ECE 5780 Lab 2 Report

Henry Riker
A02205299

Jonathan Newman
A02300100

Lab Objectives.
The purpose of this lab is to learn how to generate audio output via a DAC and an audio amplifier using FreeRTOS, Keil uVision, and the STM32L476 Nucleo-64 Board. In this lab, we were tasked with adding a third functionality to our system from the previous lab: a 440Hz sine wave emitted through a speaker connected to the circuit with an op-amp.

Procedure.
Provided in this lab was a manual for the lm326 speaker we used. On page 10 of this manual, we found an example circuit diagram using a single input to drive the speaker through the op-amp. We wired our circuit to match this diagram almost exactly, using a discrete voltage divider in lieu of a potentiometer.

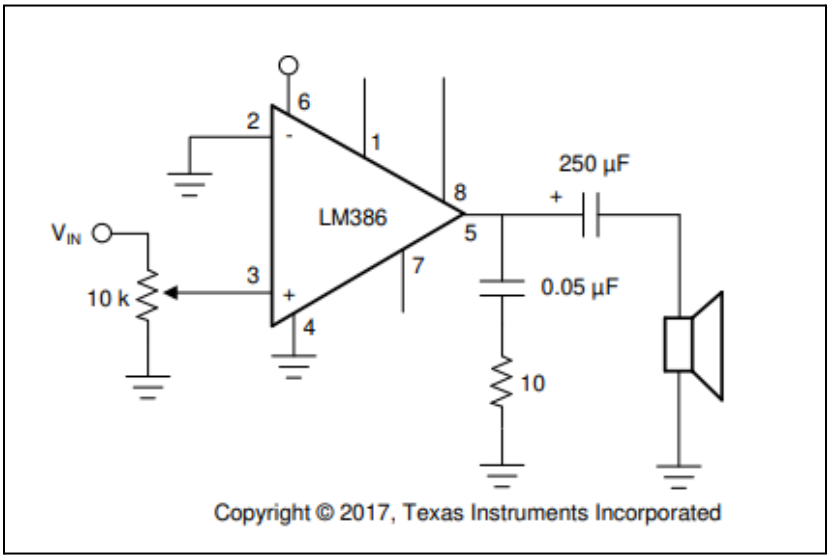


Figure 1. *The diagram on page 10 of the LM386 manual that inspired our circuit design.*

The LM386 opamp has a minimum gain of 20, and the GPIO pins of the Nucleo board can output a maximum of 3.3 V. This means that without a voltage divider, we would be cooking our speaker with 66 V, far above its maximum voltage rating. Our voltage divider was built out of a 222 ohm resistor and a 10 ohm resistor, with a gain of 0.0431, ensuring that no matter how badly our code failed, our speaker would only undergo 2.845 V.

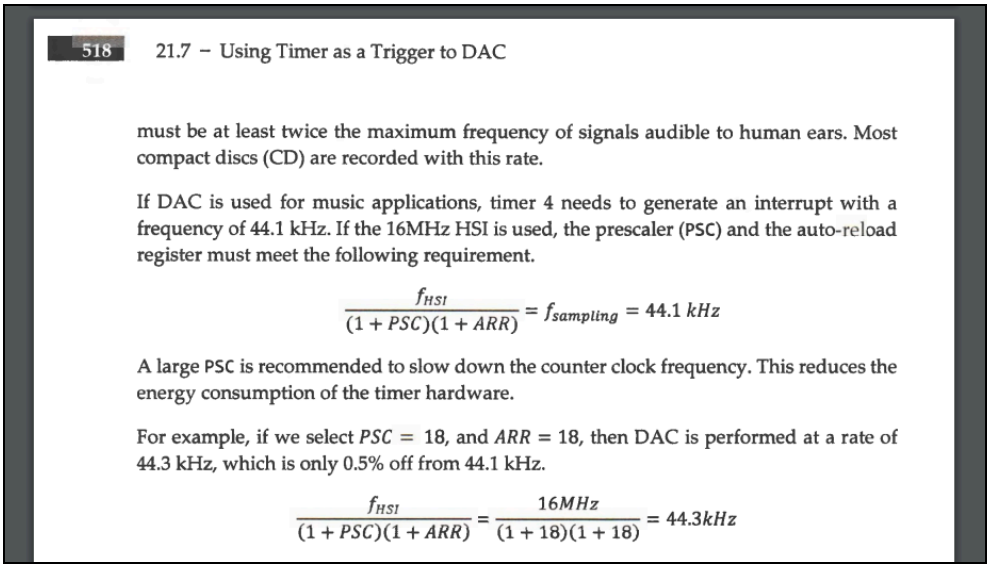
Our starting place was our code for Lab 1. In Lab 1, we had already programmed two tasks that ran simultaneously. One of them ran the button and detected presses, and the other ran the LED. It would be relatively easy, we thought, to add a third function that played a sine wave on the speaker.

We used a sine table generator (which was also provided in the lab description) to generate a sine table. At first, we generated a table with 64 entries, however; we later decided that it would be easier to generate a table with 360 entries. That way, the variable we used to call from the table could simply be the number of degrees through the sinewave to which we wanted to set the speaker. We wrote a function, lookup_sine(), that returned the desired degree value from the table.

Page 515 of the Zhu textbook shows how to enable DAC_OUT2. However, DAC_OUT2 is connected to GPIO pin A5 in its analog mode, which happens to be the same pin that our LED is using. Not wanting to sinusoidally cycle our LED, we decided to instead modify the code from the book to enable DAC_OUT1, which is tied to pin A4. We also copied code from the previous page, but realized that it was written for an incomplete sine table, and so we simplified it greatly so that only the necessary lines were in our code.

From there, we consulted the Zhu textbook further and learned how to initialize our timer. We did this, and wrote a SystemCoreClockUpdate(); in main() to reset it. After this, we spent over an hour fighting our own misunderstanding of units and c function prototypes until our code was once again able to compile.

In class the following day, Dr. Phillips recommended the use of the Nucleo's onboard HSI clock, as the system clock was already in use by FreeRTOS. We also calculated in class a prescalar of 568. We returned to the lab and updated our code to make use of this information, but we were then corrected by many of our classmates and told that it would be better to set our PSC to 18 (effectively multiplying by a factor of 19) and our ARR to 18 as well (effectively multiplying by 19 again). We would later learn that the majority of our classmates had corrected us erroneously, and that the ARR should be set higher than 18. Our suspicion of where this erroneous belief originated is an equation on page 517 of the Zhu textbook, which correctly illustrates that the use of the HSI clock with the PSC and ARR both set to 18 will produce an interrupt frequency of 44.3 kHz.



518 21.7 – Using Timer as a Trigger to DAC

must be at least twice the maximum frequency of signals audible to human ears. Most compact discs (CD) are recorded with this rate.

If DAC is used for music applications, timer 4 needs to generate an interrupt with a frequency of 44.1 kHz. If the 16MHz HSI is used, the prescaler (PSC) and the auto-reload register must meet the following requirement.

$$\frac{f_{HSI}}{(1+PSC)(1+ARR)} = f_{sampling} = 44.1\ kHz$$

A large PSC is recommended to slow down the counter clock frequency. This reduces the energy consumption of the timer hardware.

For example, if we select $PSC = 18$, and $ARR = 18$, then DAC is performed at a rate of 44.3 kHz, which is only 0.5% off from 44.1 kHz.

$$\frac{f_{HSI}}{(1+PSC)(1+ARR)} = \frac{16MHz}{(1+18)(1+18)} = 44.3kHz$$

Figure 2. *The equation in the Zhu textbook that we suspect led our classmates astray.*

At this point, our code compiled, but our circuit made no noise. After an embarrassing amount of debugging, we realized that the problem was that while we had written a function for the speaker, we had never actually used xTaskCreate() to turn it into a third function. We remedied this simple error and suddenly our code ran.

However, our code still did not do anything. The code that was copied from Lab 1 (the task watching the button and the task running the LED) worked just fine, implying that the error was in our newly implemented functions. We debugged for some time, but ended up going home unsuccessful. One thing we did notice, however, is that our “degrees” variable (indicating which value to pull from the sine table) would sometimes set itself to random numbers during lines of code that were not supposed to affect it.

Before we left the lab for the day, we decided to test our speaker with an oscilloscope to see what we could hear. True to our expectations, there was no major signal. There was, however, a significant amount of noise with a frequency of 83 MHz. As far as we could tell, our oscilloscope was acting as an FM radio antenna.

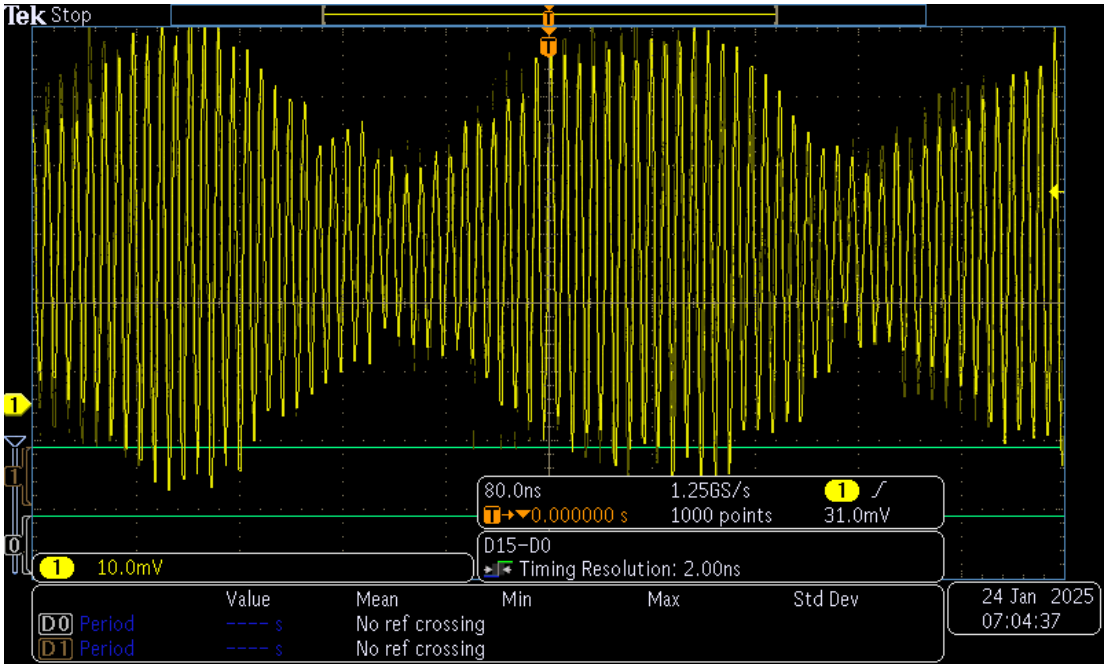


Figure 3. *FM radio waves on our oscilloscope. Note the frequency modulation.*

Jon resolved to continue debugging until we successfully got a signal. After reviewing the timer initialization code he realized that we had not enabled the capture compare interrupt. Additionally, he corrected some minor typos that could potentially affect the code. Finally, he added code to start the software trigger. After making these changes the speaker finally made a noise. However, the note played by the speaker was garbled and noticeably lower than the desired 440 Hz signal. Upon closer inspection with the oscilloscope, we found that we had created a sine wave, but it was only being transmitted one-third of the time.

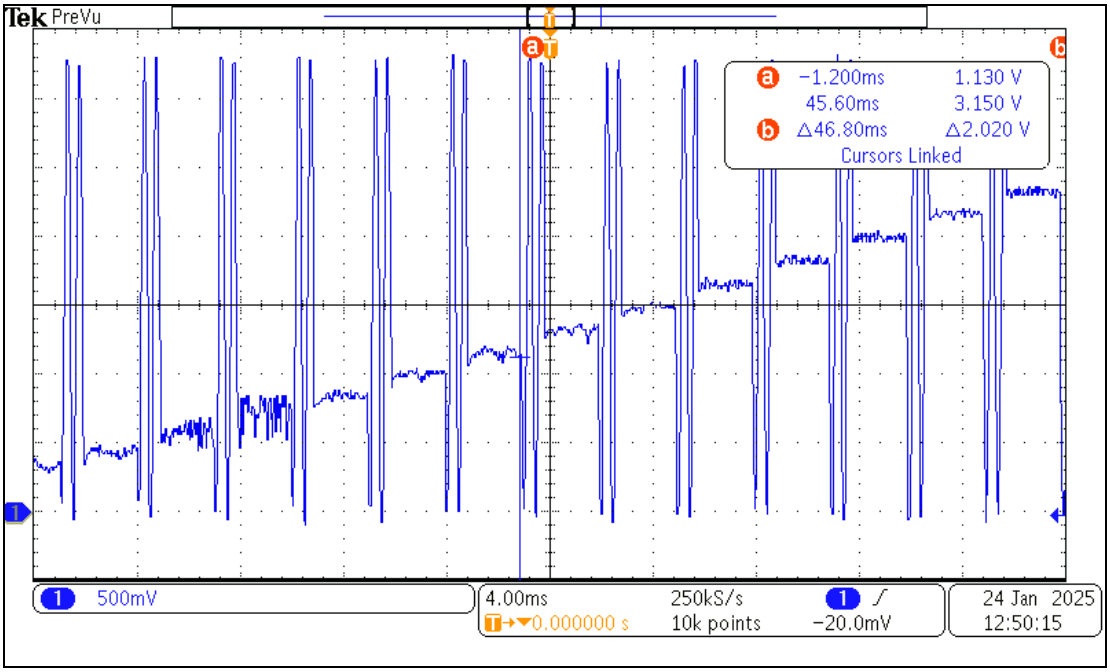


Figure 4. *One third of a successful output signal.*

Seeing that our “neutral” output voltage was slowly climbing, we turned the device back on via the onboard button and took another sample over a longer period of time. Zooming out led us to another surprise. Instead of one sinewave, we had managed to create two. One of them had a very high frequency and was only present one-third of the time, and another had a much lower frequency and was present for the other two-thirds of the time.

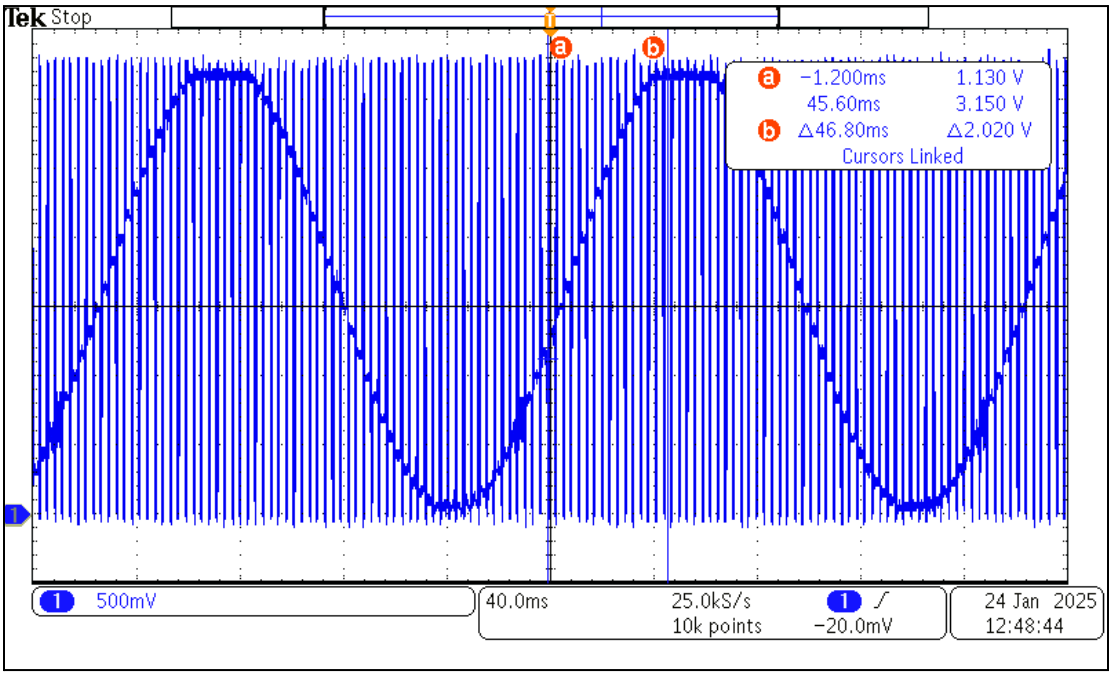


Figure 5. *Two alternating sinewaves.*

Two days later, Henry came to the lab early for the sake of showing this error to Dr. Phillips. However, he mistakenly ran old code, resulting in complete failure of the circuit. Henry began debugging, noting some details that we had not yet noticed. Eventually, Jon showed up as well and discovered Henry's error. Applying his tweaks to the more current code resulted in an even more perplexing scenario. Suddenly, we had an output waveform, and we were able to tune it to roughly 440 Hz (467 Hz) before it inexplicably stopped responding to any tweaks of PSC or ARR and stubbornly held frequency.

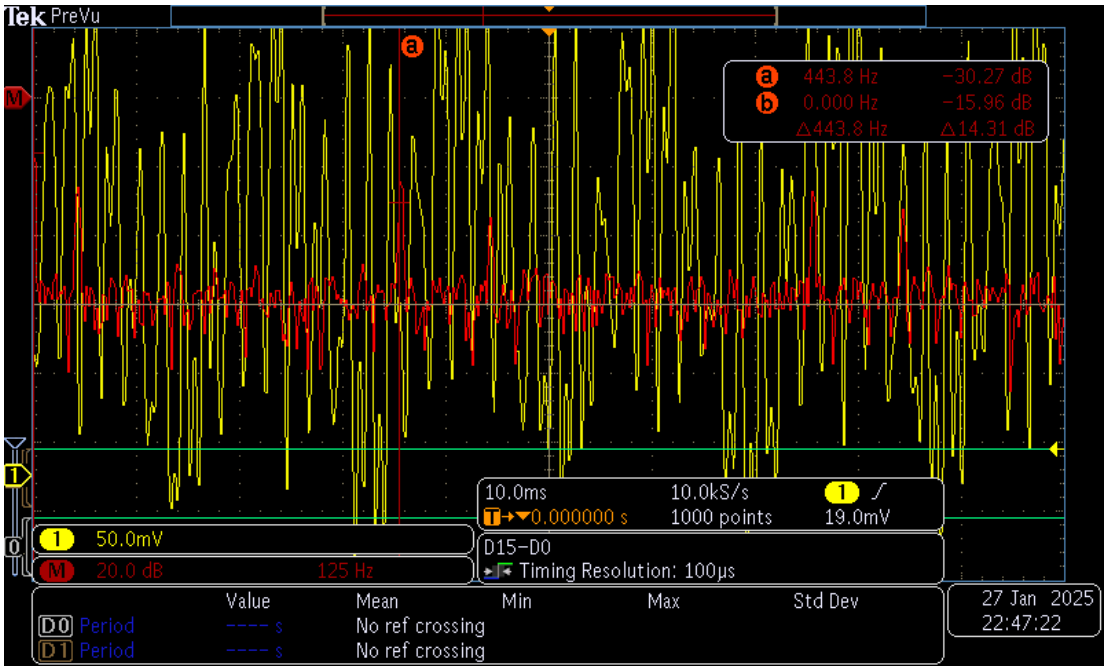


Figure 6. *A waveform best described as "horrible".*

Ready to give up, Henry went and summoned Dr. Phillips in the hopes that our waveform would merit partial credit. We were able to successfully tune it down to 440 Hz by adjusting configTICK_RATE_HZ in FreeRTOSConfig.h, literally slowing down the entire microcontroller. However, upon Dr. Phillips' arrival, he pointed out that we had not correctly handled our interrupts. In fact, we had not even implemented an interrupt handler at all, instead implementing the sine wave in the same task that we used to enable the LED. Henry was not available to finish the project, but Jon was willing to attempt to implement an interrupt handler in the limited time that we had remaining.

Fortunately, most of the requisite code had already been written, it had just been implemented in the wrong function. Jon began by following the template found in page 519 of the textbook.

```
void TIM4_IRQHandler() {

  if( (TIM4->SR & TIM_SR_CC1IF) != 0 ) {

    // Data stored in the DAC_DHRx register are automatically transferred
    // to the DAC_DORx register after one APB1 clock cycle.

    // When using dual channels, the values stored in a shared register
    // DAC->DHR12RD = sin(v)<<16 | sin(v);

    // When not using dual channels, they are set separately
    // DAC->DHR12R1 = sin(v); // DAC channel-1 12-bit Right aligned data
    DAC->DHR12R2 = sin(v); // DAC channel-2 12-bit Right aligned data

    // Adjust v appropriately for desired sine waveform frequency.
    v += degrees_desired;  // Must calculate degrees_desired. Not shown here.
    if (v >= 360) v = 0;

    // Clear CC1IF flag to prevent mistakenly re-entering the interrupt.
    // CC1IF is cleared (1) by software or (2) by hardware if CCR1 is read.
    // The handler must clear CC1IF because CCR1 is not read.
    TIM4->SR &= ~TIM_SR_CC1IF;
  }

  if( (TIM4->SR & TIM_SR_UIF) != 0 )
    TIM4->SR &= ~TIM_SR_UIF;

  return;
}
```

Example 21-8. The interrupt service handler of timer 4

Figure 7. *Example code from the Zhu textbook*

However, he quickly ran into an issue. The NVIC_EnableIRQ() function requires an argument in the form TIMx_IRQn. However, TIM1_IRQn does not exist, and the code would not compile with this argument. Jon quickly found that TIM4_IRQn raises no errors, and so he opted to use timer 4 instead. This meant that he needed to modify the existing code to use timer 4 instead of timer 1. Fortunately, this was a relatively simple change. After creating and enabling the interrupt handler, Jon moved the necessary code to the handler.

When we tested the code with these changes, we found that we no longer had the high frequency signal, and we had a sine wave of roughly the right frequency. Though the sine wave was messier than we would like, this was considerable progress.

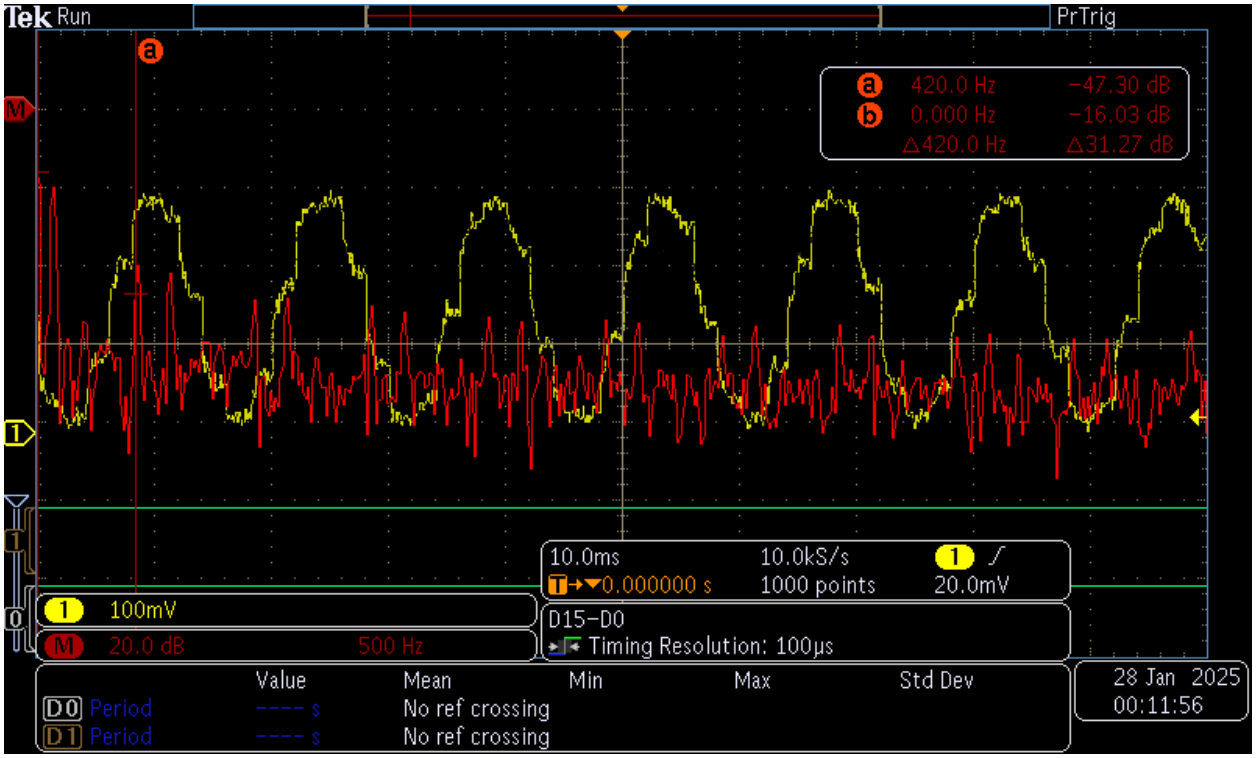


Figure 8. *Our code produced a sine wave, but not as clean as desired.*

Fortunately, this issue was easy to fix as we realized that the software trigger was still enabled in the task that handled the LED. After moving the software trigger and a little light debugging, we finally achieved what could only be described as a big, beautiful sine wave.

As a finishing touch, we modified the prescaler and auto-reload register to get as close to the desired 440 Hz frequency as possible. We eventually settled on a prescaler of 7 and an auto-reload register value of 18. With some quick arithmetic, we find that this should give us a frequency of about 1.6 kHz, but in practice this gave us a frequency of about 430 Hz. We have no idea why the actual frequency is so different from the expectation. Though 430 Hz is a little flat, the difference in pitch is barely noticeable. Finally, we modified our look-up table to reduce the amplitude of the wave and add a DC offset. This was done to prevent the op-amp from railing. Upon testing, the waveform both looked and sounded as desired.

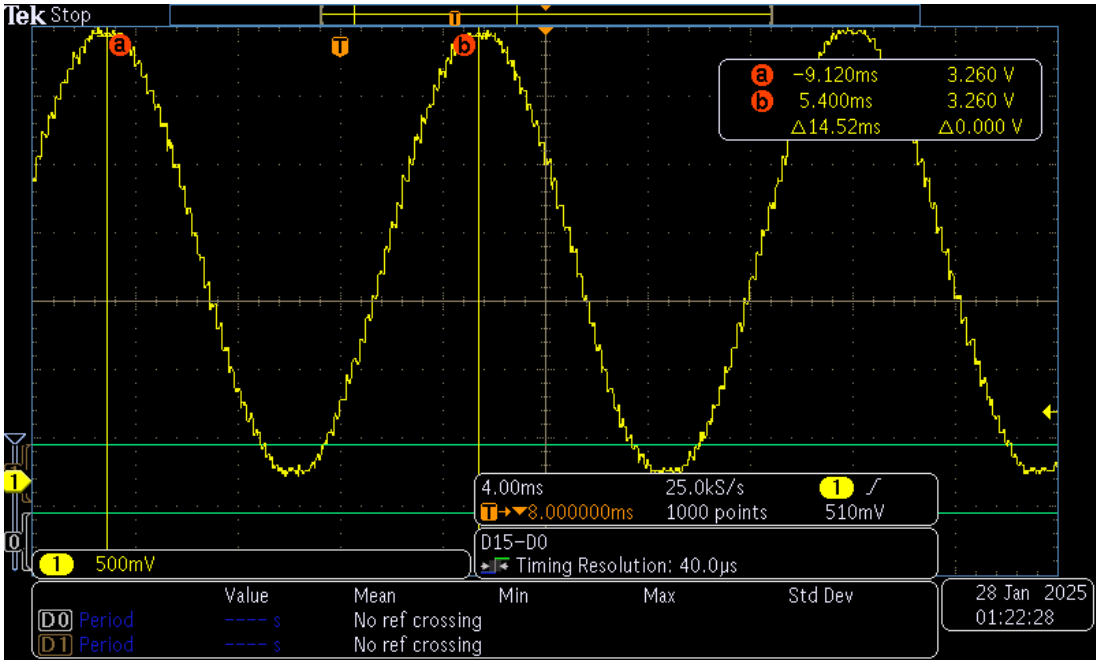


Figure 9. *Successful output of a 440 Hz 64-step unclipped sine wave.*

Conclusion.

During this lab, we have modified the code from the previous lab to play a roughly 440 Hz sine wave via an external speaker. We accomplished this by implementing both a DAC and a timer interrupt in software. We also created an external op-amp circuit to interface between the analog signal and the speaker.

main.c

```c
#include "stm32l476xx.h"
#include "setup.h"
#include "FreeRTOS.h"
#include "semphr.h"
#include "stdlib.h"
#include "portmacro.h"


void prvSetupHardware(void);
void readButton(void);
void writeLED(void);
void DAC_Channel1_Init(void);
void speaker(void);
uint32_t lookup_sine(uint32_t*, int);
void TIM4_Init(void);

static int LEDState = 0;
static int debounce = 0;

uint16_t degrees = 0;

uint32_t sine_table[360] = {0x400, 0x412, 0x424, 0x436, 0x447, 0x459, 0x46b, 0x47d,
0x48f, 0x4a0, 0x4b2, 0x4c3, 0x4d5, 0x4e6, 0x4f8, 0x509,
0x51a, 0x52b, 0x53c, 0x54d, 0x55e, 0x56f, 0x580, 0x590,
0x5a0, 0x5b1, 0x5c1, 0x5d1, 0x5e1, 0x5f0, 0x600, 0x60f,
0x61f, 0x62e, 0x63d, 0x64b, 0x65a, 0x668, 0x676, 0x684,
0x692, 0x6a0, 0x6ad, 0x6ba, 0x6c7, 0x6d4, 0x6e1, 0x6ed,
0x6f9, 0x705, 0x710, 0x71c, 0x727, 0x732, 0x73c, 0x747,
0x751, 0x75b, 0x764, 0x76e, 0x777, 0x780, 0x788, 0x790,
0x798, 0x7a0, 0x7a7, 0x7af, 0x7b5, 0x7bc, 0x7c2, 0x7c8,
0x7ce, 0x7d3, 0x7d8, 0x7dd, 0x7e2, 0x7e6, 0x7ea, 0x7ed,
0x7f0, 0x7f3, 0x7f6, 0x7f8, 0x7fa, 0x7fc, 0x7fe, 0x7ff,
0x7ff, 0x800, 0x800, 0x800, 0x7ff, 0x7ff, 0x7fe, 0x7fc,
0x7fa, 0x7f8, 0x7f6, 0x7f3, 0x7f0, 0x7ed, 0x7ea, 0x7e6,
0x7e2, 0x7dd, 0x7d8, 0x7d3, 0x7ce, 0x7c8, 0x7c2, 0x7bc,
0x7b5, 0x7af, 0x7a7, 0x7a0, 0x798, 0x790, 0x788, 0x780,
0x777, 0x76e, 0x764, 0x75b, 0x751, 0x747, 0x73c, 0x732,
0x727, 0x71c, 0x710, 0x705, 0x6f9, 0x6ed, 0x6e1, 0x6d4,
0x6c7, 0x6ba, 0x6ad, 0x6a0, 0x692, 0x684, 0x676, 0x668,
0x65a, 0x64b, 0x63d, 0x62e, 0x61f, 0x60f, 0x600, 0x5f0,
0x5e1, 0x5d1, 0x5c1, 0x5b1, 0x5a0, 0x590, 0x580, 0x56f,
0x55e, 0x54d, 0x53c, 0x52b, 0x51a, 0x509, 0x4f8, 0x4e6,
0x4d5, 0x4c3, 0x4b2, 0x4a0, 0x48f, 0x47d, 0x46b, 0x459,
0x447, 0x436, 0x424, 0x412, 0x400, 0x3ee, 0x3dc, 0x3ca,
0x3b9, 0x3a7, 0x395, 0x383, 0x371, 0x360, 0x34e, 0x33d,
0x32b, 0x31a, 0x308, 0x2f7, 0x2e6, 0x2d5, 0x2c4, 0x2b3,
0x2a2, 0x291, 0x280, 0x270, 0x260, 0x24f, 0x23f, 0x22f,
0x21f, 0x210, 0x200, 0x1f1, 0x1e1, 0x1d2, 0x1c3, 0x1b5,
0x1a6, 0x198, 0x18a, 0x17c, 0x16e, 0x160, 0x153, 0x146,
0x139, 0x12c, 0x11f, 0x113, 0x107, 0xfb, 0xf0, 0xe4,
0xd9, 0xce, 0xc4, 0xb9, 0xaf, 0xa5, 0x9c, 0x92,
0x89, 0x80, 0x78, 0x70, 0x68, 0x60, 0x59, 0x51,
0x4b, 0x44, 0x3e, 0x38, 0x32, 0x2d, 0x28, 0x23,
0x1e, 0x1a, 0x16, 0x13, 0x10, 0xd, 0xa, 0x8,
0x6, 0x4, 0x2, 0x1, 0x1, 0x0, 0x0, 0x0,
0x1, 0x1, 0x2, 0x4, 0x6, 0x8, 0xa, 0xd,
0x10, 0x13, 0x16, 0x1a, 0x1e, 0x23, 0x28, 0x2d,
0x32, 0x38, 0x3e, 0x44, 0x4b, 0x51, 0x59, 0x60,
0x68, 0x70, 0x78, 0x80, 0x89, 0x92, 0x9c, 0xa5,
```

```
0xaf, 0xb9, 0xc4, 0xce, 0xd9, 0xe4, 0xf0, 0xfb,
0x107, 0x113, 0x11f, 0x12c, 0x139, 0x146, 0x153, 0x160,
0x16e, 0x17c, 0x18a, 0x198, 0x1a6, 0x1b5, 0x1c3, 0x1d2,
0x1e1, 0x1f1, 0x200, 0x210, 0x21f, 0x22f, 0x23f, 0x24f,
0x260, 0x270, 0x280, 0x291, 0x2a2, 0x2b3, 0x2c4, 0x2d5,
0x2e6, 0x2f7, 0x308, 0x31a, 0x32b, 0x33d, 0x34e, 0x360,
0x371, 0x383, 0x395, 0x3a7, 0x3b9, 0x3ca, 0x3dc, 0x3ee};



//Turns LED on and off based on LEDState
void writeLED(void){

        while(1){

                if(digitalRead(GPIOC, 13)){
                        if(LEDState != 0){
                                //Turn LED on
                                digitalWrite(GPIOA, 5, 1);
                        }
                        else{
                                //Turn LED off
                                digitalWrite(GPIOA, 5, 0);
                        }

                }
        }
}

//Interrupt Handler
void TIM4_IRQHandler(){

        //If Sound is supposed to be on
                if((TIM4->SR & TIM_SR_CC1IF)!=0){

                        //Send Digital Value to DAC
                        DAC->DHR12R1 = sine_table[degrees]+75;

                        if(LEDState != 0){
                                //Start Software Trigger
                                DAC->SWTRIGR |= DAC_SWTRIGR_SWTRIG1;
                        }

                        //Increment sine wave
                        degrees+=6;

                        //Ensure degrees is always between 0 and 360
                        if(degrees >=360){
                                degrees-=360;
                        }

                        //Clear CC1IF Flag
                        TIM4->SR &= ~TIM_SR_CC1IF;
                }

                //If update has occured
                if((TIM4->SR & TIM_SR_UIF)!=0){
                        //Clear Update Inturrupt Flag
                        TIM4->SR &= ~TIM_SR_UIF;
                }
```

```
                return;
}

//Toggles LEDState whenever button is pressed
void readButton(void){
        while(1){
                if (!digitalRead(GPIOC, 13) && !debounce){

                        if(LEDState == 0){
                                LEDState = 1;
                        }
                        else{
                                LEDState = 0;
                        }

                        debounce = 1;
                }

                if(digitalRead(GPIOC, 13)){
                        //delay_ms(10);
                        debounce = 0;
                }
        }
}

void prvSetupHardware(void)
{
        //Enable HSI16 clk
        RCC->CR |= RCC_CR_HSION;

        //Loop Until Clock is Ready
        while(!(RCC->CR & RCC_CR_HSIRDY));

        //Enable DAC Channel 1
        DAC_Channel1_Init();

        //Enable Timer 4
        TIM4_Init();

        //Enable GPIO A&C Clocks
        RCC->AHB2ENR |= RCC_AHB2ENR_GPIOAEN;
        RCC->AHB2ENR |= RCC_AHB2ENR_GPIOCEN;

        //Enable LED (Pin A5)
        pinMode(GPIOA, 5, 1);
        setOutputType(GPIOA, 5, 0);

        //Enable Push Button (Pin C13)
        pinMode(GPIOC, 13, 0);

}

//DAC Channel 1 Initialization; DAC_OUT1 = PA4
void DAC_Channel1_Init(void){

        //DAC Clock Enable
        RCC->APB1ENR1 |= RCC_APB1ENR1_DAC1EN;

        //Disable DAC
```

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

        //Set DAC Mode for channel 1
        DAC->MCR &= ~(7U);

        //Enable Trigger
        DAC->CR |= DAC_CR_TEN1;

        //Select Software Trigger
        DAC->CR |= DAC_CR_TSEL1;

        //Enable Channel 1
        DAC->CR |= DAC_CR_EN1;

        //Enable GPIO A Clock
        RCC->AHB2ENR |= RCC_AHB2ENR_GPIOAEN;

        //Set GPIO Pin A4 to Analog
        GPIOA->MODER |= 3U<<(2*4);


}



//Timer 4 Initialization
void TIM4_Init(void){

        //Enable Timer 4 clock
        RCC->APB1ENR1 |= RCC_APB1ENR1_TIM4EN;

        //Set Counter to Count Up
        TIM4->CR1 &= ~TIM_CR1_DIR;

        //Prescalar
        TIM4->PSC = 7;          //May need to change this

        //Auto Reload
        TIM4->ARR = 18; //Will need to change this

        //Clear output comare mode
        TIM4->CCMR1 &= ~TIM_CCMR1_OC1M;


        //Enable Interrupt
        NVIC_EnableIRQ(TIM4_IRQn);

        //Select PWM Mode 1
        TIM4->CCMR1 |= TIM_CCMR1_OC1M_1 | TIM_CCMR1_OC1M_2;

        //Output 1 Preload Enable
        TIM4->CCMR1 |= TIM_CCMR1_OC1PE;

        //Set output to active high
        TIM4->CCER &= ~TIM_CCER_CC1NP;

        //Enable complementary output of channel 1
        TIM4->CCER |= TIM_CCER_CC1NE;

        //Enable Main Output
```

```
        TIM4->BDTR |= TIM_BDTR_MOE;

        //Output compare registier for channel 1
        TIM4->CCR1 = 500; //Change this in the speaker function

        //Enable Counter
        TIM4->CR1 |= TIM_CR1_CEN;

        //Enable Capture Compare 1 interrupt
        TIM4->DIER |= (1U<<1);

}



int main(void)
{
        //Setup Hardware
        prvSetupHardware();

        //Update Clock
        SystemCoreClockUpdate();

        BaseType_t xReturned;
        TaskHandle_t xHandle = NULL;


        //Create Tasks
        xReturned = xTaskCreate(readButton, "Button", 64, NULL, 3, &xHandle);
        xReturned = xTaskCreate(writeLED, "LED", 64, NULL, 3, &xHandle);

        //Black Magic
        vTaskStartScheduler();

        for(;;);

        return 0;
}
```

setup.c

```c
#include "setup.h"
//#include "stm32l476xx.h"

#define OUTPUT 1
#define INPUT 0

#define PUSHPULL 0
#define OPENDRAIN 1

#define NONE 0
#define PULLUP 1
#define PULLDOWN 2

#define HIGH 1
#define LOW 0

void delay_ms(unsigned int ms){
        volatile unsigned i, j;

        for(i = 0; i<ms; i++){
                for(j = 0; j<800; j++){
                }
        }
}

void pinMode(GPIO_TypeDef *port,unsigned int pin,unsigned int mode) {
    //port->MODER = ..................
        //int mask << pin
        //moder and mask
        //moder or mask
        unsigned int mask = (0x1<<(pin*2)) + (0x1<<((pin*2)+1));
        port->MODER &= ~mask;
        if(mode == OUTPUT){
                port->MODER |= 0x1<<(pin*2);
        }

}


void setOutputType(GPIO_TypeDef *port,unsigned int pin,unsigned int type) {
    //port->OTYPER = ..................
        if(type == PUSHPULL){
                unsigned int mask = 0xFFFFFFFE;    // 4294967294;
                mask = mask << pin;
                mask = mask + ((1<<pin)-1);
                port->OTYPER &= mask;
        }
        else if(type == OPENDRAIN){
                unsigned int mask = 1;
                mask = mask << pin;
                port->OTYPER |= mask;
        }
}

void setPullUpDown(GPIO_TypeDef *port, unsigned int pin, unsigned int pupd) {
    //port->PUPDR = ..................
        unsigned int mask = (0x1<<(pin*2)) + (0x1<<((pin*2)+1));
        port->PUPDR &= ~mask;
        if(pupd == 1){
```

```
                port->PUPDR |= 0x1<<(pin*2);
        }
        if(pupd == 2){
                port->PUPDR |= 0x1<<((pin*2)+1);
        }
//        unsigned int fmask = 0xFFFFFFFC + pupd;
//        unsigned int omask = pupd;
//        fmask = fmask << pin;
//        fmask = fmask << pin;                //henry thinks he's funny
//        omask = omask << pin;
//        omask = omask << pin;
//        unsigned int one = 1;
//        fmask = fmask + (((one<<pin)<<pin) - one);
//        port->OTYPER &= fmask;
//        port->OTYPER |= omask;
}

void digitalWrite(GPIO_TypeDef *port, unsigned int pin, unsigned int value) {
    //port->ODR = ..................
        unsigned int mask = 0x1<<pin;
        port->ODR &= ~mask;
        if(value == 1){
                port-> ODR |= mask;
        }
}

unsigned int digitalRead(GPIO_TypeDef *port,unsigned int pin) {
    //return ..................
        unsigned int mask = port->IDR;
        mask = mask >> pin;
        mask = mask %2;
        return mask;
}
```

setup.h

```
#ifndef SETUP_H
#define SETUP_H

#include "stm32l476xx.h"




#define OUTPUT 1
#define INPUT 0

#define PUSHPULL 0
#define OPENDRAIN 1

#define NONE 0
#define PULLUP 1
#define PULLDOWN 2

void pinMode(GPIO_TypeDef *port,unsigned int pin,unsigned int mode);

void setOutputType(GPIO_TypeDef *port,unsigned int pin,unsigned int type);

void setPullUpDown(GPIO_TypeDef *port,unsigned int pin,unsigned int pupd);

void digitalWrite(GPIO_TypeDef *port,unsigned int pin,unsigned int value);

unsigned int digitalRead(GPIO_TypeDef *port,unsigned int pin);

void delay_ms(unsigned int ms);

#endif
```