM2_IRQHandler**. TIM2_Init consisted of clearing and setting bits for priority, the use of handling the interrupt, setting our PSC, ARR, and CCRx bits, and enabling clocks. **TIM2_IRQHandler** involved searching our sine table created earlier and setting those values into our DAC registers for output. Our DAC registers would tell our device how high to step when making the sine wave. All other functions can be found under DAC.c in appendices.

```
void Create_Sine_Table(void)
{
    float sf;

    // for 12-bit [0, 4095(0xFFF)];
    for (int i = 0; i <= 90; i++){
        sf = sin(3.14159 * i /180);
        sine_table[i] = (1 + sf) * 2048;
        if(sine_table[i] == 0x1000){
            sine_table[i] = 0xFFF;
        }
    }
    return;
}
```

As mentioned previously, the **TIM2_Init** function sets our Prescaler (PSC), Auto-Reload Register (ARR), and Compare and Capture Register (CCRx), these are used to manipulate the frequency, or for our device, the notes outputted. The PSC is of less importance for our design as it does not need to be manipulated to change notes. ARR is the value that our timer counts to before setting back to zero and pulsing, while the CCRx affects the duty ratio of our timer. By calculations, the ARR and CCRx can be changed to create desired musical notes. An example can be seen below.



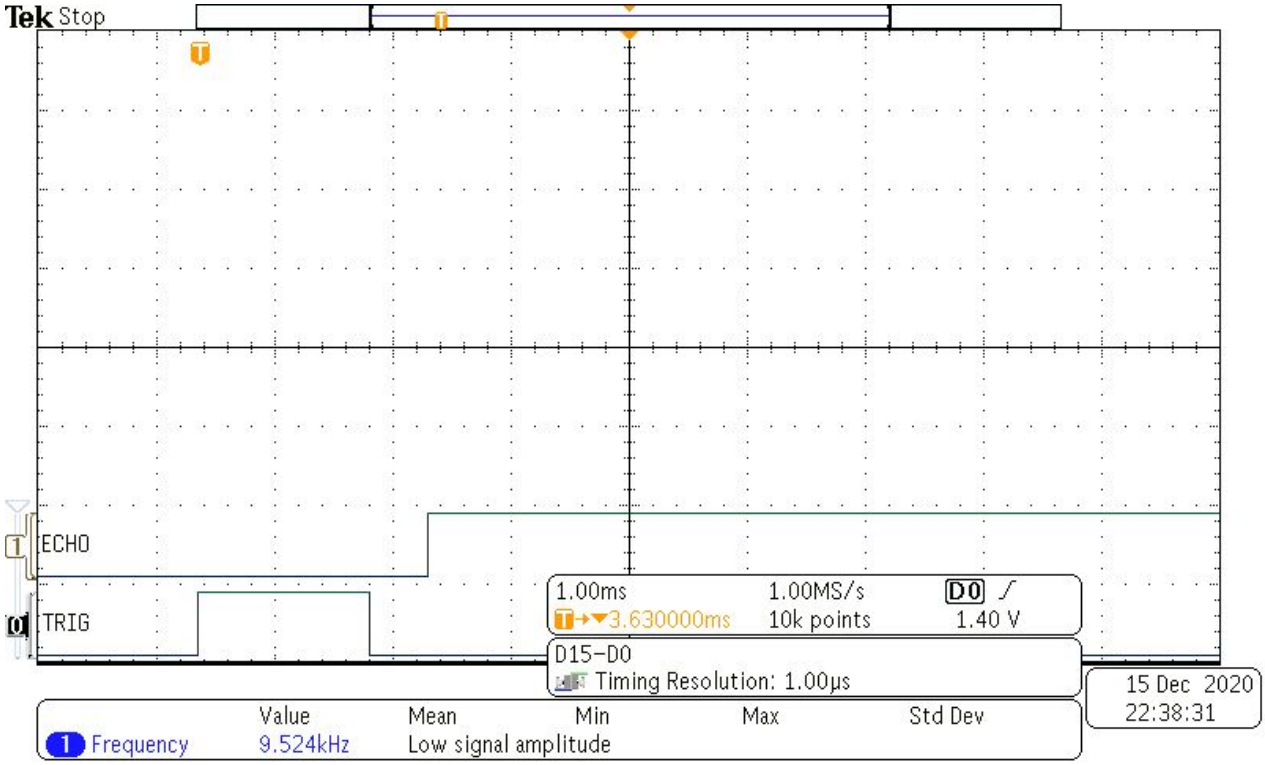
Our timer counting to ARR = 6, and Duty Cycle = 3/7

Testing

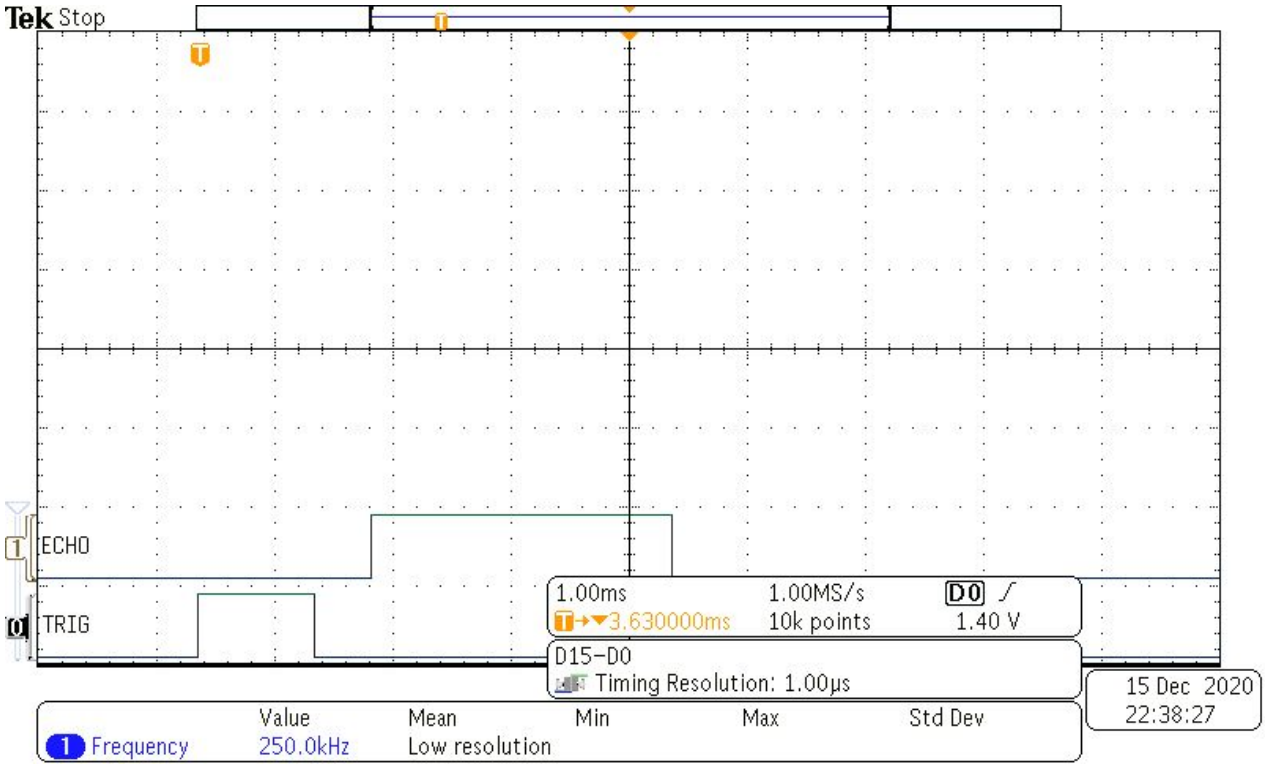
Ultrasonic Distance Sensors

Verify Sensor is Working

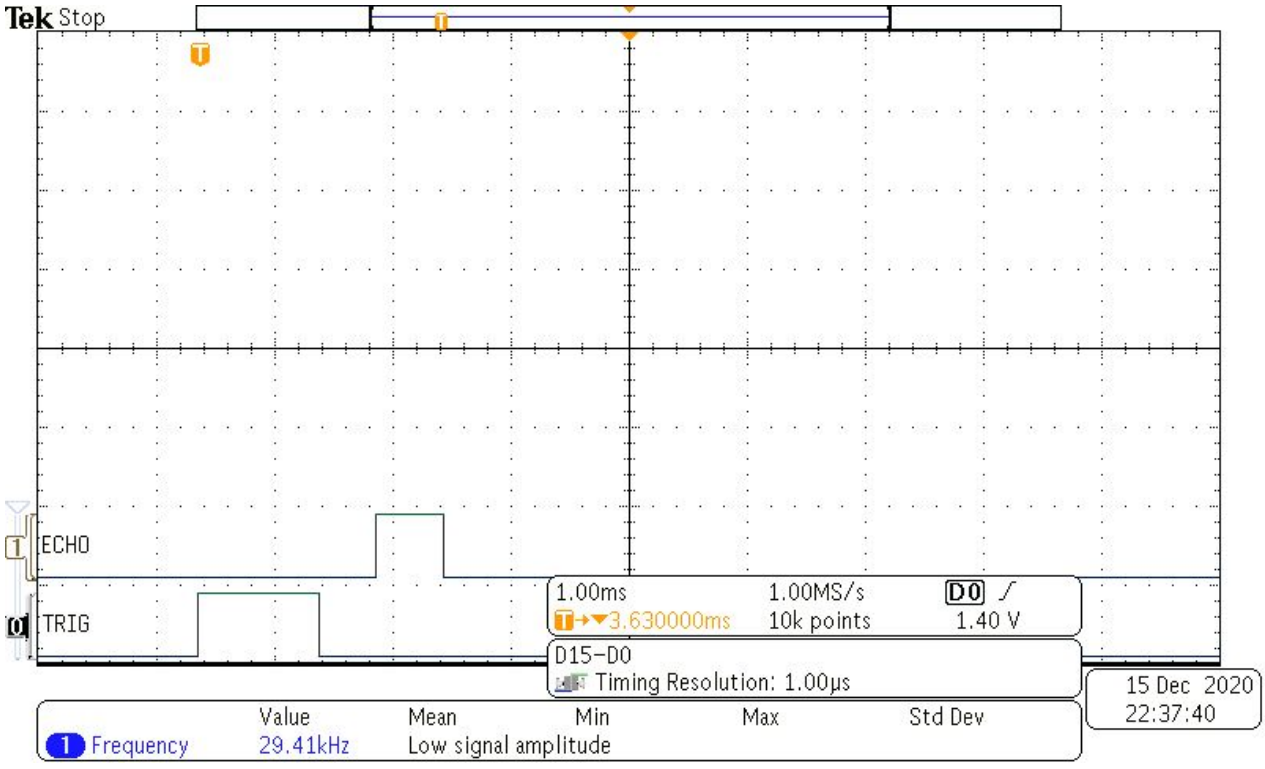
The first step to testing the ultrasonic distance sensor is to verify that it is producing the correct signals. Connect the sensor to the discovery board and write a program to send 10 microsecond pulses to the TRIG pin on the sensor. Monitor both TRIG and ECHO lines on an oscilloscope and set the oscilloscope to trigger on the rising edge of TRIG. If the sensor is working properly, you should see a reply pulse on the ECHO line around 10 microseconds after the TRIG pulse(see figures below). This pulse should be quite long if there are no objects within 2 meters of the sensor and get shorter and shorter as objects are moved closer until the pulse is only 10 microseconds or less with an object right next to the sensor. All of our sensors passed this test and behaved similarly to the results shown in the figures below.



Long pulse with no object in range.



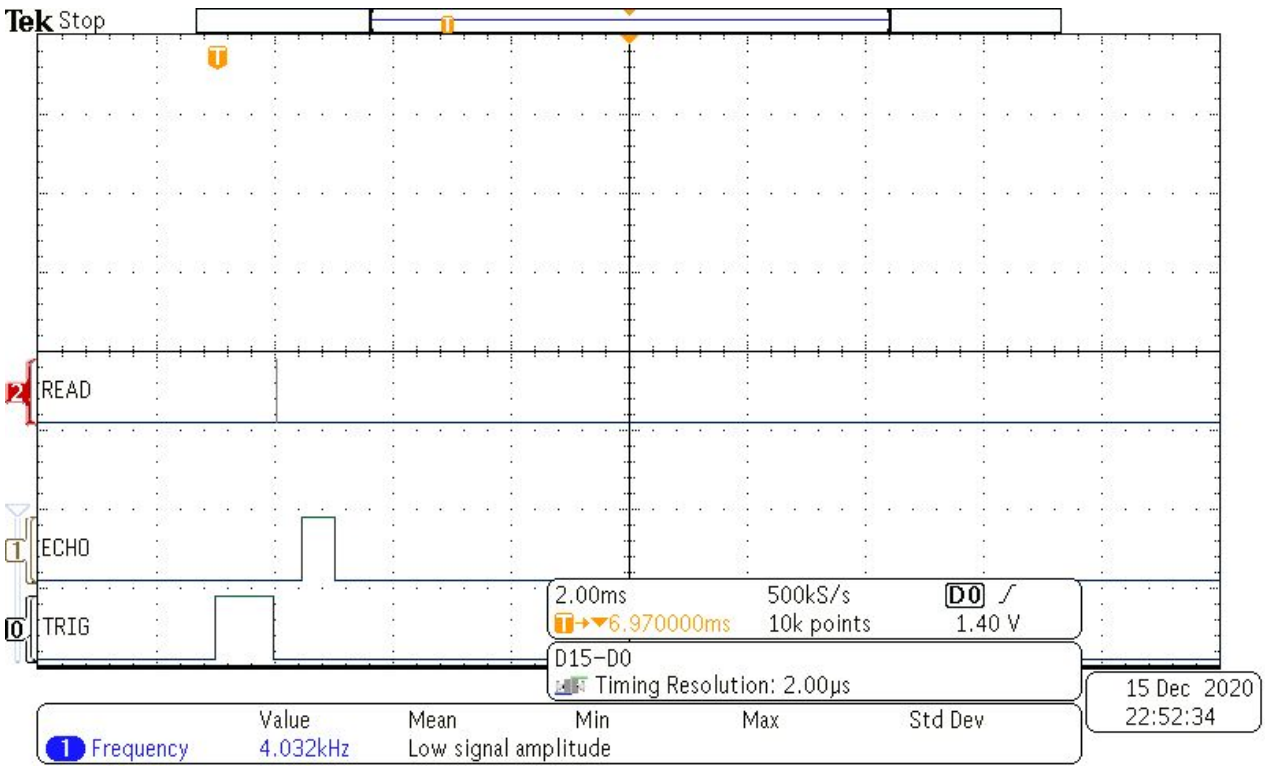
Pulse length proportional to distance



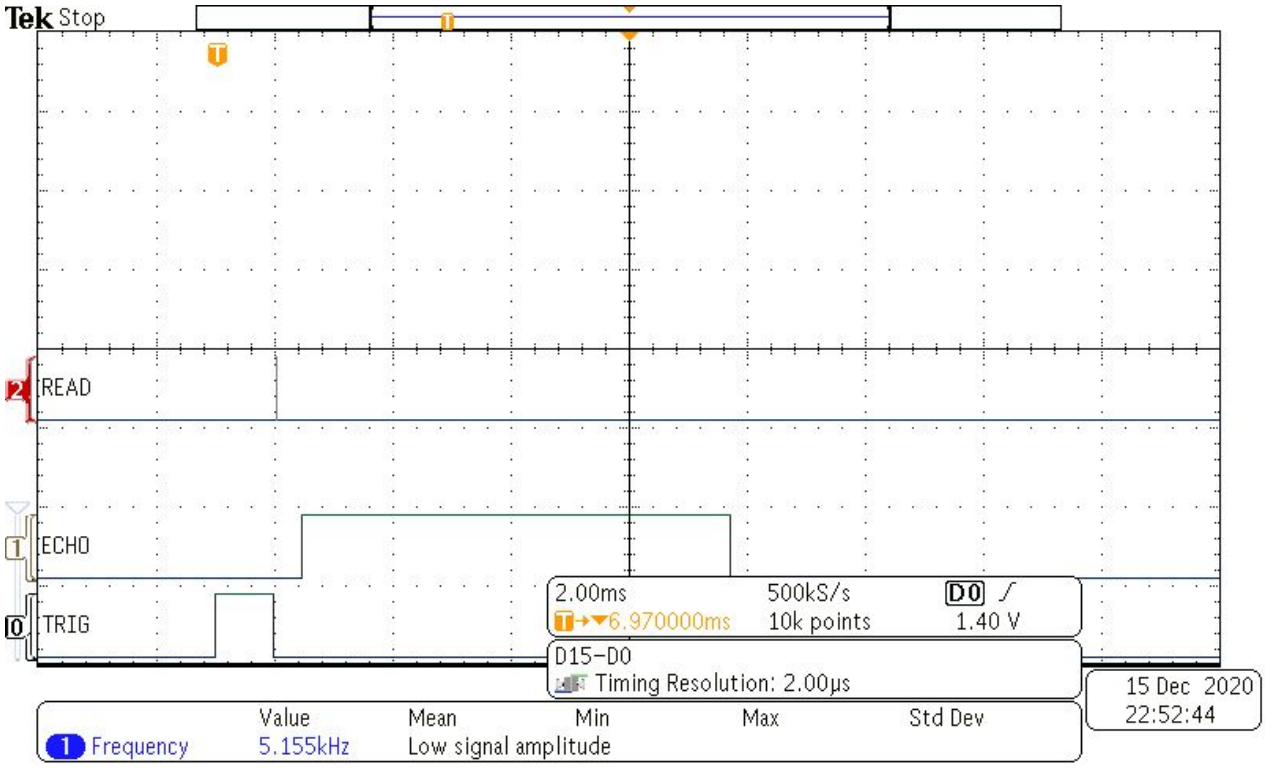
Minimum pulse received when object is directly against sensor.

Calibrate Echo Pulse Reading

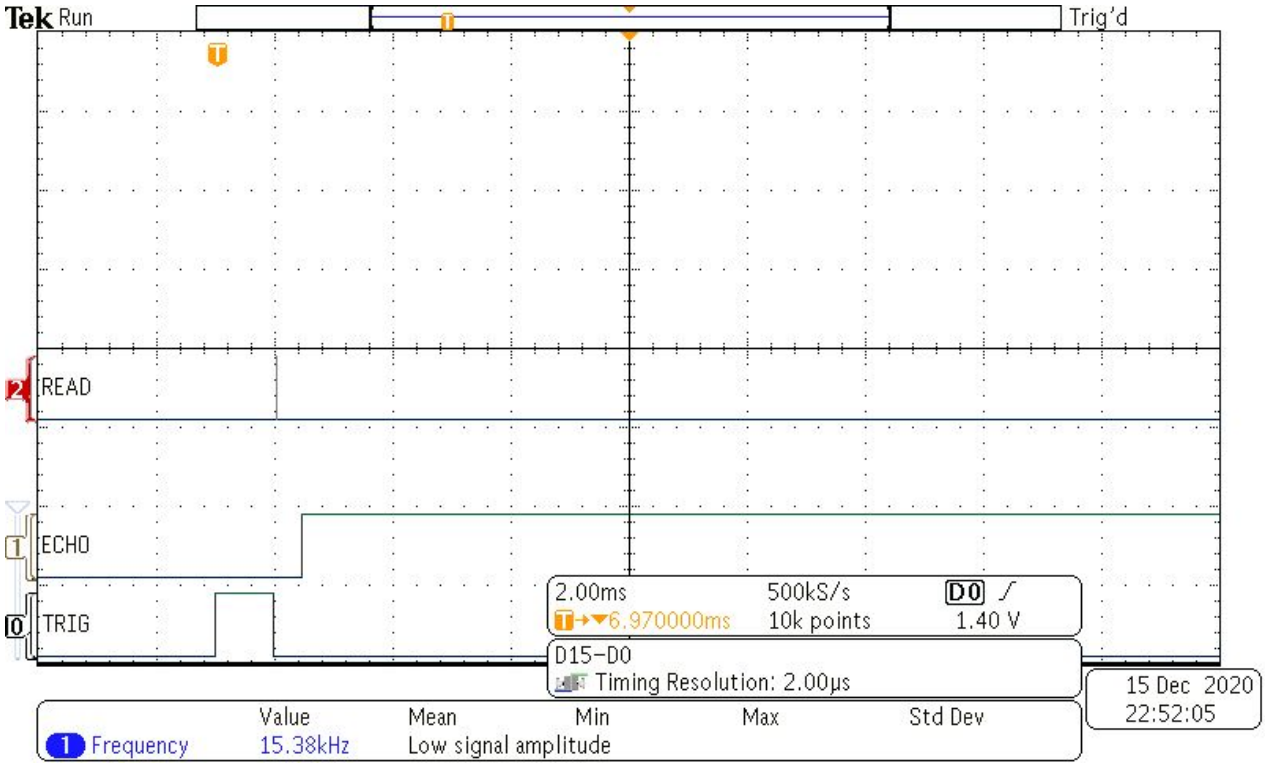
Once the sensor is tested and known to be working properly, it is important to properly calibrate the reading delay. The program to ready distance from an ultrasonic sensor must send the 10 microsecond pulse, then delay around 10 microseconds, and finally count the amount of time the echo lasts for. The easiest way to read this pulse is a simple while loop that counts how many iterations it completes while ECHO is high. In order for this approach to work, however, the loop must start while the pulse is already high, but not so late that a short pulse would be missed entirely. To test this, we pick a separate pin on the board (labeled READ in the figures) to use as a signal and set it to remain high for the duration of the while loop. When set properly, READ should trace the shape of ECHO almost exactly and look like the ECHO figures from the test above. When the delay between read and write is too small, the program will try to read before the ECHO pulse and the READ pulse will be extremely short as the while loop will instantly exit. On the other hand, too long of a delay will cause the READ pulse to be noticeably shorter than ECHO or even get cut off entirely for short ECHO pulses (see figures below). After calibration, our READ pulse very closely tracked our ECHO pulse regardless of how far an object was from the sensor.



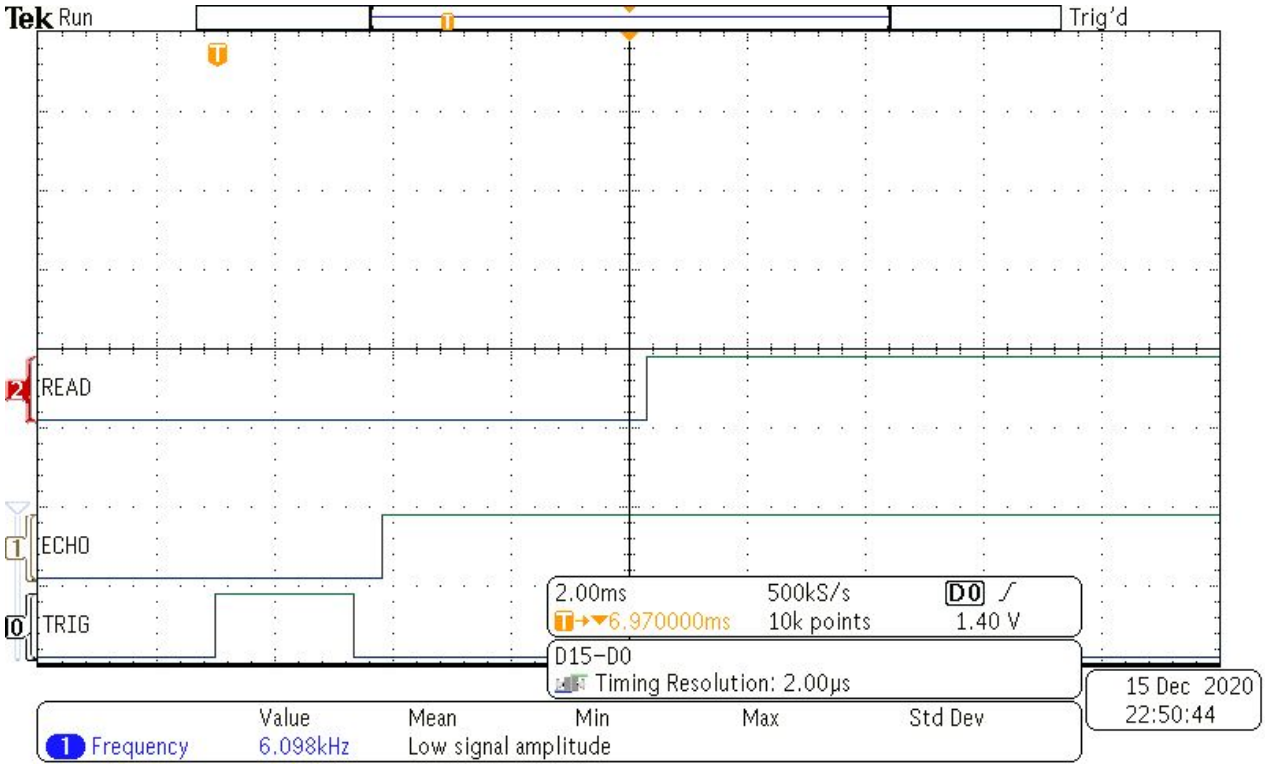
Delay too short, misses echo pulse (short range).



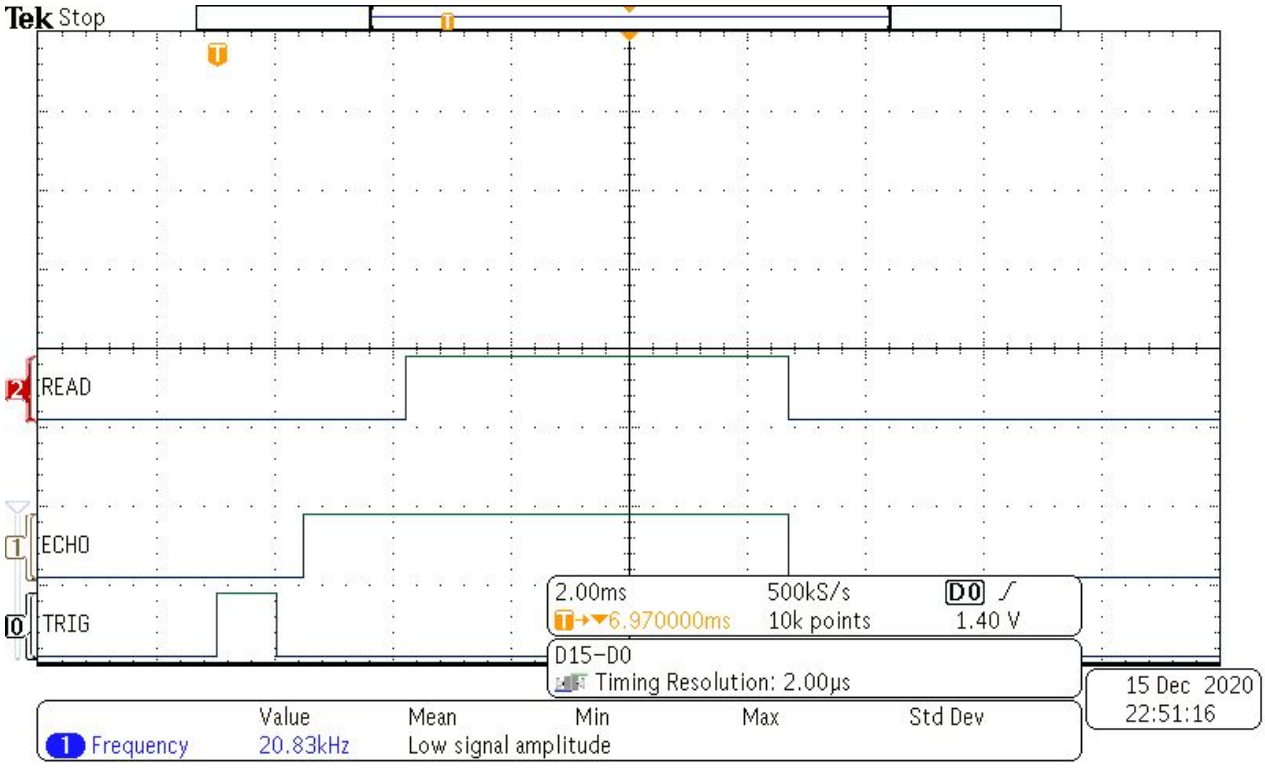
Delay too short, misses echo pulse (medium range).



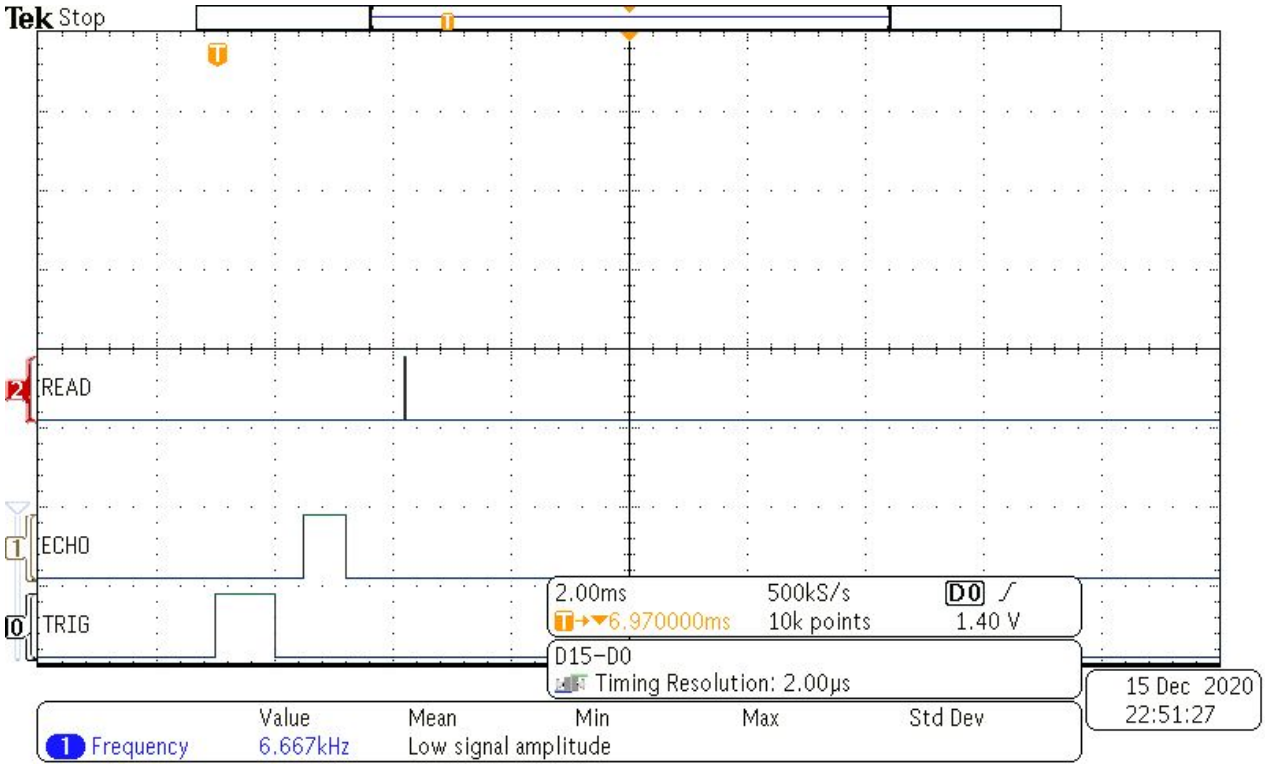
Delay too short, misses echo pulse(object out of range pulse).



Delay too long, but can still catch object out of range pulse.



Delay too long, catches medium pulse but shortens it.



Delay too long, completely misses short pulse.

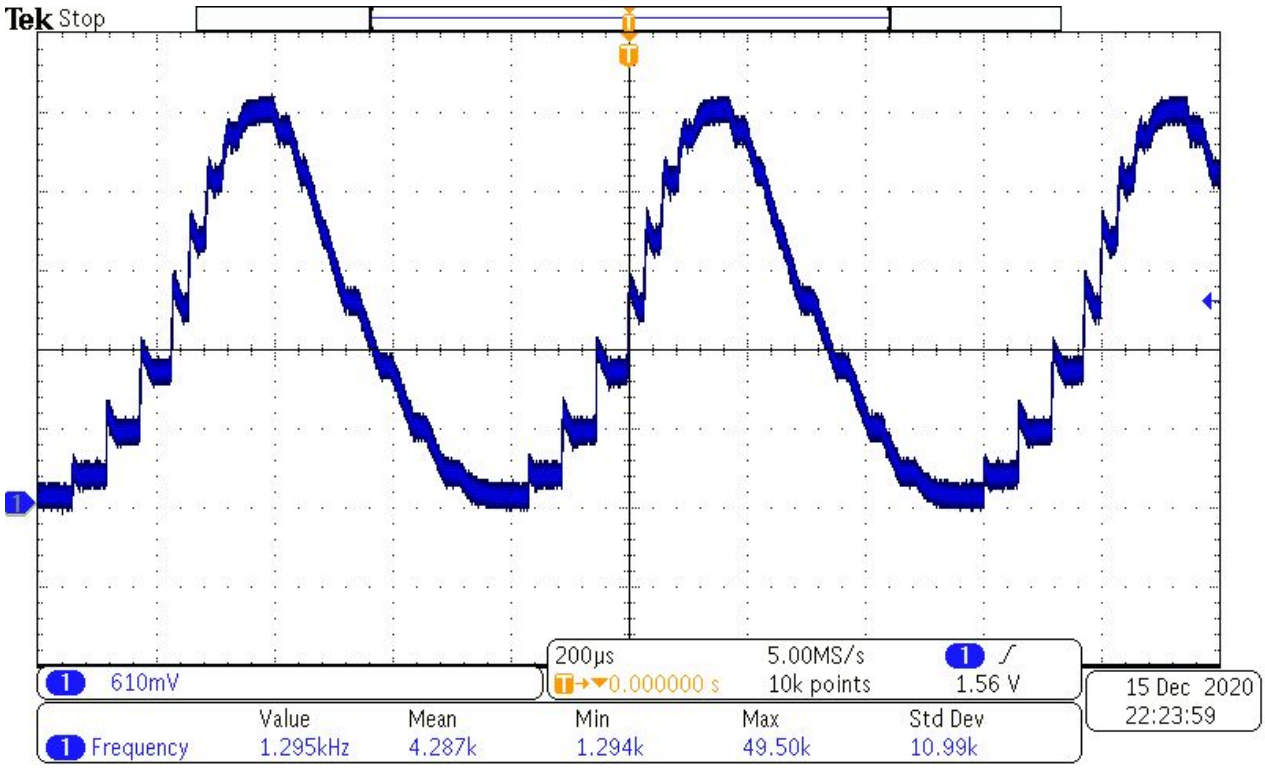
Sound and DAC

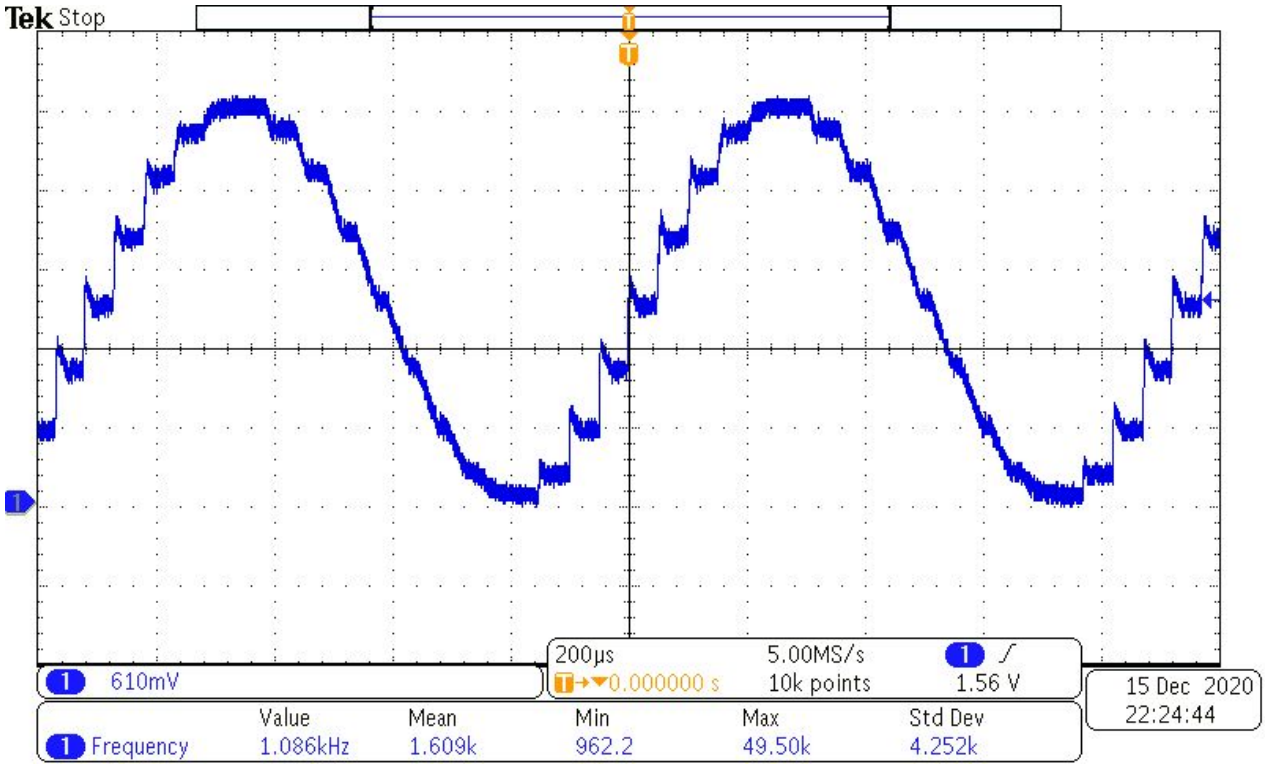
DAC Output

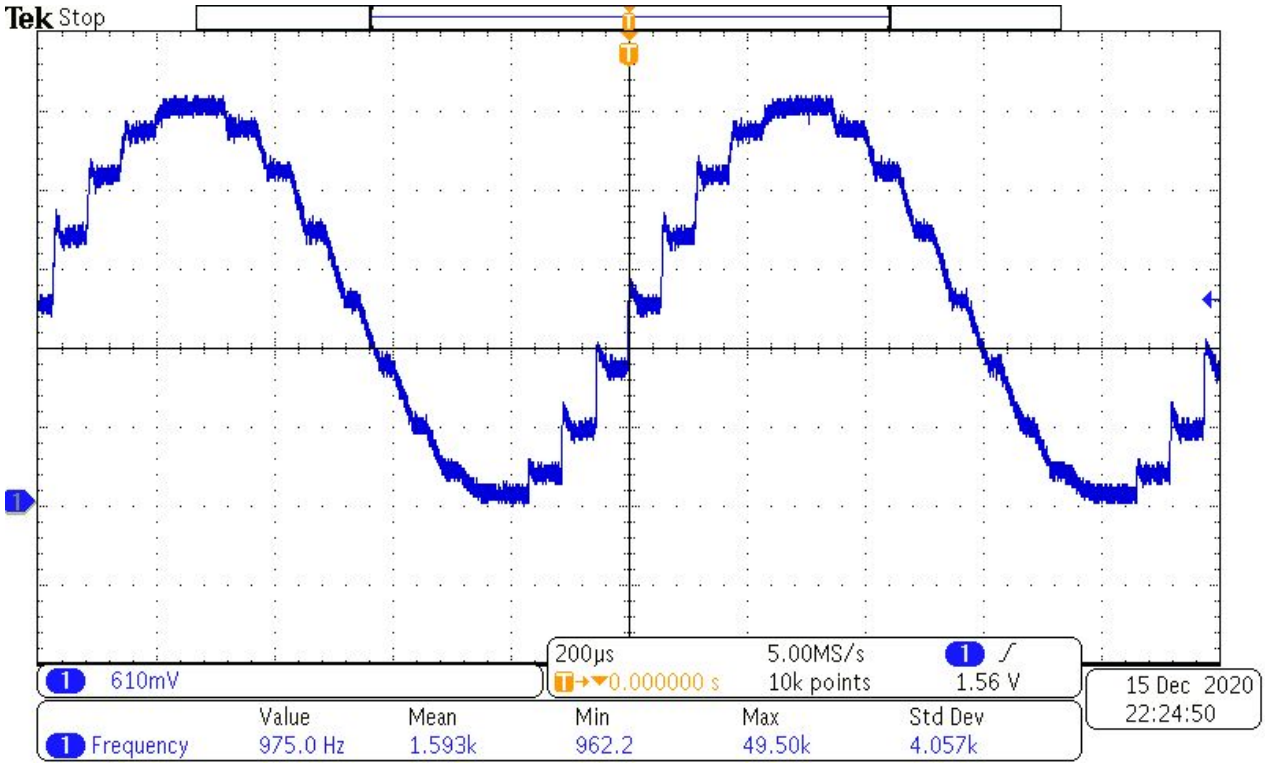
The DAC output was, in general, much easier to test than the distance sensors because the output was a repeating waveform and the measurements of interest were waveform, frequency, and amplitude, exactly what an oscilloscope is meant for. We simply connected the oscilloscope to the output pin and measured on the oscilloscope. It is important to note, however, that this is much easier before the speaker is attached. Although the piezo buzzer and earbud both decreased voltage amplitude they both left large enough waves to measure easily. The 4 ohm speaker, however, brough the voltage amplitude near zero and made all of our measurements much noisier so we always disconnected the speaker before measuring waveforms. In order to most carefully measure results, we used objects placed in front of the sensors rather than our hands as the objects had a more regular shape and, more importantly, would stay completely still so a single variable could be isolated at a time (i.e. change frequency while keeping amplitude the same or change amplitude while keeping frequency the same).

The results expected from this test are pretty straightforward. Moving the object in front of the “pitch” distance sensor should change the frequency while moving the object in front of the “amplitude” distance sensor should change the output amplitude(always moving objects on the axis perpendicular to the sensor). Note that the frequency changes in whole note steps not on a continuous spectrum and amplitude is also quantized into steps, unlike an analog theremin which

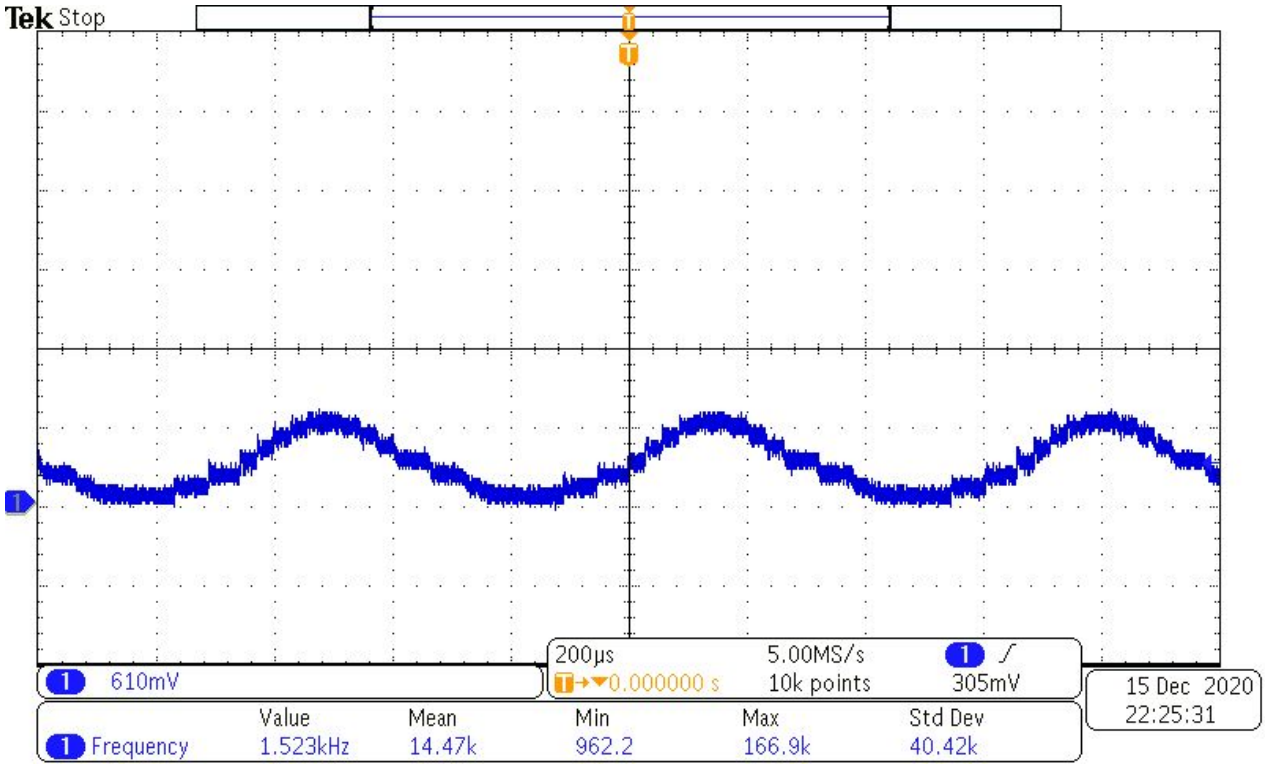
would produce both on a continuous spectrum. The figures below are screenshots from the tests, in practice, it was much easier to watch the oscilloscope in live time to verify the signals than to take each new measurement in single shot mode as was done for these screenshots.

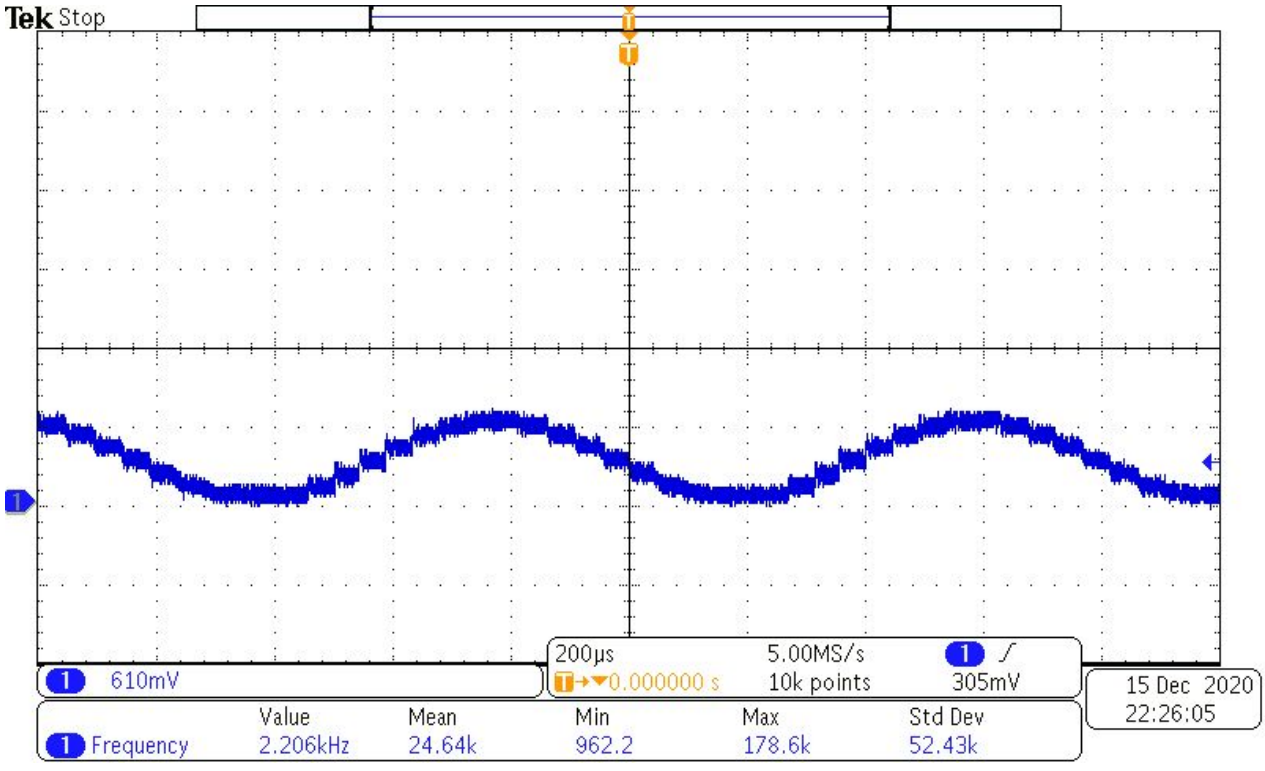


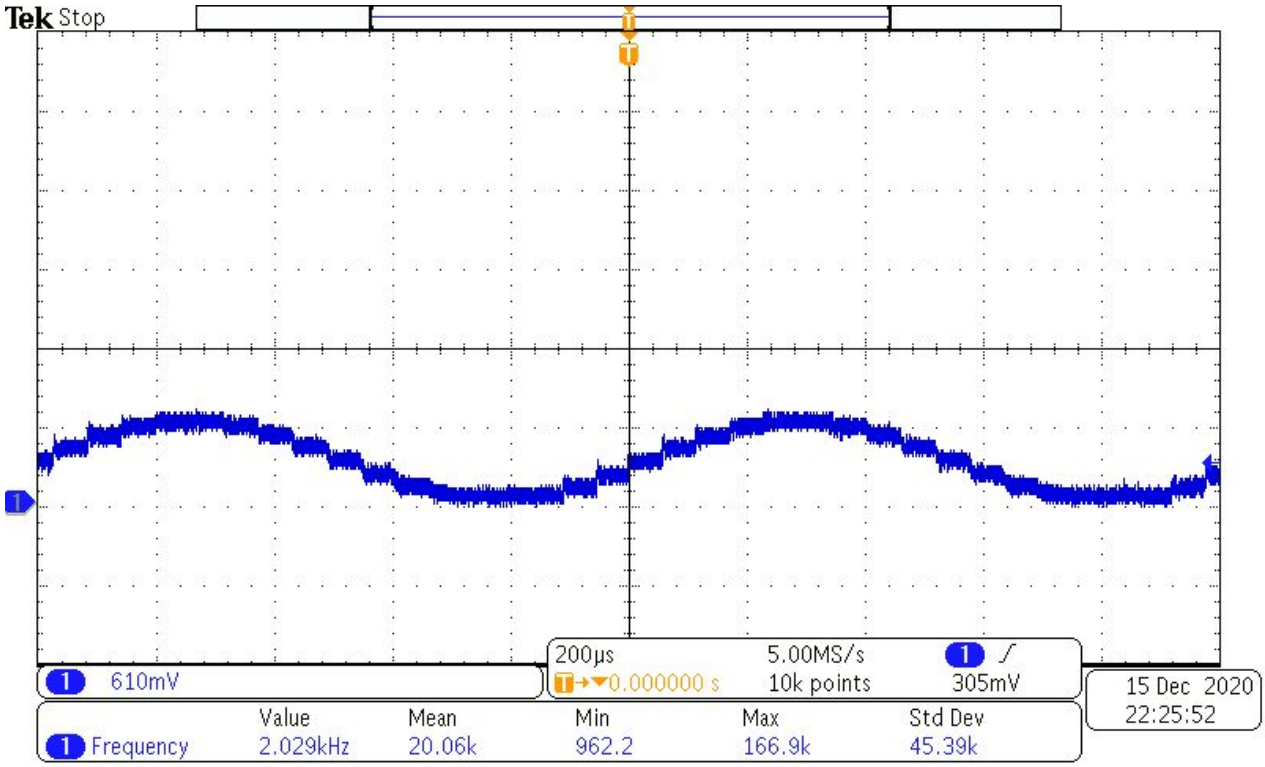




These three figures depict frequency being modulated at constant amplitude.


Tek Stop
200µs 5.00MS/s
0.000000 s 10k points
305mV
610mV
15 Dec 2020
22:25:31
Value Mean Min Max Std Dev
Frequency 1.523kHz 14.47k 962.2 166.9k 40.42k


Tek Stop
200µs 5.00MS/s
0.000000 s 10k points
305mV
610mV
15 Dec 2020
22:26:05
Value Mean Min Max Std Dev
Frequency 2.206kHz 24.64k 962.2 178.6k 52.43k



These three figures depict frequency modulation again but at a different amplitude.

Sound Output

The final test of the DAC output, of course, was simply to attach the speaker and play the theremin. The expected results, as with the previous test, were for the frequency sensor to change the pitch of the note and for the amplitude sensor to change its volume. Our results from this test were mostly qualitative, although we did also use a tuner app to check which notes we were hearing.

Although the pitch with distance as designed, we had trouble in the border zones between notes. Specifically, we intended for the notes to cleanly change at a certain distance but instead we ended up with a region in which the error in distance sensing would often cause the note to rapidly “trill” back and forth between the two notes. The amplitude change, which had looked perfect on the oscilloscope, was very noticeable but, to our ears at least, was almost more noticeable as a change in timbre than a change in pitch. This was unexpected given that the waveform appeared to have the same shape and just lower amplitude and may be some effect of the speakers we were using.

Conclusion

Our theremin technically met all of our initial design requirements but had unusual behavior at the boundaries between notes (the trills). These trills actually sounded good but were not a

desired effect and would make playing actual songs more difficult. We also added in two features that were not in our initial design but we thought were significant improvements to the project, the earbud for quiet practice and distinct notes. The distinct notes feature, in particular, makes the theremin much easier to play as, while one can still miss a note, it cannot be played out of tune (like a piano as opposed to a trombone). Our final sound was loud enough for a small audience but would have to be amplified in some way to play for a large group. We attempted to fix this with our operational amplifier but were still unable to get enough power into our 4 ohm speaker. Overall, our final result met our goal of controlling the pitch and volume of a note via hand gesture.

Appendices

Main.c

```
#include "stm32l476xx.h"
#include "sensor.h"

int note = 0;
int vol = 2;
/*
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
This code is for a digital Theremin device. The following device will have multiple inputs to the
STM32L476 discovery board,
including at least two ultrasonic distance sensors, and a speaker/buzzer depending on the need.
This device will use one distance
sensor to determine either volume or timbre. The other sensor will detect the users hand and
bend the pitch of the output to
different musical notes.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

*/
//Pin layout
                //Sensor 1 will use PA0 (trig), PA1, 5V, and GND
                //Sensor 2 will use PE10, PE11, 5V and GND
                //..... (for more sensors)
                //Speaker output will use PB6
int main(void){
        double time = 0;
        double time2 = 0;
        float dist = 0;
        float dist2 = 0;
        initialize();
```

```
	TIM2_Init();
	//TIM3_Init();
	DAC_Channel2_Init();


	//loop indefinitely
	while(1){
		sendtrigger1();                                          //outputs 10us
signal from sensor1
		delay(400);
				//delay around 10us (number determined on scope)
		time = receivetrigger1();
//receives signal and translates into how long it took the signal to come back
		sendtrigger2();
		delay(400);
		time2 = receivetrigger2();
//volume trigger
		dist = distance(time);                          //calculate distance that hand is
from sensor1
		note = determinenote(dist);                              //determine the note
depending on the distance
		dist2=distance(time2);
		determinevolume(dist2);
		switch((int)note)
	{
		case 0: //261.63
			TIM2->ARR = 75;//.4438;
//20kHz/(1+ARR) = 261.63Hz
			TIM2->CCR1 = 38;//.2219;                          //duty ratio =
50%
			break;
		case 1: //293.665
			TIM2->ARR = 67;//.1048;
//20kHz/(1+ARR) = 293.665Hz
			TIM2->CCR1 = 34;//.0524;
	//duty ratio = 50%
			break;
		case 2: //329.628
			TIM2->ARR = 59;//.6745;
			TIM2->CCR1 = 30;//.3373;                          //duty ratio =
50%
			break;
		case 3: //349.228
			TIM2->ARR = 56;//.2692;
```

```
                TIM2->CCR1 = 28;//.6346;                    //duty ratio = 50%
                break;
            case 4: //391.995
                TIM2->ARR = 50;//.0211;
                TIM2->CCR1 = 25;//.51055;                   //duty ratio = 50%

                break;
            case 5: //440
                TIM2->ARR = 44;//.4545;
                TIM2->CCR1 = 22;//.7273;                        //duty ratio =
50%
                break;
            case 6: //493.883
                TIM2->ARR = 39;//.4954;
                TIM2->CCR1 = 20;//.2477;                        //duty ratio =
50%
                break;
            case 7: //523.251
                TIM2->ARR = 37;//.2226;
                TIM2->CCR1 = 19;//.1113;                        //duty ratio =
50%
                break;

            default: //default is 261.63 (middle c)
                TIM2->ARR = 75;//.4438;
//20kHz/(1+ARR) = 261.63Hz
                TIM2->CCR1 = 38;//.2219;                        //duty ratio =
50%n
                break;
        }
        TIM2->CNT = 0;
        delay(30000);

        }
}
```

Sensor.h

```
//#ifndef __STM32L476G_DISCOVERY_FINAL
//#define __STM32L476G_DISCOVERY_FINAL
#include <stdint.h>

//sensor.c
void sendtrigger1(void);
void sendtrigger2(void);
//void delay(__IO uint32_t nCount);
float receivetrigger1(void);//sensor 1 receive
float receivetrigger2(void);
float distance(float time);
double determinenote(float dist);
void determinevolume(float dist);

//speaker.c
void playnote(int note);
void sysbuzz(void);
void Create_Sine_Table(void);
uint32_t lookup_sine(int x);
void delay(__IO uint32_t nCount);

//initialize.c
void initialize(void);

//DAC.c
void DAC_Channel2_Init(void);
void TIM3_Init(void);
void TIM2_Init(void);
void TIM3_IRQHandler(void);
void TIM2_IRQHandler(void);
void Create_Sine_Table(void);
uint32_t lookup_sine(int x);
extern volatile int v;

//#endif /* __STM32L476G_DISCOVERY_FINAL */
```

Sensor.c

```
#include "stm32l476xx.h"
#include "sensor.h"

extern int note;
extern int vol;

void sendtrigger1(){
	//needs to have PA0 set as high for 10us
	//10us has a period of 100,000Hz
		//so our delay = 16MHz/100kHz = 160
	for(int i = 0; i <= 160; i++){
		GPIOA->ODR |= 0x00000001;
	}
	GPIOA -> ODR &= 0xFFFFFFFE;
}

void sendtrigger2(){
	//needs to have PE10 set as high for 10us
	//10us has a period of 100,000Hz
		//so our delay = 16MHz/100kHz = 160
	for(int i = 0; i <= 160; i++){
		GPIOE->ODR |= 0x00000400;
	}
	GPIOE -> ODR &= 0xFFFFFBFF;
}

void delay(__IO uint32_t nCount){
	while(nCount--);
}

float receivetrigger1(){
	//while receiving, count
	//once done receiving, calculate distance from time the input was high
	float count = 0;
	float time = 0;

	GPIOE->ODR |= 0x00000400; //For testing on scope

	while((GPIOA->IDR & 0x00000002) != 0){
		count += 1;
		if(count >= 20000){ //if count counts for too long, break
			break;
```

```
                }
        }

        GPIOE -> ODR &= 0xFFFFFBFF;//For testing on scope

        time = count * (6.25 * (10e-8));

        return time;
}


float receivetrigger2(){
                        //while receiving, count
        //once done receiving, calculate distance from time the input was high
        float count = 0;
        float time = 0;
        while((GPIOE->IDR & 0x00000800) != 0){
                count += 1;
                if(count >= 20000){
                        break;
                }
        }
        time = count * (6.25 * (10e-8));

        return time;
}

float distance(float time){
        //Distance (cm) = (Time x SpeedOfSound) / 2
        float speedofsound = 500000;  //Adjusted based on experimental results *note the real
speed of sound* 500000 is a good starting value
        float dist = (time * speedofsound) / 2;
        return dist; //returns in cm
}

double determinenote(float dist){
        //Min note is middle C, going up the scale, our highest note is D5

        //sensor has a range of 2cm to 400cm, I decided our range should be 3cm to 46cm (or
1.5ft)
        //increments in 5.4cm from 3cm to 46cm
                if (dist <= 8.4){ //note 0
                        //note = 261.63;          //middle C freq in Hz
                        note = 0;
```

```
		}
		else if (dist > 8.4 && dist <= 13.8){ //note 1
			//note = 293.665; //D4 freq
			note = 1;
		}
		else if (dist > 13.8 && dist <= 19.2){ //note 2
			//note = 329.628; //E4 freq
			note = 2;
		}
		else if (dist > 19.2 && dist <= 24.6){ //note 3
			//note = 349.228; //F4 freq
			note = 3;
		}
		else if (dist > 24.6 && dist <= 30){ //note 4
			//note = 391.995; //G4 freq
			note = 4;
		}
		else if (dist > 30 && dist <= 35.4){ //note 5
			//note = 440;			//A4 freq
			note = 5;
		}
		else if (dist > 35.4 && dist <= 40.8){ //note 6
			//note = 493.883;	//B4 freq
			note = 6;
		}
		else if (dist > 40.8 && dist <= 46.2){ //note 7
			//note = 523.251;	//C5 freq top note of scale
			note = 7;
		}
		else if (dist > 46.2){ //note 7 again? //Note 8 so we can play a full scale C to C?
			note = 7; //adjust to be note past our furthest distance from the sensor
		}
	return note;
}

void determinevolume(float dist){
	//The closer to the sensor the louder the speaker is and vice versa
	//int vol = 0;

	//sensor has a range of 2cm to 400cm, I decided our range should be 3cm to 46cm (or
1.5ft)
	//increments in 5.4cm from 3cm to 46cm
		if (dist <= 3 || (dist >= 3 && dist <= 8.4)){
```

```
                        vol = 5;//2.50; //2.5 times our original volume (sets amplitude to 2.5 for
sinewave if possible)
                }
                if (dist > 8.4 && dist <= 13.8){
                        vol = 5;//2.25; //2.25 times our original volume
                }
                if (dist > 13.8 && dist <= 19.2){
                        vol = 3; //2 times our volume
                }
                if (dist > 19.2 && dist <= 24.6){
                        vol = 3;//1.75; //1.75 times our volume
                }
                if (dist > 24.6 && dist <= 30){
                        vol = 3;//1.50; //1.5 times our volume
                }
                if (dist > 30 && dist <= 35.4){
                        vol = 3;//1.25; //1.25 times our volume
                }
                if (dist > 35.4 && dist <= 40.8){
                        vol = 1;//1.00;
                }
                if (dist > 40.8 && dist <= 46.2){
                        vol = 1;//0.75;
                }
                if (dist > 46.2){
                        vol = 1;//0.50; //half our regular volume, if the amplitude wasnt adjusted
                }
        return;// vol;
}
```

Initialize.c

```
#include "stm32l476xx.h"
#include "sensor.h"
void initialize(){
	//Enable clocks
	RCC -> CR |= RCC_CR_HSION;
	while(((RCC -> CR) & RCC_CR_HSIRDY) == 0){
		//wait for hsi to be ready
	}
		//Sensor 1 will use PA0, PA1, 5V, and GND
		//Sensor 2 will use PE10, PE11, 5V and GND
		//..... (for more sensors)
		//Speaker output will use PB6
	//enable GPIO port A
				- Sensor 1 initialize
	RCC -> AHB2ENR |= RCC_AHB2ENR_GPIOAEN;
	GPIOA->MODER &= 0xFFFFF3F0;
	//Set PA0 as output, PA1 as input
	GPIOA->MODER |= 0x00000001;
	//For DAC
	// Now set PA5 to output mode (01)
	GPIOA->MODER |= 0x00000C00;

	//Enable GPIO port E
				- Sensor 2 initialize
	RCC -> AHB2ENR |= RCC_AHB2ENR_GPIOEEN;
	GPIOE->MODER &= 0xFF0FFFFF;
	//Set PE10 as output and PE11 as input
	GPIOE->MODER |= 0x00100000;

	//Enable GPIO port B
				- Speaker initialize
	RCC -> AHB2ENR |= RCC_AHB2ENR_GPIOBEN;
	GPIOB->MODER &= 0xFFFFCFFF;
	//Set PB6 as output
	GPIOB->MODER |= 0x00001000;
}
```

DAC.c

```
#include "stm32l476xx.h"
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "sensor.h"

volatile int v = 0;
signed int sine_table[91];

extern int note; //extern just allows this variable to be global across all files. It's declared in main.c
extern int vol;
extern int flag;

void DAC_Channel2_Init(void)
{
	//DAC channel 2: DAC_OUT2 = PA 5
	//enable DAC clock
	RCC->APB1ENR1 |= RCC_APB1ENR1_DAC1EN;

	//Disable DAC
	DAC->CR &= ~(DAC_CR_EN1 | DAC_CR_EN2);

	DAC->MCR &= ~(7U<<16);                                          //mode = 000;

	//Enable Trigger for DAC channel 2
	DAC->CR |= DAC_CR_TEN2;

	//clear trigger selection bits for channel 2
	DAC->CR &= DAC_CR_TSEL2;                                  //set to software trigger

	//select TIM4_TRGO as the trigger of DAC channel 2 (100)
	DAC->CR |= (DAC_CR_TSEL2_2);
	//DAC->CR |= DAC_CR_EN2;

	Create_Sine_Table();

	//Enable DAC Channel 2
	DAC->CR |= DAC_CR_EN2;

}
```

```
void TIM2_Init(void)  //timer for DAC frequency updated every 2ms
{
	RCC->APB1ENR1 |= RCC_APB1ENR1_TIM2EN;   //Enable Clock of Timer 2
	TIM2->CR1 &= ~TIM_CR1_CMS;                    //Clear Edge-Aligned mode
bits
	TIM2->CR1 &= ~TIM_CR1_DIR;                    //Counting direction:
Up-counting;

	TIM2->CR2 &= ~TIM_CR2_MMS;                    //Clear master mode selection
bits
	TIM2->CR2 |= TIM_CR2_MMS_2;                   //select 100 = OC1REF as
TRGO

	//OC1M:  output Compare 1 mode
	TIM2->CCMR1 &= ~TIM_CCMR1_OC1M;
	TIM2->CCMR1 |= TIM_CCMR1_OC1M_1 | TIM_CCMR1_OC1M_2;

	//Timer driving frequency = 80 MHz/(1 + PSC) = 80 MHz/(1+799) = 10kHz
	//Trigger frequency = 10kHz/(1 + ARR) = 1kHz - 100Hz
	TIM2->PSC = 2;                                          //max 65535;
	TIM2->ARR = 10;                                         //max 65535;
	TIM2->CCR1 = 5;                                         //Duty ration
50%

	TIM2->DIER |= TIM_DIER_UIE;                   //Enable update interrupts
	NVIC_EnableIRQ(TIM2_IRQn);                    //Enable TIM2 Interrupt in
NVIC_EnableIRQ
	NVIC_SetPriority(TIM2_IRQn, 3);               //Set TIM2 interrupt priority to 3

	TIM2->CCER |= TIM_CCER_CC1E;                  //OC1 signal is output
	TIM2->CR1 |= TIM_CR1_CEN;                     //Enable Timer
}


void TIM2_IRQHandler(void)                                  //put next value from
table into DAC
{
		v += 20;
		v = v % 360;
		DAC->DHR12R2 = lookup_sine(v)/vol;
		DAC->CR |= DAC_CR_EN2;
	//this line may be neccessary
	//DAC->SWTRIGR |= DAC_SWTRIGR_SWTRIG2;
```

```
        if((TIM2->SR & TIM_SR_UIF) != 0)
        {
                TIM2->SR &= ~TIM_SR_UIF;
        }
}

void Create_Sine_Table(void)
{
   //int i;
   float sf;

   // for 12-bit ADC, [0, 4095(0xFFF)];
   for (int i = 0; i <= 90; i++){
      sf = sin(3.14159 * i /180);
      sine_table[i] = (1 + sf) * 2048;
      if(sine_table[i] == 0x1000){
         sine_table[i] = 0xFFF;
                                    }
   }
   return;
}

uint32_t lookup_sine(int x)
{
   // x is the input in degrees
   x = x % 360; // shift into first period
   if (x < 90) return (sine_table[x]);
   if (x < 180) return (sine_table[180-x]);
   if (x < 270) return ((4096 - sine_table[x-180]));
   return ((4096 - sine_table[360-x]));
}
```