ECE 355 Final Project Report

Felipe Goldbach

Nishchint Dhawan

Table of Contents

1.0 Problem Description

This project required us to solve two main challenges, generate a pulse width modulated (PWM) signal using the NE555 timer, then calculate and display onto the LCD(on STM32f0 Discovery board) the instantaneous values for the frequency of the waveform respective to the potentiometer. The instantaneous resistance of the potentiometer must also be determined and displayed properly.

This project includes 5 main components: a PWM signal generator, Analog to Digital converter (ADC), Digital to Analog converter (DAC), SPI data transfer, and LCD display. These components are integrated through the software written in the Eclipse environment.

2.0 Specifications

The STM32f0 discovery board is interfaced with the MCU project board student learning kit(PBMCUSLK). The STM32f0 contains a Cortex-M0 microprocessor with an operating frequency of 48 MHz. The PBMCUSLK has a breadboard where circuits can be wired and integrated with other components on the board. For this lab the value of d will be equal to 12 as that is the resolution of the ADC and the max reference voltage for the board is 5V. The PBMCUSLK has an on-board potentiometer which is used as an external 5k standalone potentiometer. Resistance is not calculated by measuring the current, but instead is measured using the ratio in equation (1) below with the ADC Value.

$$Resitance = \frac{(ADC->DR) * 5000}{2^d - 1} \quad (1)$$

An NE555 timer IC generates pulses with frequencies depending on the capacitance of C_2 and resistances of R_1 and R_2. The frequency can be calculated using equation (2) below.

$$Frequency = \frac{1}{ln(2) * C_2 * (R_1 * 2R_2)} \quad (2)$$

The 555 timer is implemented using the 4N35 optocoupler as the value for R_2 in parallel with a variable resistor. Refer to Figure 4 for more information on the wiring and location of the capacitor and resistors.

3.0 Solution

We initiated the implementation of our project by following the sample circuit given in the

project orientation lab. These connections were made using a breadboard and stm32f0 microcontroller. Using the microcontroller, we were able to implement the software in our project through the use of the code written in the first lab to calculate the frequency of a pulse train signal, then adding the necessary methods for the 5 main components.

3.1 Design

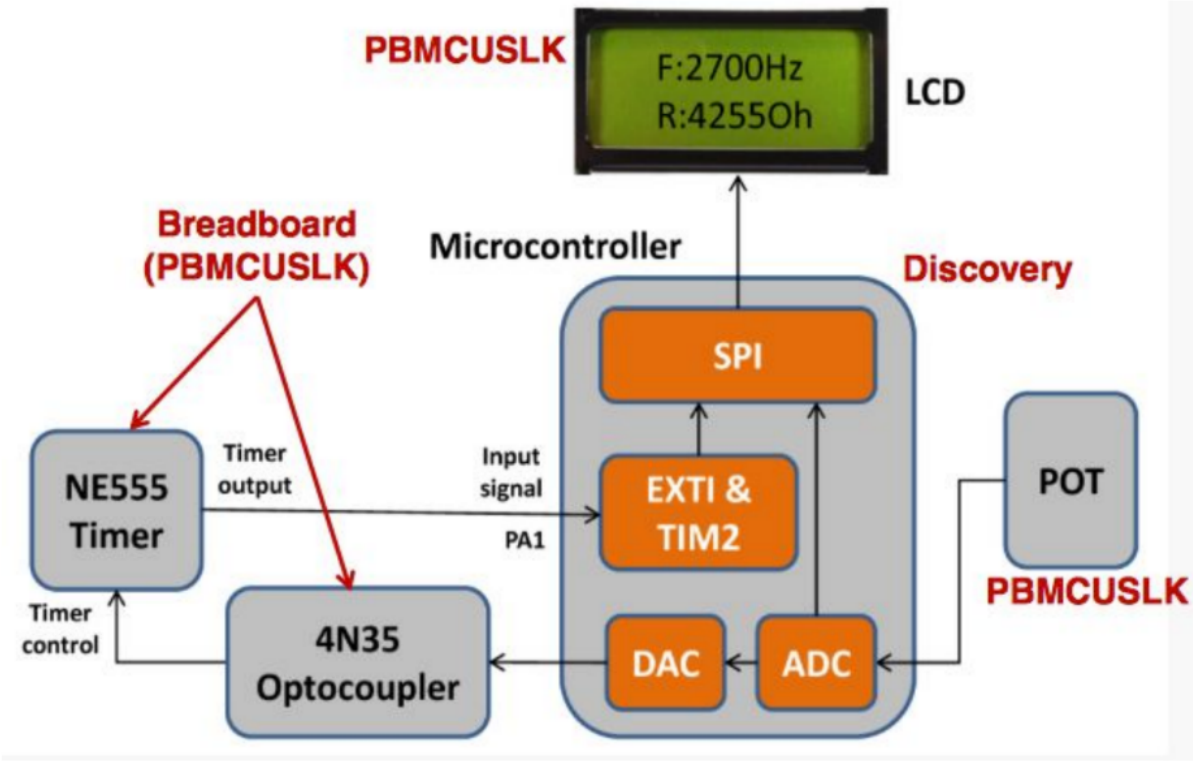

Figure 1: System Component Outline [1]

3.1.1 EXTI & TIM2

The lab 1 part 2 code focused on measuring the square wave from the function generator. The final project code is built upon this code. In particular, the code involved initializing both external interrupts(EXTI) and timer TIM2. For the external interrupt, EXTI was mapped to channel 1 and set to trigger on pin PA1 as input. The GPIOA mode register for pin 1 was not set, because the input mode was, by default, the reset value. A value of 0x0000, 0XFFFFFFFF and was set in TIM2->PSC and TIM2->ARR as an auto-reload delay. Therefore after 90 seconds (calculated from the max count of 2^{32} divided by the system core clock 48 MHz), the timer will read the

value of 0 seconds (0x0000). Once the timer does hit 90 seconds, however, TIM2 will generate an interrupt, then TIME2_IRQHandler, will check the TIM2 status register(TIM2->SR) to see if the interrupt flag was set. If set, the handler will clear it to restart the timer.

When PA detects input, the EXTI0_1_IRQHandler is asserted. This is used to start and stop the counter register TIM2_CNT to measure the clock cycles passed. The rising edge is detected by using a flag called “edge”. When this interrupt handler is called, we check if the interrupt pending flag is set and we then check if the edge flag is 1 or not. The flag is 1 if the edge is riding and 0 if falling. When the edge is 1, we clear the TIM2_CNT register to start the counter and set the edge to 0. When the falling edge is detected, the counter value is read and stored in a local variable. The count is in number of clock cycles. This value is divided by the frequency of the system clock (clock cycles per second) to find the time taken in seconds to complete the clock cycles. The inverse of the seconds gives us the frequency value which is set in a global variable. Furthermore, during this, interrupts need to be masked so that another interrupt doesn’t happen during the current one. The main function can access the value of frequency and send it to the display.

3.1.2 Analog to Digital converter (ADC)

It is used to convert the analog signal from the potentiometre to a digital signal. It is located on the STM32F0 discovery board. In the myADC_Init function, the key features are that it was set to continuous mode, it was mapped to channel 2 (mapped to pin PA2 for analog mode), and the sampling rate was set to a constant rate of 7(the maximum value). A constant rate of 7 was used instead of the suggested 239.5 cycles as it seemed to yield a better resolution. The GPIOA mode register was used to set PA2 to analog, by setting all bits to 1, because this pin is receiving an analog input. A while loop was implemented that continues as long as the ADC is not ready, meaning the first bit of ADC1->ISR is 1, this only needs to be done in the initialization. During implementation, the end of conversion (EOC) flag is checked so values aren’t sent while the ADC is converting.
The ADC was 12 bit which means it will output values between 0 and 2^{12}-1, or from 0 to 4095. A maximum of 5V was input into pin PA2 means that the resolution (equation (1)) for the ADC was 5V/ 4095 = 1.22mV per bit change. Therefore the minimum voltage change has to be 1.22mV for the ADC to change values.

3.2.3 Digital to Analog Converter (DAC)

The selected channel for the DAC was set to channel 1 in DAC->CR. The channel was mapped to pin PA4 as analog mode, by setting the GPIOA mode register for PA4 to 11. Enabling channel 1 for the output of the DAC automatically maps to pin PA4, but it is a good idea to ensure it is in analog mode prior to that. The last thing of importance for the DAC is that the SWTRIGR register was set for software triggering.

3.1.4 Serial Peripheral Interface (SPI)

Latch clock (LCK), shift clock (SCK) and the master output slave input (MOSI) are all mapped to GPIOB and need to be sent by the SPI. Then, they are received by the shift registers that is connected to the LCD. PB4 (LCK) is set to output mode, as it is essential to manually control the LCK signal when sending data with the SPI. Both PB3 (SCK) and PB5 (MOSI) are set to their alternate function modes, which utilize the SPI. These pins have to be connected to the corresponding J5 MCU pins on the breadboard, which were given on the “tips” section of the lab website. On the J5 connectors, LCK is connected to M25, SCK is connected to M21, and MOSI is connected to M17. The SPI will then be able to send data through the MOSI and latch data through the shift register.

3.1.5 LCD Display

The SPI was used to send the data to the LCD. the SPI library was enabled in the src directory of the project and was excluded from the build. The LCD will need to be sent an 8 bit word in two steps (4 bits at a time), regardless of whether the LCD is in 8 bit or 4 bit mode. The least-significant bits in 8-bit mode don't get used because the most significant 2 bits are used for the register select (RS) and enable (EN) bits, and the remaining two bits are don't cares. The next 4 bits is half of the instruction or data to be sent. Therefore the higher order bits with RS and EN need to be present in both the sends. When RS is set to 0, the LCD receives a command and when RS is 1, the LCD receives data to be displayed. .

The LCD will only update an instruction when EN becomes 1. Therefore each 4 bit nibble of data will need to be sent 3 times (EN bit goes from 0 1 0), and to send and 8 bit word, the data needs to be sent 6 times; 3 times for each nibble of data. To simplify this process, functions were created so that all that needs to be done is input the 8 bit word and the functions will handle separating high and low order bits, appending the RS and EN bits, and sending the data 3 times for the enable pulse. Referencing the code in section 6.0, the SendCommand function sends a command to the LCD and the SendLCD function sends the ASCII data to be displayed on the LCD. Both these functions separate the 8 bit word into two low order 4 bit nibbles, then appends the RS and EN bits to the higher order. There are two functions because to send a command (RS=0) via the SendCommand function, the high order bits, which include RS and EN, must follow a pattern of 0X, 8X, 0X where X is the nibble of data to be sent and the EN pulse is represented by high order bits changing for 0-8-0. Similarly, data to be displayed (RS=1) to the LCD is sent with the SendLCD function which must send a pattern of 4-C-4. A devised delay function, which used a while loop based on the input size, was used to simulate delays for the LCD. Particular delays that were needed were for the switching of the LCK signal and every stage of the LCD initialization process. The worst case execution times for each LCD process can be seen in the LCD reference manual found on the web page[1].

3.1.6 NE555 Timer

The NE555 timer was connects to the optocoupler in the circuit. However, to avoid future problems, before connecting to the optocoupler, the NE555 timer was wired and tested. The frequency is reliant on the resistance of R_1 and R_2, and the capacitance of C_1, as per equation 2. The values used in the project were: a 5V high signal was used for V_{cc}, R_1= R_2 = 1k, and the capacitor used was a 0.01 microfarad capacitor. The diagram for the NE555 timer can be seen in Figure 4 and its wiring with the optocoupler can be seen in Figure 5.

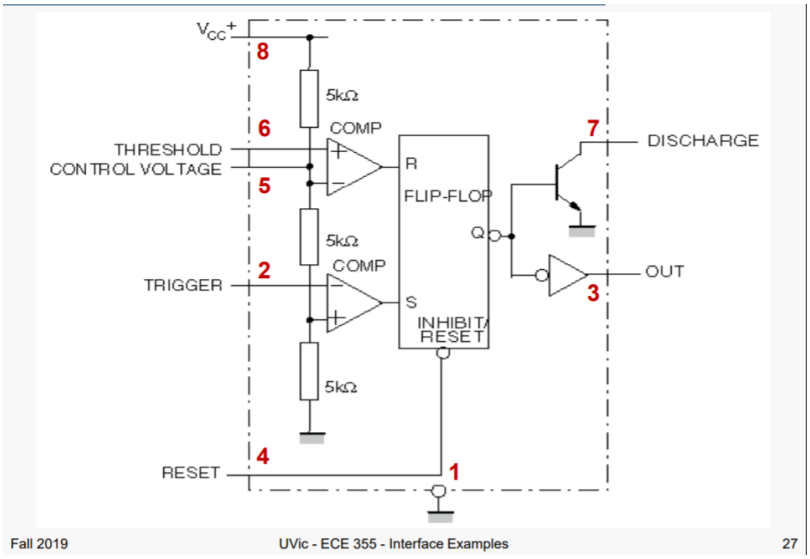

Figure 4: The NE555 timer [2]

3.1.7 4N35 Optocoupler

The 4N35 optocoupler effectively isolates the two parts of the circuit from each other. Depending on the amount of current received across pin 1-2. This comes from the discovery board PA4 and is the output of the DAC. This turns on an internal infrared emitter. The intensity of light produced is related to the amount of current through pin 1-2. The infrared collector between pin 5-4 will produce a corresponding resistance value depending on how intense the light is. This effectively gives us a variable resistor that we can connect in parallel across a 20k potentiometer as our R_2. The wiring of the optocoupler can be seen in Figure 5.

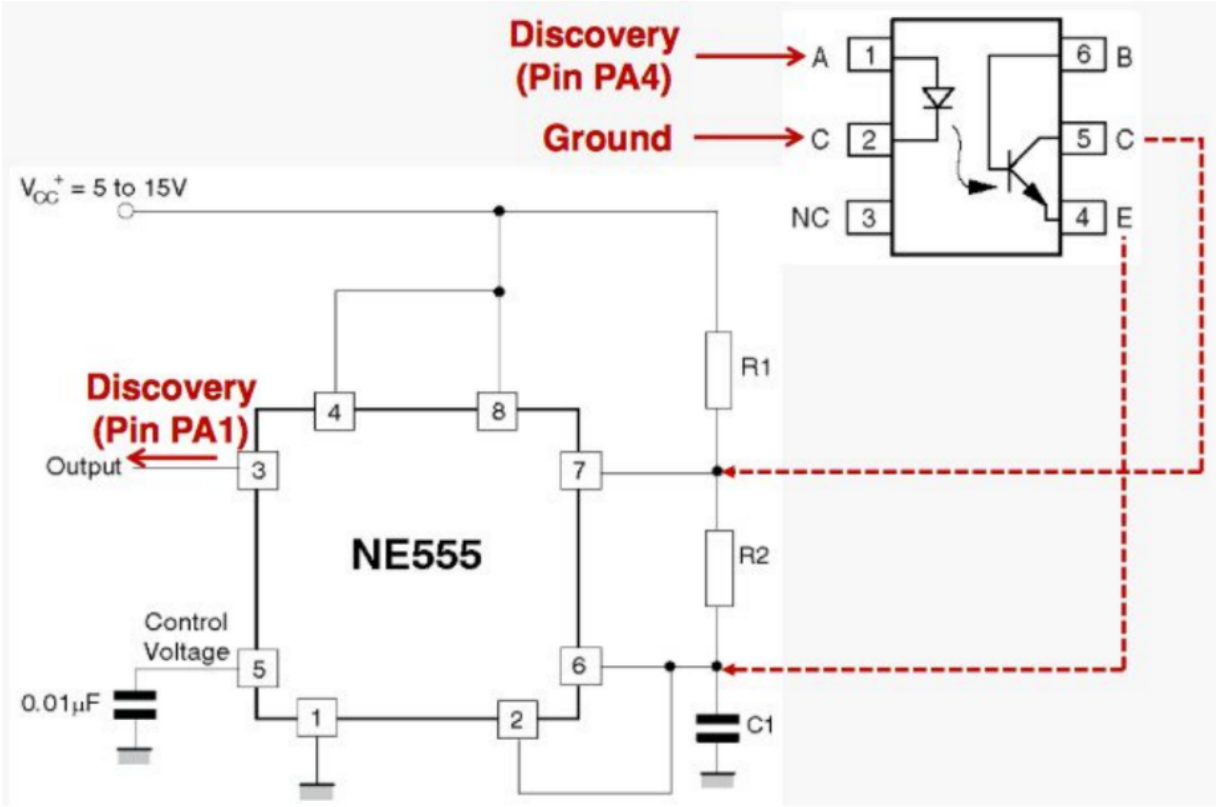

Figure 5: Reference Wiring Diagram [2]

After implementing each of the elements above and testing until the desired component was functioning, combining them together was not difficult. The next section outlines our test methodology.

3.2 Testing

The first step was to test the ADC and DAC. The ADC was tested by printing the contents of ADC1->DR (ADC data register) to the console. The ADC is 12 bit, therefore it has values ranging from 0 to 4095. In the DAC, the ADC value stored in ADC1->DR is used to convert to a voltage value. Since the DAC was configured with a software trigger, the ADC value is stored in the which is the data holding register for the DAC in right aligned mode (DAC->DHR12R1). The contents of this register are then buffered to a read only data output register DAC->DOR, which outputs the converted value to DAC-OUT1, connected to pin PA4. An oscilloscope was used to test the SPI by observing the waveforms from the GPIOB pins. it was observed that the MOSI is low for 3 cycles of LCK because when sending data to the LCD via the SPI data needs to be sent 3 times.

The procedure was to first get the LCD to clear properly which was all done through initialization. Initialization only required the EnableSend function as characters were not going to be written to the LCD during this stage. The next step was to write ASCII characters to the LCD and fill both rows with full control of which characters are written to the LCD. Proper initialization was required with the ASCII code for the described character to be displayed. Since a devised delay function was created the delays were sufficient as long as they were all higher than the worst case execution time for each LCD action.

The resistance from the potentiometer was then directly fed to the LCD using equation (1) to calculate the resistance to display. Next, the frequency of the 555 timer was confirmed by varying the output frequency with a potentiometer in place of R_2 in equation (2). Once full control of the resistance and frequency was confirmed, the optocoupler was then connected to the 555 timer as mentioned previously. The optocoupler input is connected with the output of the DAC at pin PA4 and the 555 timer output is then connected to pin PA1, which measures the frequency of the timer. The final step was to take the two digital values and isolate each number to be displayed to the LCD. This was done by dividing out any higher order parts and using the modulus operator to get a single digit.

Each isolated digit was then converted to ASCII by adding a “0” to the integer values. Doing this makes the compiler think the result is a character and interprets it as ASCII. Functions FrequencyOutput and ResistanceOutput in section 6.0 accomplish the isolation of digits and conversion to ASCII.

3.3 Results

When the potentiometer voltage is changed from closed to fully open, the resistance increases from 0 Ohms to 5000 Ohms and the frequency reduces to about 0.7 times. The frequency value does not change much. This is because we are using a 0.01μF capacitor. These values are consistent with the expected values obtained from equations 1 and 2.

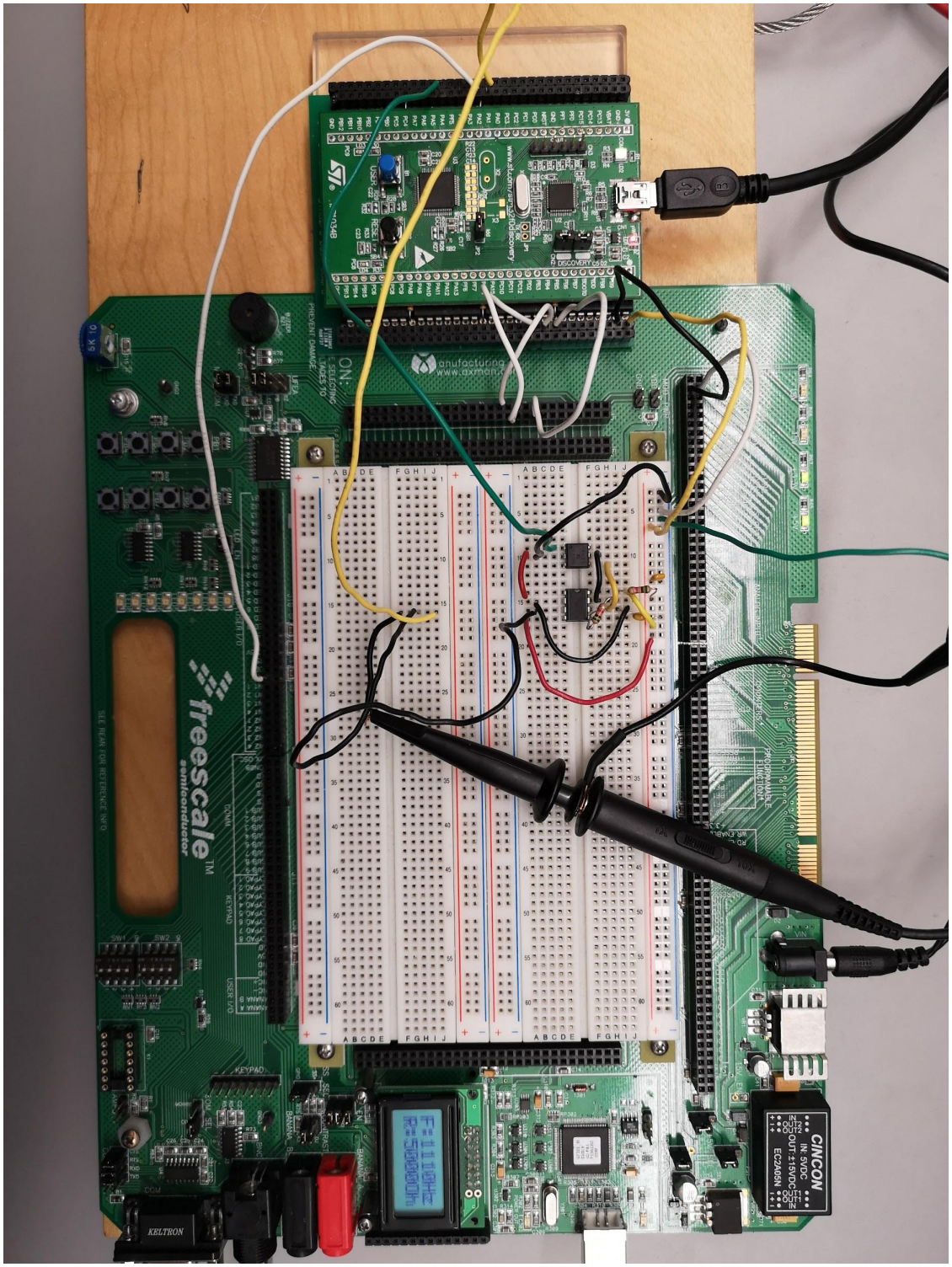

Figure 5: Fully Connected System

Since a ceramic disk capacitor in the µF range was used with little changes to frequency over a large change in resistance, a box capacitor in the nF range would yield a larger range of

frequencies over changes in resistance due to the capacitance being inversely proportional to the frequency range. Figures 6, 7, and 8 shown below represent the LCD display at minimum and maximum resistance, and ADC waveform at maximum resistance.

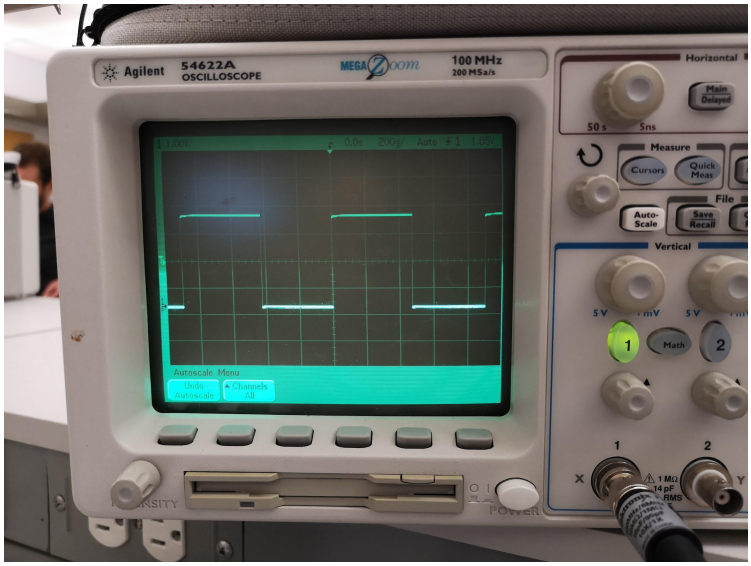

Figure 6: ADC Waveform at Maximum Resistance

Figure 7: Frequency at Maximum Resistance

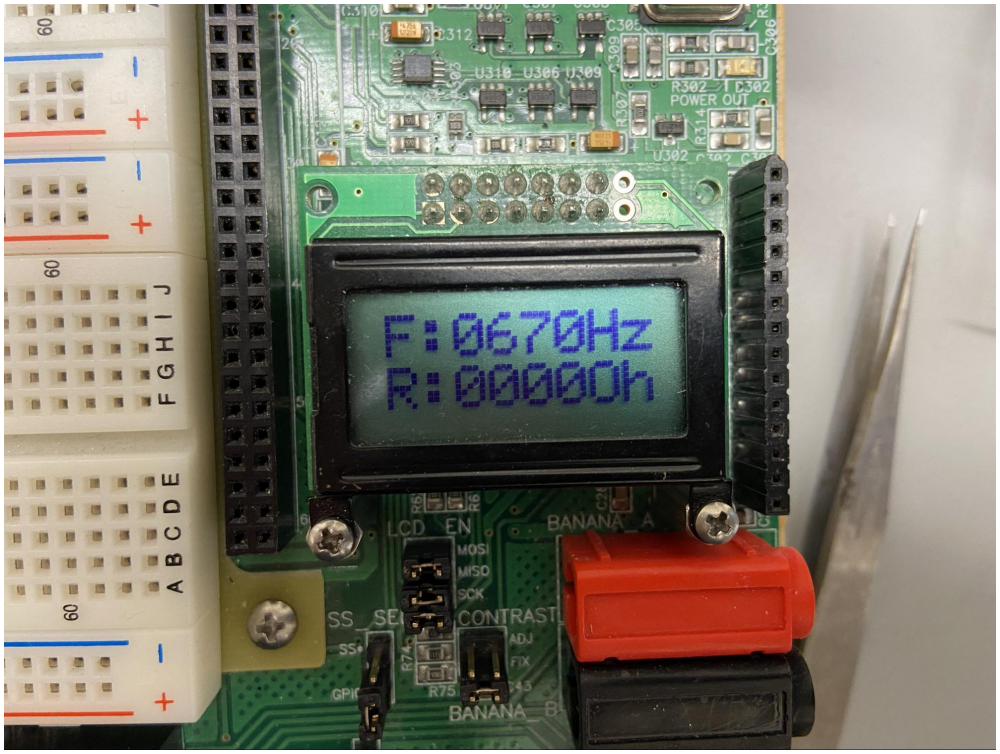

Figure 8: Frequency at Minimum Resistance

4.0 Discussion

The divide and conquer of this project proved to be very efficient and productive. However, the journey taken to complete this project did not go without its share of challenges. One observation is that working at different lab stations, the DAC varied from station to station. One test station used produced a DAC voltage of 3.86V and another test station produced a DAC voltage of 2.1V. However, this was easily overcome by taking full claim of our station. This variation could have been due to parasitic capacitance or voltage leakages. This did not hinder the results much, however, it could have been more accurate if some calibration was implemented by software, if it isn't already done with hardware.

A major road-block of the project was the implementation of the LCD display. This component required a few layers of troubleshooting. Firstly, we were wrongly receiving values of 0 for the potentiometer resistance. We found the issue causing this was divisions by integers in our method, so we simply converted our values to floats to assure the proper values were being displayed. Secondly, our values were not updating properly, so we had to implement a delay in the LCD display to account for the slow refreshing screen. The value of this delay was originally based off the value told in the labs, but we had to experiment with some trial and error to get the precise value for the delay. This project proved to be a successful way for learning the implementation of microprocessor programming and embedded systems. Even

though there were many challenges along the way, we managed to be successful in the completion and implementation of the project. The experimental values were consistent with the actual values measured by an oscilloscope. Although the frequency range of the system was quite miniscule, a PWM circuit modification could have increased the frequency range due to the inverse proportionality of capacitance to frequency range. Overall, the project was successful in both producing the proper values and allowing us to learn about the world of microprocessor based systems.

5.0 Code Design and Documentation

```
#include <stdio.h>
#include "diag/Trace.h"
#include "cmsis/cmsis_device.h"
// ----------------------------------------------------------------------------
//
// STM32F0 empty sample (trace via $(trace)).
//
// Trace support is enabled by adding the TRACE macro definition.
// By default the trace messages are forwarded to the $(trace) output,
// but can be rerouted to any device or completely suppressed, by
// changing the definitions required in system/src/diag/trace_impl.c
// (currently OS_USE_TRACE_ITM,
OS_USE_TRACE_SEMIHOSTING_DEBUG/_STDOUT).
//
// ----- main() ---------------------------------------------------------------
// Sample pragmas to cope with warnings. Please note the related line at
// the end of this function, used to pop the compiler diagnostics status.
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wunused-parameter"
#pragma GCC diagnostic ignored "-Wmissing-declarations"
/* Clock prescaler for TIM2 timer: no prescaling */
#define myTIM2_PRESCALER ((uint16_t)0x0000)
/* Maximum possible setting for overflow */
#define myTIM2_PERIOD ((uint32_t)0xFFFFFFFF)
SPI_InitTypeDef SPI_InitStructInfo;
SPI_InitTypeDef* SPI_InitStruct = &SPI_InitStructInfo;
void myGPIOA_Init(void);
void myTIM2_Init(void);
void myEXTI_Init(void);
void myADC_Init(void);
void myDAC_Init(void);
```

```
void mySPI_Init(void);
void SPISendData(uint8_t);
void CreateDelay(volatile long);
void LCD_Init(void);
void EnableSend(uint8_t);
void EnableCommand(uint8_t);
void SendToLCD(uint8_t);
void EnableData(uint8_t);
void ResitanceOutput(int);
void FrequencyOutput(int);
// Your global variables...
//volatile unit32-t timer pulse;
unsigned int edge = 1;
unsigned int Resistance;
int thousand = 0;
int hundred = 0;
int tens = 0;
int ones = 0;
unsigned int frequency;
unsigned int counter;

int main(int argc, char* argv[])
{
	//trace_printf("This is Part 2 of Introductory Lab...\n");
	trace_printf("This is Part 2 of Introductory Lab...\n");
	trace_printf("System clock: %u Hz\n", SystemCoreClock);
	myGPIOA_Init(); /* Initialize I/O port PA */
	myTIM2_Init(); /* Initialize timer TIM2 */
	myEXTI_Init(); /* Initialize EXTI */
	EXTI0_1_IRQHandler();
	myADC_Init();
	myDAC_Init();
	mySPI_Init();
	LCD_Init();
	while (1)
	{
		ADC1 -> CR = ADC_CR_ADSTART; //start of conversion
		while((ADC1 -> ISR & ADC_ISR_EOC)==0); // wait for end of
conversion
		DAC->DHR12R1 = ADC1 ->DR;
		ResitanceOutput((ADC1->DR)*5000/4095);
		FrequencyOutput(frequency);
		}
```

```
	return 0;
}

void ResitanceOutput(int Resistance) {
	//trace_printf("Resistance value: %u\n", Resistance);
	thousand = Resistance/1000;
	hundred = (Resistance/100) % 10;
	tens = (Resistance/10) % 10;
	ones = (Resistance) % 10;
	EnableSend(0xC0);
	SendToLCD(0x52);
	SendToLCD(0x3A);
	SendToLCD(thousand+'0');
	SendToLCD(hundred+'0');
	SendToLCD(tens+'0');
	SendToLCD(ones+'0');
	SendToLCD(0x4F);
	SendToLCD(0x68);
}

void FrequencyOutput (int Frequency) {
	//trace_printf("Frequency value: %u\n", Frequency);
	thousand = Frequency/1000;
	hundred = (Frequency/100) % 10;
	tens = (Frequency/10) % 10;
	ones = (Frequency) % 10;
	EnableSend(0x80);
	SendToLCD(0x46);
	SendToLCD(0x3A);
	SendToLCD(thousand+'0');
	SendToLCD(hundred+'0');
	SendToLCD(tens+'0');
	SendToLCD(ones+'0');
	SendToLCD(0x48);
	SendToLCD(0x7A);
}

void SPISendData(uint8_t Data) {
//Force LCK signal to 0
	GPIOB->BRR |= GPIO_BRR_BR_4;
	CreateDelay(5000);
	while((SPI1->SR & SPI_SR_BSY) !=0); //check if BSY is 0/SPI ready
	SPI_SendData8(SPI1,Data);
```

```
	while((SPI1->SR & SPI_SR_BSY) !=0); //check if BSY is 0/SPI ready
	//Force LCK signal to 1
	GPIOB->BSRR = GPIO_BSRR_BS_4;
	CreateDelay(5000);
}

void EnableCommand(uint8_t Word){
	uint8_t EN = Word | 0x80;
	SPISendData(Word);
	CreateDelay(500);
	SPISendData(EN);
	CreateDelay(500);
	SPISendData(Word);
	CreateDelay(500);
}

void EnableSend(uint8_t Word){
	uint8_t HighOrder = ((Word >> 4) & 0x0F);
	EnableCommand(HighOrder);
	uint8_t LowOrder = (Word & 0x0F);
	EnableCommand(LowOrder);
}

void EnableData(uint8_t Word){
	uint8_t H = Word | 0xC0;
	uint8_t L = Word | 0x40;
	SPISendData(L);
	CreateDelay(500);
	SPISendData(H);
	CreateDelay(500);
	SPISendData(L);
	CreateDelay(500);
}

void SendToLCD(uint8_t Word){
	uint8_t HighOrder = ((Word >> 4) & 0x0F);
	EnableData(HighOrder);
	uint8_t LowOrder = (Word & 0x0F);
	EnableData(LowOrder);
}

void CreateDelay(volatile long nCount){
	while (nCount!=0){
```

```
		nCount--; //decrement count value
	}
}

void LCD_Init() {
	CreateDelay(5000);
	for(int i = 0; i<3; i++){
		SPISendData(0x03);
		CreateDelay(5000);
		SPISendData(0x83);
		CreateDelay(5000);
		SPISendData(0x03);
		CreateDelay(5000);
	}
	SPISendData(0x02);
	CreateDelay(100);
	SPISendData(0x82);
	CreateDelay(100);
	SPISendData(0x02);
	CreateDelay(100);
	EnableSend(0x28); //DL = 0, N = 1, F = 0
	EnableSend(0x0c); //D = 1, C = 0, B = 0
	EnableSend(0x06); // I/D = 1, S = 0
	EnableSend(0x01); // Clear Display
	CreateDelay(5000);
}

void myADC_Init() {
	RCC -> APB2ENR |= RCC_APB2ENR_ADCEN; //ADC1 clock enable
	ADC1 -> CFGR1 |= ADC_CFGR1_CONT; //Continuous conversion
	ADC1->CHSELR |= ADC_CHSELR_CHSEL2; //Select Channel 2
	ADC1->SMPR |= ADC_SMPR_SMP; //SMP[2:0] bits (Sampling time
selection)
	ADC1->CR |= ADC_CR_ADEN; //ADC Enable Control
	while((ADC1->ISR & ADC_ISR_ADRDY)==0); //wait till ADC is ready
}

void myDAC_Init() {
	RCC->APB1ENR |= RCC_APB1ENR_DACEN; //DAC clock enable
	DAC->CR |= DAC_CR_EN1; //DAC Channel 1 enable
	DAC->CR |= DAC_CR_TSEL1;
	DAC->SWTRIGR |= DAC_SWTRIGR_SWTRIG1; //Channel 1 software
trigger
```

```
}

void mySPI_Init() {
	RCC->APB2ENR |= RCC_APB2ENR_SPI1EN; //SPI1 clock enable
	SPI_InitStruct->SPI_Direction = SPI_Direction_1Line_Tx;
	SPI_InitStruct->SPI_Mode = SPI_Mode_Master;
	SPI_InitStruct->SPI_DataSize = SPI_DataSize_8b;
	SPI_InitStruct->SPI_CPOL = SPI_CPOL_Low;
	SPI_InitStruct->SPI_CPHA = SPI_CPHA_1Edge;
	SPI_InitStruct->SPI_NSS = SPI_NSS_Soft;
	SPI_InitStruct->SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_2;
	SPI_InitStruct->SPI_FirstBit = SPI_FirstBit_MSB;
	SPI_InitStruct->SPI_CRCPolynomial = 7;
	SPI_Init(SPI1, SPI_InitStruct);
	SPI_Cmd(SPI1, ENABLE);
}

void myGPIOA_Init()
{
/* Enable clock for GPIOA peripheral */
RCC->AHBENR |= RCC_AHBENR_GPIOAEN;
RCC->AHBENR |= RCC_AHBENR_GPIOBEN;
GPIOB->AFR[2] = ((uint32_t)0x00000000);
/* Configure PA2, PA4 as Analog */
GPIOA->MODER |=(GPIO_MODER_MODER2 | GPIO_MODER_MODER4);
GPIOB->MODER |=(GPIO_MODER_MODER3_1 | GPIO_MODER_MODER4_0 |
GPIO_MODER_MODER5_1);
/* Ensure no pull-up/pull-down */
GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR1 | GPIO_PUPDR_PUPDR2 |
GPIO_PUPDR_PUPDR4);
GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR3 | GPIO_PUPDR_PUPDR4 |
GPIO_PUPDR_PUPDR5);
}

void myTIM2_Init()
{
	/* Enable clock for TIM2 peripheral */
	RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;
	/* Configure TIM2: buffer auto-reload, count up, stop on overflow,
	* enable update events, interrupt on overflow only */
	// Relevant register: TIM2->CR1
	TIM2->CR1 = ((uint16_t)0x008C);
	/* Set clock prescaler value */
```

```
        TIM2->PSC = myTIM2_PRESCALER;
        /* Set auto-reloaded CreateDelay */
        TIM2->ARR = myTIM2_PERIOD;
        /* Update timer registers */
        // Relevant register: TIM2->EGR
        TIM2->EGR = ((uint16_t)0x0001);
        /* Assign TIM2 interrupt priority = 0 in NVIC */
        // Relevant register: NVIC->IP[3], or use NVIC_SetPriority
        NVIC_SetPriority(TIM2_IRQn, 0);
        /* Enable TIM2 interrupts in NVIC */
        // Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
        NVIC_EnableIRQ(TIM2_IRQn);
        /* Enable update interrupt generation */
        // Relevant register: TIM2->DIER
        TIM2->DIER |= TIM_DIER_UIE;
}

void myEXTI_Init()
{
/* Map EXTI1 line to PA1 */
SYSCFG->EXTICR[0] = 0x0000;
/* EXTI1 line interrupts: set rising-edge trigger */
// Relevant register: EXTI->RTSR
EXTI->RTSR |= EXTI_RTSR_TR1;
/* Unmask interrupts from EXTI1 line */
// Relevant register: EXTI->IMR
EXTI->IMR |= EXTI_IMR_MR1;
/* Assign EXTI1 interrupt priority = 0 in NVIC */
// Relevant register: NVIC->IP[1], or use NVIC_SetPriority
NVIC_SetPriority(EXTI0_1_IRQn, 0);
/* Enable EXTI1 interrupts in NVIC */
// Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
NVIC_EnableIRQ(EXTI0_1_IRQn);
}
/* This handler is declared in system/src/cmsis/vectors_stm32f0xx.c */
void TIM2_IRQHandler()
{
/* Check if update interrupt flag is indeed set */
if ((TIM2->SR & TIM_SR_UIF) != 0)
{
trace_printf("\n*** Overflow! ***\n");
/* Clear update interrupt flag */
// Relevant register: TIM2->SR
```

```
TIM2->SR &= 0xFFFE;
/* Restart stopped timer */
// Relevant register: TIM2->CR1
TIM2->CR1 |= TIM_CR1_CEN;
}
}
/* This handler is declared in system/src/cmsis/vectors_stm32f0xx.c */
void EXTI0_1_IRQHandler()
{
	float seconds = 0;
	unsigned int temp;
	unsigned int temp_counter;
	/* Check if EXTI1 interrupt pending flag is indeed set */
	if ((EXTI->PR & EXTI_PR_PR1) != 0)
	 {
		if(edge == 1){
			TIM2->CNT = 0;
			edge = 0;
			TIM2->CR1 |= TIM_CR1_CEN;
		}

		else {
			TIM2->CR1 &= ~(TIM_CR1_CEN);
			temp_counter = (unsigned int)TIM2->CNT;

			TIM2->CR1 &= ~(TIM_CR1_CEN);
			seconds = ((float)temp_counter)/48000000;
			temp = (unsigned int)(1/seconds);
			frequency = temp;
			counter = temp_counter;
			edge = 1;
		}

		EXTI->PR &= EXTI_PR_PR1;
	}
}
#pragma GCC diagnostic pop
// ----------------------------------------------------------------------
```

6.0 References

[1] A. Jooya and D. Rakhmatov, “Home Page of ECE 355, “University of Victoria, [Online]. Available: http://www.ece.uvic.ca/~ece355/lab/. [Accessed November 2019].
[2] D. Rakhmatov, “ECE 355 Microprocessor-Based Systems,” [Online]. Available: http://www.ece.uvic.ca/~daler/courses/ece355/. [Accessed November 2019].