University of Victoria
Department of Electrical Engineering & Computer Science
ECE 355 – Microprocessor-Based Systems
Project Report
Fall Term 2020

PWM Signal Generation and Monitoring System

December 2, 2020

Branden Voss
V00913539
Electrical Engineering
brandenvoss97@hotmail.com
December 2, 2020
B09

Report Marking
The lab report marks are distributed as follows:

- Problem Description/Specifications (5) _____
- Design/Solution (15) _____
- Testing/Results (10) _____
- Discussion (15) _____
- Code Design and Documentation (15) _____
- Total (60) _____

Executive Summary

Electrical and Computer Engineering 355 is a course intended to develop a general understanding of the operation, programming, application and design of 32-bit microprocessor-based systems. In particular, the lab section of this course requires students to develop a system for PWM signal generation and monitoring. An external 4N35 Optocoupler, driven by a microcontroller will be used to control the frequency of the PWM signal generated by an external 555 timer. The microcontroller on the STMF0 Discovery board is used to measure the voltage across a potentiometer and relay it to the optocoupler for signal frequency control. Both the measured frequency and corresponding potentiometer resistance are to be displayed on an LCD. Students are expected to develop an embedded system using the C-programming language for this specific implementation [1].

Table of Contents

List of Figures

List of Tables

Glossary

PWM	Pulse width modulation
NE555 Timer	Timing circuits that are highly stable controllers capable of producing accurate time delays or oscillation
4N35 Optocoupler	Electronic component for transferring signals between two isolated circuits
POT	Potentiometer
Microcontroller	A small computer with memory and programmable input/output
ADC	Analog-to-digital converter
DAC	Digital-to-analog converter
LCD	Liquid crystal display
trace_printf	C-programming syntax used in the Eclipse IDE on the STM32F051R8T6 microcontroller for printing to console

Project Description

Through the use of techniques and topics discussed in ECE 355 and other applicable areas of study, students are expected to individually develop an embedded system for monitoring and controlling a PWM signal. For this project, an LCD is used to display the signal frequency and potentiometer resistance. The system utilizes the STM32F051R8T6 microcontroller mounted on

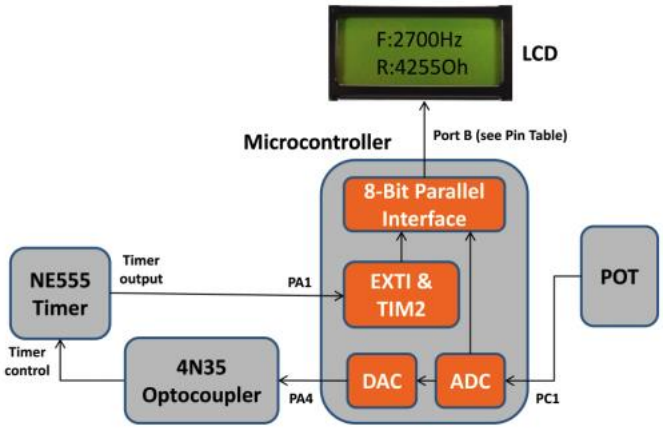


Figure 1: PWM Signal Generation and Monitoring System Diagram [1]

the STM32F0 Discovery board along with external; potentiometer, NE555 timer, 4N35 optocoupler and LCD as presented in figure 1. The STM32F0 Discovery board features a built-in analog-to-digital converter (ADC) used to measure the analog voltage signal from the potentiometer through a polling approach. Additionally, the board houses a digital-to-analog converter (DAC) used to drive the optocoupler to adjust the signal frequency of the 555 timer based off of the ADC voltage reading. The corresponding pin, signal and direction are mapped out for each respective piece of hardware in the above figure. Lastly, an 8-bit parallel interface is used to communicate with the LCD through 4 control pins and 8 data pins are also mapped [1].

Design Process

The ensuing portion of the report will provide insight into the design procedure for this project. The next two subsections will expand upon the initial design intentions and development, followed by the final implementation. This part of the report is expected to give insight into both the design and function of the system for simplicity of explanation and understanding.

Design Approach

In part 2 of the introductory lab, students measured and wrote to the console both the period and frequency of a square-wave signal from a function generator. A C-program template was provided for the laboratory session and required students to write code in order to access the necessary ports, registers and timers. This program was intended to determine the number of clock cycles elapsed between two consecutive rising edges of the function generators signal. The lab set-up is shown in figure 2 below. This implementation made use of a general-purpose timer TIM2 to measure the frequency of

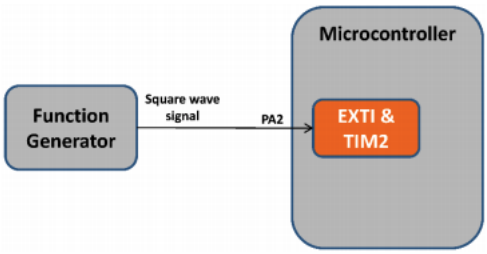


Figure 2: Introductory Lab Part 2 Set-up [1]

the input signal and record the current count value of the timer pulses in the TIM2_CNT counter register which was configured to increment every cycle of TIM2's clock [1]. Then, using the external interrupt line EXTI2 setup with pin PA2 on the microcontroller, interrupt requests (IRQ) prompted an IRQ handler necessary for counter start, stop and read operations [2]. Finally, code was written in order to calculate the period and frequency of the input signal which was to be displayed on the console.

Completion of part 2 of the introductory lab built a foundation for the PWM Generation and Monitoring System project. As per the project description previously discussed, the

Table 1: Pin Map for LCD Interface [1]

STM32F0	SIGNAL	DIRECTION
PA0	USER PUSH BUTTON	INPUT
PC8	BLUE LED	OUTPUT
PC9	GREEN LED	OUTPUT
PA1	555 TIMER	INPUT
PA2	FUNCTION GENERATOR (for Part 2 only)	INPUT
PA4	DAC	OUTPUT (Analog)
PC1	ADC	INPUT (Analog)
PB4	ENB (LCD Handshaking: "Enable")	OUTPUT
PB5	RS (0 = COMMAND, 1 = DATA)	OUTPUT
PB6	R/W (0 = WRITE, 1 = READ)	OUTPUT
PB7	DONE (LCD Handshaking: "Done")	INPUT
PB8	D0	OUTPUT
PB9	D1	OUTPUT
PB10	D2	OUTPUT
PB11	D3	OUTPUT
PB12	D4	OUTPUT
PB13	D5	OUTPUT
PB14	D6	OUTPUT
PB15	D7	OUTPUT

existing code was designed to function as specified. The corresponding pin, signal and direction are mapped out for each respective piece of hardware as in table 1. As partial completion of the design approach, pseudo code as shown in figure 3 was written to

```
114  void myLCD_Display ()
115  {
116      //<<8 to left shift data
117      //0x20 to write data, bit 5 set to 1
118      //send prefix frequency display format
119      //DDRAM address set to top row first position
120      //LCD handshake enable
121      //Done
122      //deassert enable
123      //reset Done
124
125      //write F to top row LCD
126      //LCD handshake enable
127      //Done
128      //deassert enable
129      //reset Done
130
131
132      //write : to top row LCD
133      //LCD handshake enable
134      //Done
135      //deassert enable
136      //reset Done
137
138      //send frequency data
139      //write first frequency digit
140      //LCD handshake enable
141      //Done
```

Figure 3: Pseudo Code

begin to put together the appropriate operations and initializations involved in developing

the system. This would require initializations for; port B to operate the LCD, PC1 for polling the ADC reading, and additional port A pins. Furthermore, initialization functions would be needed for LCD configuration and ADC/DAC configuration. At this stage in the design process, it was determined that an infinite while loop in the main function would be responsible for administering LCD write and ADC read operations.

Design Implementation

The solution for this project can be further divided into 3 subsections to aid in understanding the congruent function of the PWM Generation and Monitoring System project.

ADC

The built-in analog-to-digital converter on the STM32F051R8T6 microcontroller is configured as analog input on pin PC1 as in figure 4 below. In order to continuously measure the analog voltage signal from the potentiometer the ADC data register (ADC_DR) is polled from the infinite while loop (while (1)) in the programs main function. However, before this can work the ADC must be properly initialized. In order for the project to work access to the ADC channel

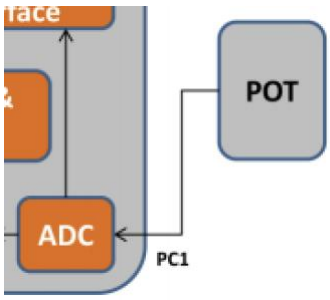


Figure 4: PC1 as Analog Input to ADC [1]

selection (ADC_CHSELR), data (ADC_DR), control (ADC_CR), sampling time (ADC_SMPR), interrupt and status (ADC_ISR) and configuration

(ADC_CFGR1) registers are needed [3]. Within an initialization function titled myConverter_Init the following steps are performed in this order;

1. Clock enabled for ADC
2. Data resolution is set to 12
3. Overrun management mode set to overwrite ADC_DR contents (can only be written while ADSTART = 0)
4. Continuous conversion mode is set (can only be written while ADSTART = 0)
5. Set ADC enable bit to 1 of the ADC_CR
6. Set sampling time to 239.5 ADC clock cycles (can only be written while ADSTART = 0)
7. Set ADC conversions to channel 11
8. Start the ADC

```
410 void myConverter_Init()
411 {
412     RCC->APB2ENR |= RCC_APB2ENR_ADCEN;//enable ADC clock
413
414     RCC->APB1ENR |= RCC_APB1ENR_DACEN;//enable DAC clock
415
416     ADC1->CFGR1 &= ~(ADC_CFGR1_RES);//set data resolution to 12
417
418     ADC1->CFGR1 |= ADC_CFGR1_OVRMOD;//set overrun management mode to overwrite ADC_DR contents
419
420     ADC1->CFGR1 |= ADC_CFGR1_CONT;//set continuous conversion mode
421
422     ADC1->CR |= ADC_CR_ADEN;//sets ADC enable bit to 1
423
424     ADC1->SMPR |= ADC_SMPR_SMP;//sets sampling time to 239.5 ADC clock cycles when ADSTART = 0
425
426     ADC1->CHSELR |= ADC_CHSELR_CHSEL11;//set to channel 11 for ADC conversion.
427
428     while((ADC1->ISR & ADC_ISR_ADRDY) != 1);//wait until ADC ready flag is 1
429
430     ADC1->CR |= ADC_CR_ADSTART;//starts ADC
431
432     DAC->CR = DAC_CR_EN1;//enabled DAC
```

Figure 5: myConverter_Init Function

A fully commented screenshot of code is provided in figure 5. Following step 7 a wait statement is inserted to ensure the ADC ready flag is set to 1. This bit is set after the ADC is enabled and when it becomes ready to accept conversion

requests. Two more wait statements within the main functions infinite while loop check to ensure the ADC is ready for conversion and that the channel conversion is complete. The ADC is now ready for conversion and can be polled through PC1.

DAC

The digital signal from the ADC is passed on to the DAC where it is restored to an analog signal. The analog signal from the DAC drives the optocoupler to adjust the frequency of the PWM signal generated from the 555 timer as in figure 6. The DAC is configured as analog output on pin PA4 of the microcontroller. In the same initialization function as before called myConverter_Init, the appropriate steps are performed by accessing the DAC control register (DAC_CR). First, a

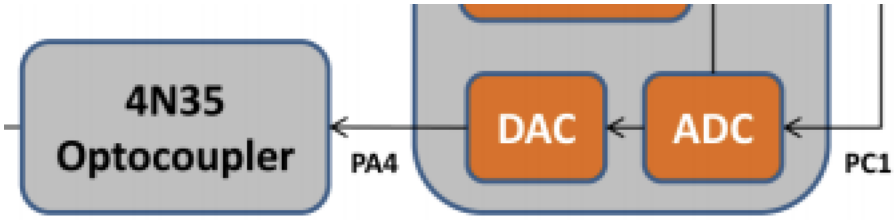


Figure 6: System Path (DAC to Optocoupler) [1]

clock is enabled for the DAC followed by setting the DAC channel1 enable bit to 1. These operations can be referred to in the previous sections figure 5. Additionally, the contents of the ADC data register are passed on to the DAC data register (DAC_DHR12R1) in the while (1) statement in the programs main function. These are necessary procedures in setting up the DAC for conversion of the signal received from the ADC [3].

LCD

The LCD is implemented to display the potentiometer resistance and the frequency of the 555-timer input signal; both of which are dependent on the voltage across the potentiometer. It is important to understand that the values sent to be displayed by the LCD must be sent digit by digit as their appropriate ASCII value [4]. This deliverable is met by applying the initialization functions myLCD_Init, myLCD_Display and functions for converting the values of both the frequency and resistance to ASCII called myRes_Calc and myFreq_Calc respectively. First, myLCD_Init is completes the following steps;

1. DDRAM access is performed using 8-bit interface
2. Display is turned on
3. DDRAM access is auto-incremented after each access so characters written to display are not overlapped
4. Clear display

Each instruction in the initialization function is preceded by four instructions necessary to establish handshaking with the LCD. These instructions first make

```
381 void myLCD_Init()
382 {
383     GPIOB->ODR = 0b0011000000000000;//DDRAM access is performed using 8-bit interface
384     GPIOB->ODR |= GPIO_ODR_4;//LCD handshake enable
385     while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_SET);//Done
386     GPIOB->ODR &= ~(GPIO_ODR_4);//deassert enable
387     while(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_RESET);//reset Done
388
389     GPIOB->ODR = 0b0000110000000000;//display on
390     GPIOB->ODR |= GPIO_ODR_4;//LCD handshake enable
391     while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_SET);//Done
392     GPIOB->ODR &= ~(GPIO_ODR_4);//deassert enable
393     while(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_RESET);//reset Done
```

Figure 7: myLCD_Init Function

bit 4 of PB to become 1 in order to assert enable followed by a while statement that waits for bit 7 of PB to become 1 signaling the latter step is done. Then, a

software write makes PB become 0 again to deassert enable followed by a second while which waits for bit 7 of PB to again be 0 to indicate enable has been deasserted [5]. Figure 7 above shows an example of this procedure.

Next, the data present in ADC_DR and the calculated frequency must be sent in ASCII, digit by digit, through separate instructions to the LCD display. Before this data can be passed into the myLCD_Display function, it first must be

```
232 void myRes_Conv()
233 {
234     //variables are type int
235     //decimal value for each digit place
236     thous_res = resistance / 1000;//returns single digit value for thousandths place
237     huns_res = (resistance - thous_res * 1000) / 100;//returns single digit value for hundredths place
238     tens_res = (resistance - thous_res * 1000 - huns_res * 100) / 10;//returns single digit value for tenths place
239     ones_res = (resistance - thous_res * 1000 - huns_res * 100 - tens_res * 10);//returns single digit value for ones pla
240
241     //convert to ASCII value
242     thous_res += 48;
243     huns_res += 48;
244     tens_res += 48;
245     ones_res += 48;
246 }
```

Figure 8: myRes_Calc Function (Identical to myFreq_Calc Function)

manipulated by the myRes_Calc and myFreq_Calc functions. Both functions work in the same manner with the only difference being that on handles resistance and one handles frequency. The myRes_Calc function can be seen in figure 8 above [4].

Since the LCD is required to display 4 digits for resistance each digit is assigned a place value with the right most digit being the thousandths place and the left most digit being the ones place value. The function calculates the thousandths digit first. The total decimal value called ‘resistance’ is divided by 1000. The result of this division will be a single digit number ranging from 0-9 possibly with non-zero values after the decimal. However, because the variable that holds the thousandths place value is of type int the decimals are truncated leaving a single digit ranging from 0-9. The hundredths place value is determined next by dividing

by 100 after the thousandths place value is subtracted from the original value. Then the tenths place is calculated by dividing by 10 after the thousandths and hundredths place values are subtracted from the original value. Lastly the ones place value is found by a trivial division by 1 after subtracting the all the other place values from the original number. Once each of the 4 digits are calculated, decimal 48 is added to each one respectively in order to determine its ASCII value [4]. An example is done by hand to show how this would work below. Note that the myFreq_Calc function works exactly the same.

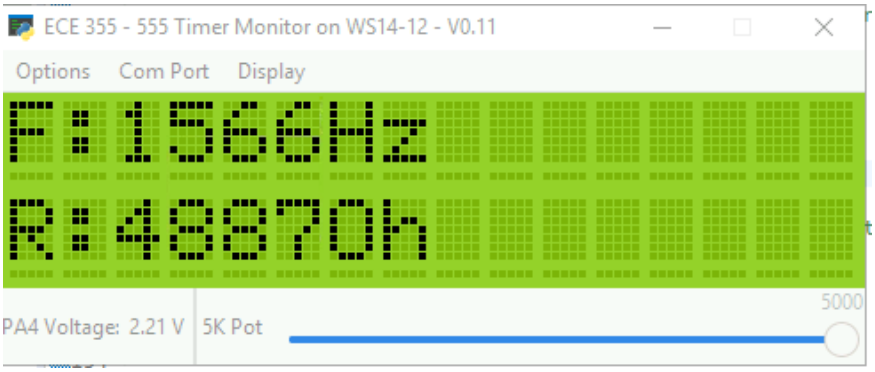
Finally, the myLCD_Display function is used to send each digit or character in ASCII separately to the display shown in figure 9. As previously discussed, each write instruction is followed by 4 handshaking instructions. The LCD display appears as shown in figure 8. The resistance and frequency data are sent as variables and therefore must be left aligned after a bitwise ‘or’ with 0x0020. This ensures that none of the data is altered when writing to the LCD.

```
112 void myLCD_Display ()
113 {
114     //<<8 to left shift data
115     //0x20 to write data, bit 5 set to 1
116     //send prefix frequency display format
117     GPIOB->ODR = 0x8000;//DDRAM address set to top row first position
118     GPIOB->ODR |= GPIO_ODR_4;//LCD handshake enable
119     while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_SET);//Done
120     GPIOB->ODR &= ~(GPIO_ODR_4);//deassert enable
121     while(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_RESET);//reset Done
122
123     GPIOB->ODR = 0x4620;//write F to top row LCD
124     GPIOB->ODR |= GPIO_ODR_4;//LCD handshake enable
125     while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_SET);//Done
126     GPIOB->ODR &= ~(GPIO_ODR_4);//deassert enable
127     while(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_RESET);//reset Done
```

Figure 9: myLCD_Display Function

Analysis

Throughout the design process it is necessary to assess the project to ensure correctness. One method that was used frequently was to write trace_printf statements to the console. These prints could be compared to the LCD display values for both frequency and resistance. Several trace_printf statements were also used to track the program flow and locate any possible structural issues. An example is shown in figure 10. These are some techniques used for implementing the ADC, DAC and LCD components. The development of this project required extensive troubleshooting and analysis to understand how the system operates as a whole.



a)

```
frequency: 1570.423706
resistance: 4913
frequency: 1565.506714
resistance: 4888
frequency: 1573.203125
resistance: 4890
frequency: 1567.858887
resistance: 4890
frequency: 1564.639160
resistance: 4890
```

b)

Figure 10: Comparison of LCD a) and Console Output b)

One particular issue encountered was rooted in the failure to appropriately establish the handshaking procedure with the LCD. Initially the program was written with the incorrect handshaking steps as discussed earlier and the LCD was not being cleared. A flow diagram of the handshaking procedure is seen in figure 11. A large portion of understanding the system came from hands-on experience and observations. Some measurement errors may arise from software such as the variable type used to store numbers and how those numbers are rounded or manipulated by operations. Also, the program flow and overhead caused by executing function could create error in a system that relies on real time measurements as well. However, the project encounters limitations in its ability to both measure frequency and measure resistance due to the system. The frequency is determined by dividing the system clock of 48MHz with the timer pulse count recorded in TIM2_CNT. Limitations arise on both the upper and lower ends of the measured frequency. First, it must be taken into consideration that an interrupt request takes time to service. Once a rising edge is detected the system will prompt the IRQ handler. If before the handler operation can be executed, the 555 timer sends a second rising edge it will be missed. This would effect lower frequency readings in a significant manner. Additionally, for higher frequencies the calculation may become inaccurate in instances where consecutive rising edges occur outside the maximum range of TIM2. TIM2 is a 32-bit counter and therefore is capable of counting to $2^{32} - 1 = 4294967295$. If the time between two consecutive rising edges exceeds this

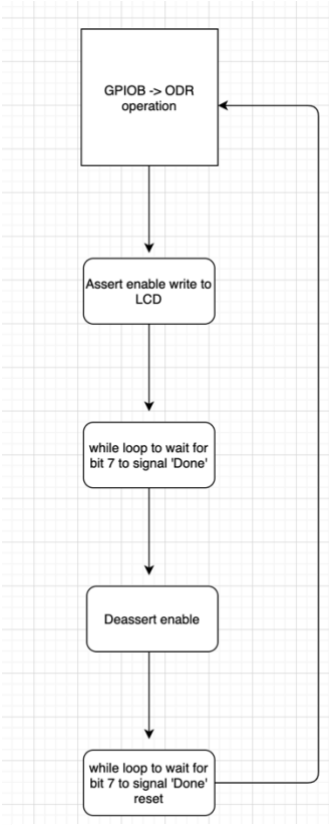


Figure 11: Flow Diagram of Handshaking Procedure

range TIM2 will experience overflow and will begin the count over again as it is an auto-reload counter [6].

Likewise, the resistance measurement also experiences some short comings due to system design. In the myADC_Init function the ADC's resolution is set to 12 meaning that the ADC_DR is capable of recording $2^{12} - 1 = 4095$. Unfortunately, the maximum resistance value of the potentiometer is 5000. For this reason, a scaling factor was introduced to bring the resistance to a more accurate reading. Considering that the ADC_DR value maximum is 4095 and the true maximum is 5000, the scaling factor was determined by dividing 5000 by 4095. In order to implement this the ADC_DR value is assigned to a variable called ADC_val inside while (1) in the main function as shown in figure 9. ADC_val multiplied 5000 and the result is then divided by 4095 [6]. This method proves to give an accurate resistance reading on the LCD display when compared with the true value and trace_printf statements.

Reflection and Recommendations

Through techniques discussed earlier the project presented in this report proves to accurately achieve the goals outlined in the project description. Upon further study and analysis it can be seen that there are many areas subject to potential improvement. The developed C-program is written in a basic format meaning that procedures are done on an instruction-by-instruction basis without the use of many iterative procedures. Refer to a complete listing of commented code in Appendix A at the back of this report. The program was developed in the way for ease of understanding and a better ability to observe the commands written line by line. This program also makes use of functions and function calls in an attempt to keep the format clean and easy to follow. Some pivotal lessons acquired from completing this project should also be noted. One example in particular was the use of the four handshaking instructions necessary for writing to

the LCD. This enabled the LCD to display the desired output. It is also important to note that multiple trace_printf statements can create a fair amount of overhead on the system software. While a trace_printf statement is running, it will prevent the processor from executing other instructions. This can slow down the system and present other errors within such a system that relies on real-time data. This triggered issues early on when trying to read frequency and resistance.

Conclusion

The PWM Signal Generation and Monitoring System project was proven to be attainable through the use of methods, techniques and knowledge discussed in this report. Extensive work was placed on both the development and testing of the system in order to meet the specifications outlined in the lab manual and the previous Project Description section. The measurements required are accurately determined by the use of several hardware components working simultaneously under the designed software. The implemented design successfully controls and measures a square-wave signal generated from the 555 timer. The measured resistance value from the external potentiometer correctly determines this frequency using the built-in ADC and DAC features. This is a necessary part to drive the external optocoupler and ultimately adjust the signal frequency at the 555 timer. Lastly, the designed system is able to successfully communicate with the LCD through the 8-bit parallel interface in order to display the measurements as an accurate reading. The design process and final result has served as a fundamental tool in understanding the development and applications of embedded systems.

Bibliography

[1] D. Rakhmatov, "ECE 355: Microprocessor-Based Systems Laboratory Manual (ONLINE)," University of Victoria, Victoria, 2020.

[2] D. Rakhmatov, "I/O Examples," University of Victoria, Victoria, 2020.

[3] D. Rakhmatov, "Lecture 17," University of Victoria, Victoria, 2020.

[4] D. Rakhmatov, "Lecture 16," University of Victoria, Victoria, 2020.

[5] L. Hitachi, "HD44780U (LCD-II) Hitachi," Hitachi, Victoria, 1998.

[6] STMicroelectronics, "STM32F0 Reference Manual," STMicroelectronics - All Rights Reserved, Victoria, 2014.

[7] D. Rakhmatov, "Interface Examples," University of Victoria, Victoria, 2020.

[8] V. Semiconductors, "Optocoupler, Phototransistor Output, with Base Connection," Vishay, Victoria, 2012.

[9] STMicroelectronics, "NE555 General-purpose single bipolar timers," STMicroelectronics - All Rights Reseerved, Victoria, 2012.

Appendix A – Complete Listing of Annotated Code

This section provides a full listing of well-commented source code to be referenced as necessary.

The following C-program is a direct copy of the code implemented for this project and discussed in this report.

```
//
// This file is part of the GNU ARM Eclipse distribution.
// Copyright (c) 2014 Liviu Ionescu.
//

// ----------------------------------------------------------------------------
// School: University of Victoria, Canada.
// Course: ECE 355 "Microprocessor-Based Systems".
// This is template code for Part 2 of Introductory Lab.
//
// See "system/include/cmsis/stm32f0xx.h" for register/bit definitions.
// See "system/src/cmsis/vectors_stm32f0xx.c" for handler declarations.
// ----------------------------------------------------------------------------

#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"
#pragma GCC diagnostic ignored "-Wreturn-type"


/* Clock prescaler for TIM2 timer: no prescaling */
#define myTIM2_PRESCALER ((uint16_t)0x0000)
/* Maximum possible setting for overflow */
#define myTIM2_PERIOD ((uint32_t)0xFFFFFFFF)
#define myTIM3_PRESCALER ((uint16_t)47999)

void myGPIO_Init(void);
void myLCD_Init(void);
void myTIM2_Init(void);
void myEXTI_Init(void);
void myConverter_Init(void);
void myLCD_Display(void);
void myRes_Conv(void);
void myFreq_Conv(void);

// Declare/initialize your global variables here...
// NOTE: You'll need at least one global variable
// (say, timerTriggered = 0 or 1) to indicate
// whether TIM2 has started counting or not.
int timerTriggered = 0;
int frequency = 0;
float count_val = 0;
//float period = 0;
int ADC_val = 0;
float scalar = (5000/4095);
int resistance = 0;
```

```
//using type int to truncate decimals during conversion functions
int thous_res = 0;
int huns_res = 0;
int tens_res = 0;
int ones_res = 0;
int thous_freq = 0;
int huns_freq = 0;
int tens_freq = 0;
int ones_freq = 0;

int main(int argc, char* argv[])
{

    trace_printf("This is the final part of ECE 355 Lab...\n");
    trace_printf("System clock: %u Hz\n", SystemCoreClock);

    myGPIO_Init();/* Initialize I/O port PA */
    myTIM2_Init();/* Initialize timer TIM2 */
    myEXTI_Init();/* Initialize EXTI */
    myLCD_Init();/* Initialize EXTI */
    myConverter_Init();/* Initialize ADC and DAC*/

    trace_printf("Initialization Done\n");

    while (1)
    {
        while((ADC1->ISR & 0x1) == 0);//While ADC ready

        while((ADC1->ISR & ADC_ISR_EOC) != ADC_ISR_EOC);//while channel conversion complete

        ADC_val = ADC1->DR;//read ADC_DR and assign to variable

        DAC->DHR12R1 = ADC1->DR;//pass ADC data to DAC

        resistance = ADC_val * scalar;//operation to calculate POT resistance

        myRes_Conv();//call to convert individual digits to ASCII
```

```
        myFreq_Conv();//call to convert individual digits to ASCII

        myLCD_Display();//call to LCD display function
    }

    return 0;

}

void myLCD_Display ()
{
    //<<8 to left shift data
    //0x20 to write data, bit 5 set to 1
    //send prefix frequency display format
    GPIOB->ODR = 0x8000;//DDRAM address set to top row first position
    GPIOB->ODR |= GPIO_ODR_4;//LCD handshake enable
    while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_SET);//Done
    GPIOB->ODR &= ~(GPIO_ODR_4);//deassert enable
    while(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_RESET);//reset Done

    GPIOB->ODR = 0x4620;//write F to top row LCD
    GPIOB->ODR |= GPIO_ODR_4;//LCD handshake enable
    while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_SET);//Done
    GPIOB->ODR &= ~(GPIO_ODR_4);//deassert enable
    while(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_RESET);//reset Done


    GPIOB->ODR = 0x3A20;//write : to top row LCD
    GPIOB->ODR |= GPIO_ODR_4;//LCD handshake enable
    while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_SET);//Done
    GPIOB->ODR &= ~(GPIO_ODR_4);//deassert enable
    while(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_RESET);//reset Done

    //send frequency data
    GPIOB->ODR = 0x20 | (thous_freq<<8);//write first frequency digit
    GPIOB->ODR |= GPIO_ODR_4;//LCD handshake enable
```

```
while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_SET);//Done
GPIOB->ODR &= ~(GPIO_ODR_4);//deassert enable
while(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_RESET);//reset Done

GPIOB->ODR = 0x20 | (huns_freq<<8);//write second frequency digit
GPIOB->ODR |= GPIO_ODR_4;//LCD handshake enable
while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_SET);//Done
GPIOB->ODR &= ~(GPIO_ODR_4);//deassert enable
while(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_RESET);//reset Done

GPIOB->ODR = 0x20 | (tens_freq<<8);//write third frequency digit
GPIOB->ODR |= GPIO_ODR_4;//LCD handshake enable
while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_SET);//Done
GPIOB->ODR &= ~(GPIO_ODR_4);//deassert enable
while(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_RESET);//reset Done

GPIOB->ODR = 0x20 | (ones_freq<<8);//write fourth frequency digit
GPIOB->ODR |= GPIO_ODR_4;//LCD handshake enable
while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_SET);//Done
GPIOB->ODR &= ~(GPIO_ODR_4);//deassert enable
while(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_RESET);//reset Done

//send suffix frequency display format
GPIOB->ODR = 0x4820;//write H to top row LCD
GPIOB->ODR |= GPIO_ODR_4;//LCD handshake enable
while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_SET);//Done
GPIOB->ODR &= ~(GPIO_ODR_4);//deassert enable
while(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_RESET);//reset Done

GPIOB->ODR = 0x7A20;//write z to top row LCD
GPIOB->ODR |= GPIO_ODR_4;//LCD handshake enable
while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_SET);//Done
GPIOB->ODR &= ~(GPIO_ODR_4);//deassert enable
while(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_RESET);//reset Done

//send prefix resistance display format
GPIOB->ODR = 0xC000;//DDRAM address set to bottom row first position
```

```
GPIOB->ODR |= GPIO_ODR_4;//LCD handshake enable
while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_SET);//Done
GPIOB->ODR &= ~(GPIO_ODR_4);//deassert enable
while(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_RESET);//reset Done

GPIOB->ODR = 0x5220;//write R to bottom row LCD
GPIOB->ODR |= GPIO_ODR_4;//LCD handshake enable
while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_SET);//Done
GPIOB->ODR &= ~(GPIO_ODR_4);//deassert enable
while(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_RESET);//reset Done

GPIOB->ODR = 0x3A20;//write : to bottom row LCD
GPIOB->ODR |= GPIO_ODR_4;//LCD handshake enable
while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_SET);//Done
GPIOB->ODR &= ~(GPIO_ODR_4);//deassert enable
while(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_RESET);//reset Done

//send resistance data
GPIOB->ODR = 0x20 | (thous_res<<8);//write first resistance digit
GPIOB->ODR |= GPIO_ODR_4;//LCD handshake enable
while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_SET);
GPIOB->ODR &= ~(GPIO_ODR_4);//deassert enable
while(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_RESET);//reset Done

GPIOB->ODR = 0x20 | (huns_res<<8);//write second resistance digit
GPIOB->ODR |= GPIO_ODR_4;//LCD handshake enable
while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_SET);//Done
GPIOB->ODR &= ~(GPIO_ODR_4);//deassert enable
while(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_RESET);//reset Done

GPIOB->ODR = 0x20 | (tens_res<<8);//write third resistance digit
GPIOB->ODR |= GPIO_ODR_4;//LCD handshake enable
while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_SET);//Done
GPIOB->ODR &= ~(GPIO_ODR_4);//deassert enable
while(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_RESET);//reset Done

GPIOB->ODR = 0x20 | (ones_res<<8);//write fourth resistance digit
```

```
    GPIOB->ODR |= GPIO_ODR_4;//LCD handshake enable
    while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_SET);//Done
    GPIOB->ODR &= ~(GPIO_ODR_4);//deassert enable
    while(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_RESET);//reset Done

    //send suffix resistance display format
    GPIOB->ODR = 0x4F20;//write O to bottom row LCD
    GPIOB->ODR |= GPIO_ODR_4;//LCD handshake enable
    while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_SET);//Done
    GPIOB->ODR &= ~(GPIO_ODR_4);//deassert enable
    while(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_RESET);//reset Done

    GPIOB->ODR = 0x6820;//write h to bottom row LCD
    GPIOB->ODR |= GPIO_ODR_4;//LCD handshake enable
    while (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_SET);//Done
    GPIOB->ODR &= ~(GPIO_ODR_4);//deassert enable
    while(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_7) != Bit_RESET);//reset Done
}

void myRes_Conv()
{
    //variables are type int
    //decimal value for each digit place
    thous_res = resistance / 1000;//returns single digit value for thousandths place
    huns_res = (resistance - thous_res * 1000) / 100;//returns single digit value for hundredths place
    tens_res = (resistance - thous_res * 1000 - huns_res * 100) / 10;//returns single digit value for tenths place
    ones_res = (resistance - thous_res * 1000 - huns_res * 100 - tens_res * 10);//returns single digit value for ones
place

    //convert to ASCII value
    thous_res += 48;
    huns_res += 48;
    tens_res += 48;
    ones_res += 48;
}

void myFreq_Conv()
```

```
{
    //variables are type int
    //decimal value for each digit place
    thous_freq = frequency / 1000;//returns single digit value for thousandths place
    huns_freq = (frequency - thous_freq * 1000) / 100;//returns single digit value for hundredths place
    tens_freq = (frequency - thous_freq * 1000 - huns_freq * 100) / 10;//returns single digit value for tenths place
    ones_freq = (frequency - thous_freq * 1000 - huns_freq * 100 - tens_freq * 10);       //returns single digit value for
ones place

    //convert to ASCII value
    thous_freq += 48;
    huns_freq += 48;
    tens_freq += 48;
    ones_freq += 48;
}

void myGPIO_Init()
{
    /* Enable clock for GPIOA peripheral */
    // Relevant register: RCC->AHBENR
    RCC->AHBENR |= RCC_AHBENR_GPIOAEN;

    /* Enable clock for GPIOB peripheral */
    // Relevant register: RCC->AHBENR
    RCC->AHBENR |= RCC_AHBENR_GPIOBEN;

    /* Enable clock for GPIOC peripheral */
    // Relevant register: RCC->AHBENR
    RCC->AHBENR |= RCC_AHBENR_GPIOCEN;

    /* Configure PA1 as input */
    // Relevant register: GPIOA->MODER
    GPIOA->MODER &= ~(GPIO_MODER_MODER1);//PA1 input from 555 timer

    /* Ensure no pull-up/pull-down for PA1 */
    // Relevant register: GPIOA->PUPDR
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR1);
```

```
    /* Configure PC1 as analog input */
    // Relevant register: GPIOC->MODER
    GPIOC->MODER |= GPIO_MODER_MODER1;//ADC Analog in from POT

    /* Configure PA4 as output */
    // Relevant register: GPIOA->MODER
    GPIOA->MODER |= GPIO_MODER_MODER4;//DAC Analog out to opto

    /* Configure PB7 as input */
    // Relevant register: GPIOB->MODER
    GPIOB->MODER &= ~(GPIO_MODER_MODER7);//PB7 DONE LCD

    /* Configure PB4-6 and 8-15 as output */
    // Relevant register: GPIOB->MODER
    GPIOB->MODER |= (GPIO_MODER_MODER4_0);//PB4 ENB LCD

    GPIOB->MODER |= (GPIO_MODER_MODER5_0);//PB5 RS

    GPIOB->MODER |= (GPIO_MODER_MODER6_0);//PB6 R/W

    GPIOB->MODER |= (GPIO_MODER_MODER8_0);//PB8 D0

    GPIOB->MODER |= (GPIO_MODER_MODER9_0);//PB9 D1

    GPIOB->MODER |= (GPIO_MODER_MODER10_0);//PB10 D2

    GPIOB->MODER |= (GPIO_MODER_MODER11_0);//PB11 D3

    GPIOB->MODER |= (GPIO_MODER_MODER12_0);//PB12 D4

    GPIOB->MODER |= (GPIO_MODER_MODER13_0);//PB13 D5

    GPIOB->MODER |= (GPIO_MODER_MODER14_0);//PB14 D6

    GPIOB->MODER |= (GPIO_MODER_MODER15_0);//PB15 D7
}
```

```
void myTIM2_Init() //unchanged from part 1
{
    /* Enable clock for TIM2 peripheral */
    // Relevant register: RCC->APB1ENR
    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 delay */
    TIM2->ARR = myTIM2_PERIOD;

    /* Update timer registers */
    // Relevant register: TIM2->EGR
    TIM2->EGR = TIM_EGR_UG;//((uint32_t)0x00000001);

    /* 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 EXTI2 line to PA2 */
    // Relevant register: SYSCFG->EXTICR[0]
    SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI1_PA;//&= 0x0000;

    /* EXTI2 line interrupts: set rising-edge trigger */
    // Relevant register: EXTI->RTSR
    EXTI->RTSR |= EXTI_RTSR_TR1;

    /* Unmask interrupts from EXTI2 line */
    // Relevant register: EXTI->IMR
    EXTI->IMR |= EXTI_IMR_MR1;

    /* Assign EXTI2 interrupt priority = 0 in NVIC */
    // Relevant register: NVIC->IP[2], or use NVIC_SetPriority
    NVIC_SetPriority(EXTI0_1_IRQn, 0);

    /* Enable EXTI2 interrupts in NVIC */
    // Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
    NVIC_EnableIRQ(EXTI0_1_IRQn);
}

void myLCD_Init()
{
    GPIOB->ODR = 0;

    GPIOB->ODR = 0b0011000000000000;//DDRAM access is performed using 8-bit interface
    GPIOB->ODR |= GPIO_ODR_4;//enable write to LCD
    while((GPIOB->IDR & GPIO_IDR_7) == 0);//handshake
    GPIOB->ODR ^= GPIO_ODR_4;//disable write to LCD
    while((GPIOB->IDR & GPIO_IDR_7) != 0);//handshake

    GPIOB->ODR = 0b0000110000000000;//display on
    GPIOB->ODR |= GPIO_ODR_4;//enable write to LCD
    while((GPIOB->IDR & GPIO_IDR_7) == 0);//handshake
    GPIOB->ODR ^= GPIO_ODR_4;//disable write to LCD
    while((GPIOB->IDR & GPIO_IDR_7) != 0);//handshake
```

```
    GPIOB->ODR = 0b0000011000000000;//DDRAM address is auto-incremented after each access
    GPIOB->ODR |= GPIO_ODR_4;//enable write to LCD
    while((GPIOB->IDR & GPIO_IDR_7) == 0);//handshake
    GPIOB->ODR ^= GPIO_ODR_4;//disable write to LCD
    while((GPIOB->IDR & GPIO_IDR_7) != 0);//handshake

    GPIOB->ODR = 0b0000000100000000;//clear display
    GPIOB->ODR |= GPIO_ODR_4;//enable write to LCD
    while((GPIOB->IDR & GPIO_IDR_7) == 0);//handshake
    GPIOB->ODR ^= GPIO_ODR_4;//disable write to LCD
    while((GPIOB->IDR & GPIO_IDR_7) != 0);//handshake

    trace_printf("LCD Init done\n");
}

void myConverter_Init()
{
    RCC->APB2ENR |= RCC_APB2ENR_ADCEN;//enable ADC clock

    RCC->APB1ENR |= RCC_APB1ENR_DACEN;//enable DAC clock

    ADC1->CFGR1 &= ~(ADC_CFGR1_RES);//set data resolution to 12

    ADC1->CFGR1 |= ADC_CFGR1_OVRMOD;//set overrun management mode to overwrite ADC_DR contents

    ADC1->CFGR1 |= ADC_CFGR1_CONT;//set continuous conversion mode

    ADC1->CR |= ADC_CR_ADEN;//sets ADC enable bit to 1

    ADC1->SMPR |= ADC_SMPR_SMP;//sets sampling time to 239.5 ADC clock cycles when ADSTART = 0

    ADC1->CHSELR |= ADC_CHSELR_CHSEL11;//set to channel 11 for ADC conversion.

    while((ADC1->ISR & ADC_ISR_ADRDY) != 1);//wait until ADC ready flag is 1

    ADC1->CR |= ADC_CR_ADSTART;//starts ADC
```

```
    DAC->CR = DAC_CR_EN1;//enabled DAC
}

/* 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 &= ~(TIM_SR_UIF);

        /* Restart stopped timer */
        // Relevant register: TIM2->CR1
        TIM2->CR1 |= TIM_CR1_CEN;//TIM2_CR1_CEN
    }
}


/* This handler is declared in system/src/cmsis/vectors_stm32f0xx.c */
void EXTI0_1_IRQHandler()
{
    // Declare/initialize your local variables here...
    //float count_val = 0;
    //float period = 0;
    //float frequency = 0;
    /* Check if EXTI2 interrupt pending flag is indeed set */
    if ((EXTI->PR & EXTI_PR_PR1) != 0)
    {
        //
        // 1. If this is the first edge:
        if(timerTriggered == 0){
            //  - Clear count register (TIM2->CNT).
```

```
    TIM2->CNT = 0;
    //  - Start timer (TIM2->CR1).
    TIM2->CR1 |= TIM_CR1_CEN;
    timerTriggered = 1;
}
//    Else (this is the second edge):
else{
    //  - Stop timer (TIM2->CR1).
    TIM2->CR1 &= ~(TIM_CR1_CEN);
    ////////////////////////////////EXTI->IMR &= ~(EXTI_IMR_MR1);
    //  - Read out count register (TIM2->CNT).
    count_val = TIM2->CNT;

    //  - Calculate signal period and frequency.
    frequency = 48000000 / count_val;

    //period = 1 / frequency

    timerTriggered = 0;

    //  - Print calculated values to the console.
    //    NOTE: Function trace_printf does not work
    //    with floating-point numbers: you must use
    //    "unsigned int" type to print your signal
    //    period and frequency.
    //
    //trace_printf("Count value: %f\n", count_val);
    //trace_printf("Period: %f s\n", period);
    //trace_printf("Frequency: %f Hz\n", frequency);
    //trace_printf("Resistance: %d Ohms\n", resistance);
}
// 2. Clear EXTI2 interrupt pending flag (EXTI->PR).
// NOTE: A pending register (PR) bit is cleared
// by writing 1 to it.
//
EXTI->IMR |= EXTI_IMR_MR1;
```

```
        EXTI->PR = EXTI_PR_PR1;
    }
}

#pragma GCC diagnostic pop

// ----------------------------------------------------------------------------
```