University of Victoria

Department of Electrical and Computer Engineering

CENG 355 - Microprocessor-Based Systems

Laboratory Project Report

Name:	Justin Richard - V00773113	
Problem Description/Specifications:	(5)	______________.
Design/Solution	(15)	______________.
Testing/Results	(10)	______________.
Discussion	(15)	______________.
Code Design and Documentation:	(15)	______________.
Total	**(60)**	______________.

Table of Contents

Introduction

The purpose of this project was to develop an embedded system for monitoring and controlling a pulse-width-modulated (PWM) signal that was generated by an external timer (NE555 Timer). An external optocoupler (4N35 IC), driven by the microcontroller on the STMF0 Discovery board was used to control the frequency of the PWM signal. The microcontroller was used to measure the voltage across a potentiometer (POT) on the PBMCUSLK board and relay it to the external optocoupler for controlling the PWM signal frequency. The measured frequency and corresponding POT resistance were displayed on the LCD on the PBMCUSLK board. The overall system diagram is shown below in figure 1.

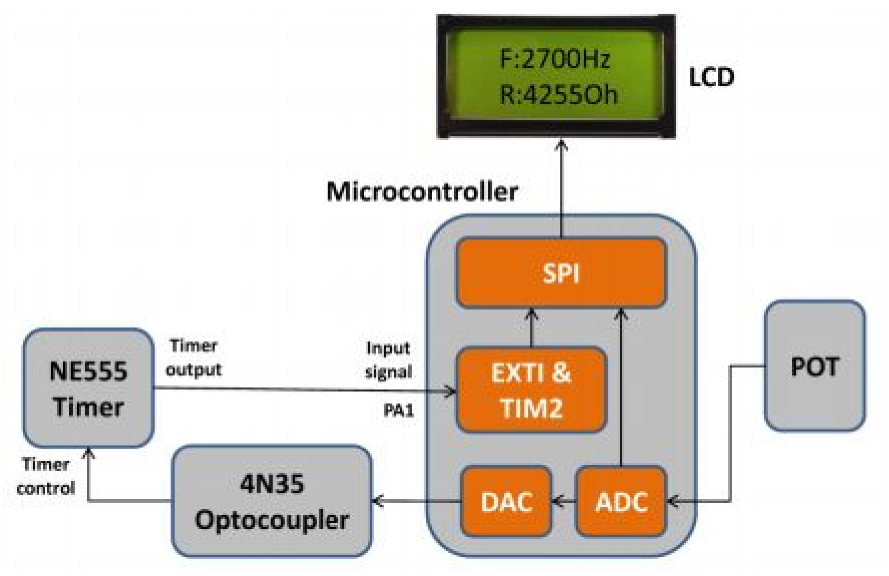

Figure 1. System diagram

The project specifications are as above alongside a few further specifications outlined in the lab manual. The analog voltage signal coming from potentiometer should be continuously measured using a polling approach. The digital value obtained from ADC will adjust frequency of PWM signal generated by the NE555 timer. The upper and lower limits of the potentiometer values should be measured. LCD should be written to by the SPI since there is no direct write access.

Implementation

The initial template for this project was the code from part 2 of the first laboratory session. The code was mostly kept the same, except for small details such as the period calculation being removed since it was not necessary for this project. The main, GPIO init, EXTI, and TIM2 initializations and handlers were kept and reused, modified only where needed. The project was created using an agile software development methodology, where parts were built incrementally in cycles in order to test components as we went. This method was chosen opposed to a predetermined waterfall method since we were unfamiliar with working with the components and wanted to ensure we could adequately adapt along the way. A circuit/wiring diagram can be seen below the following sections that outline the implementation of each component.

TIM2 Timer

The TIM2 timer initialization was identical to the initialization from the first lab assignment. The period and prescaler values for the timer were also the same. This was because the TIM2 timer serves the exact same purpose as in the first assignment, except this time the input signal is coming from the NE555 timer opposed to a signal generator. The TIM2 timer counts how many clock cycles have past between peaks in the input signal. Since it was known the clock speed is 48Mhz, one can then calculate the signals frequency from 48Mhz divided by the counted cycles.

External Interrupts

A single external interrupt, EXTI1 was used and mapped to trigger on PA1 as input, to fire on a rising edge. Pin PA1 was enabled and marked as an input pin in the GPIO initialization. The interrupt was unmasked so that another interrupt doesn't happen while the current one was still being processed. If PA1 detects an input then the interrupt handler EXTI0_1_IRQHandler will be called and processed which will calculate the frequency of the signal using the TIM2 timer value and the system clock speed. This will update a global variable for frequency so that the latest value can be accessed from other functions.

Analog to Digital Converter

The analog to digital converter (ADC) was initialized in the myADC_Init function. This function enables a few settings. First, the ADC clock is enabled, followed by enabling continuous conversion, setting the channel to use to channel 2 which corresponds to pin PA2, and setting the sampling time to the maximum value, 7. The maximum value was used for the sampling time to obtain more accurate results. Pin PA2 was enabled and set to analog mode in the GPIO initialization. Once those settings were set, the ADC was then set to calibrate through masking with the control register. Once the calibration was complete, the ADC was enabled for use.

The ADC used a simple analog signal as input, coming from the 3.3V pin on the PBMCUSLK board passing through a potentiometer to vary the values. The ADC is 12-bits, which means the output value varies between 0 and 4095. Since the maximum input voltage is 3.3V, that means that the minimum voltage change for the ADC to read and change values is 3.3V/4095=0.805mV.

Digital to Analog Converter

The digital to analog converter (DAC) was configured in the myDAC_Init function. The DAC was enabled on the default channel, channel 1 which maps its output to PA4. Prior to enabling the DAC pin PA4 was enabled and set to analog in the GPIO initialization. The last part of the DAC initialization is to set the DAC to trigger by software so that we can manually pass the ADC value into it when we want to which is done in the main method.

NE555 Timer

The NE555 timer has a couple different modes which suit it for a couple different purposes, but for this project the timer used as a pulse train generator. The timer was wired according to the Timer Tutorial page [4], as well as from the course slides on the optocoupler connection [2]. The output frequency depends on the values of resistors used in the circuit. A 100kΩ resistor was used as it was found that it increased the system stability and made it less prone to changes from noise. For the other resistor a separate potentiometer was used. This allowed it to vary the frequency alongside the resistance easily at any moment while the project was running. A supply voltage of 5V was supplied as recommended in the lab manual.

4N35 Optocoupler

The 4N35 Optocoupler is used to isolate two parts of the circuit from each other while still transmitting data across. This is often done to prevent a high voltage circuit from frying a lower voltage circuit. Depending on the amount of current received across pins 1 and 2 an infrared emitter will emit a different intensity light. A phototransistor or infrared collector between pins 4 and 5 will produce a corresponding resistance depending on the intensity of the light. This gives us a variable resistor we can connect in parallel to our NE555 timer. A wiring diagram to show how this is setup can be seen in the wiring section below.

Serial Peripheral Interface

The serial peripheral interface (SPI) was used in order to control the LCD pins. The LCD pins can only be controlled through the shift register on the PBMCUSLK board. The shift register must receive 8-bit words from SPI using the MOSI port, appropriately timed using the latch clock LCK and serial shift register SCK. The initialization procedure was taken directly from the course slides on interface examples [2] and processed inside the LCD initialization function since they are tightly related. The wiring to connect to the J5 connector was given on the course lab site on the tips page [1]. The GPIO initialization enabled PA4 as output mode, and PB3 and PB5 set to alternate function modes to utilize SPI. PB3 corresponds to SCK, PB4 to LCK, and PB5 to MOSI.

LCD Display

The LCD display was initialized in the myLCD_Init function. It first initializes SPI, and then follows the steps outlined in the interfaces examples slides [2]. First only the upper half of a command is sent to change the LCD to 4-bit mode, and then 4 more commands are sent in order to set information such as two-lines, 5x8 font, display on, cursor off, cursor blink off, auto increment cursor position, display shift off, and finally to clear the display to start. The LCD was accessed through the SPI. A function from the SPI library, SPI_SendData8 allowed us to send an 8-bit word to the LCD. Following the guides from the lecture slide examples in interfaces [2] we are able to create a few functions to assist with outputting to the screen.

The first function created was a simple mySPI_SendData function which was used to call the built in SPI_SendData8 function. This was more than just an alias, as it first forced the LCK signal to 0, waited until SPI1 was not busy, then called the SPI_SendData8, waited until SPI1 was ready again, then forced the LCK signal back to 1. Following that cycle, it would then call a function myTIM3_Wait which was used to create a delay between sending data to the LCD. The software and microcontroller can process data faster than the LCD can handle, so a delay had to be introduced to ensure we wouldn't flood the LCD with data and stop it from functioning correctly. TIM3 is further described in the next section, TIM3 Timer. The delay was set to 1ms.

The next two functions, myLCD_SendData and myLCD_SendCommand are very similar in nature. They both take an 8 bit char and send it to the LCD using the appropriate method. The lecture slides on interface examples [2] outlined how it was to be sent. Data had to be sent in 4-bit chunks, starting with the high order bits first, then the low order. Each 4 bit chunk was sent as the low end of an 8-bit section, with the high order being the RS and EN bits, followed by 2 unused bits. Each nibble had to be sent 3 times, toggling EN between 0/1/0. The RS bit changes how the LCD processes the instruction, so the myLCD_SendData and myLCD_SendCommand set it appropriately depending on if we are sending a character to display, or if you are sending an actual command to move the cursor or clear the screen etc.

The final function made was called mySendToLCD which takes in two parameters, frequency and resistance and writes out their values to the screen. The function calls myLCD_SendCommand to move the cursor to the first line and then calls myLCD_SendData to send each character to the screen. The characters are sent by their ASCII codes. For the numbers, each digit is extracted through division and modulus and then added to 48 to get the ASCII value for that digit since ASCII 48 = 0, and ASCII 57 = 9. The function then issues another command to move the cursor to the second line, then prints out the other numbers. The cursor does not have to be moved manually between columns since the auto increment cursor position flag was set during LCD initialization.

TIM3 Timer

The TIM3 timer was used for a time delay in between sending commands to the LCD. It was initialized in myTIM3_Init similarly to TIM2, using buffer auto-reload, count up, stop on overflow etc, but the timer does not generate interrupts. TIM3's prescaler value was set to 0xBB80/48000 such that with our clock speed of 48Mhz each tick would represent 1 millisecond. Using this, a function myTIM3_Wait was created which took in a parameter of how many milliseconds to wait for. The function clears the timer count value through the CNT register, sets the timeout value to the one given in the parameter, updates the register and starts the timer. It then waits until the timer hits the desired count, then clears the interrupt flag and stops the timer and finishing allowing execution of other methods to continue.

Wiring

The wiring for the project was done with assistance from the wiring diagrams found in the lecture slides [2]. An important one, the optocoupler and NE555 timer wiring diagram can be seen below in figure 2. Completed wiring circuits of the project can also be seen below in figures 3 and 4.

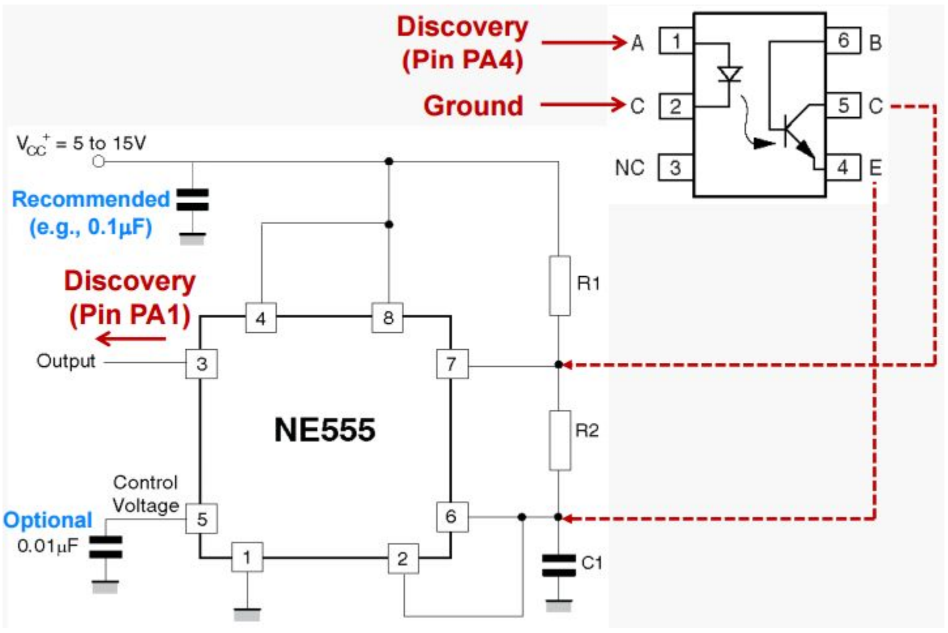

Figure 2. Optocoupler and NE555 Timer Wiring Diagram

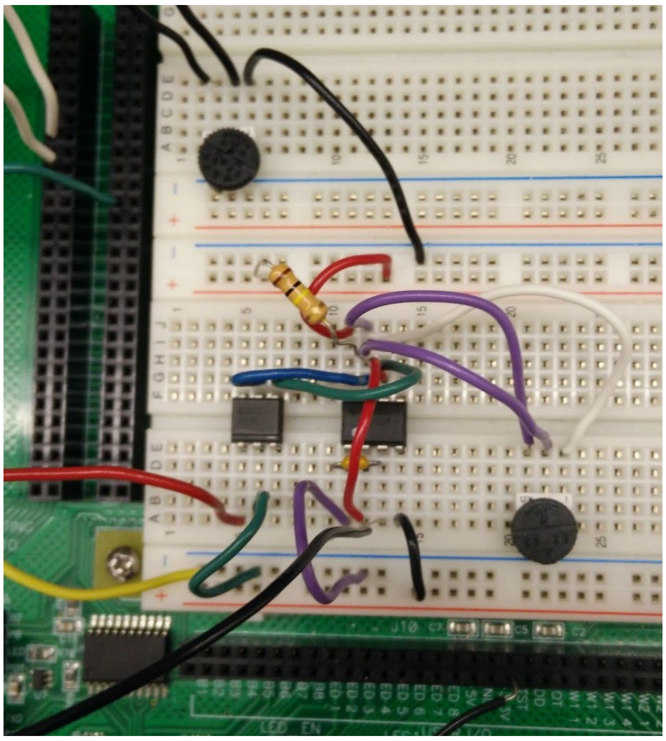

Figure 3. Project Wiring

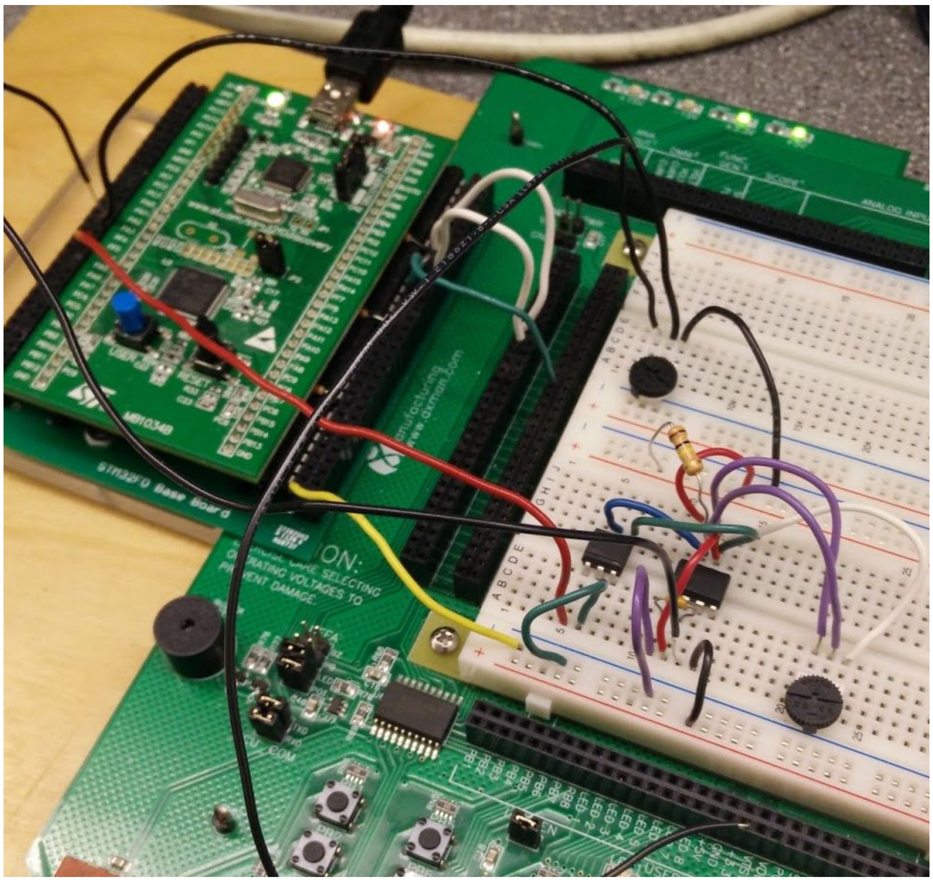
Figure 4. Project Wiring 2

Testing

The first part to be tested was the ADC and DAC. To start, It was known that the ADC was 12-bit, which meant its output would be between 0 and 4095, so values were outputted to the console to ensure that it was in this range. A potentiometer was connected to the ADC input to change the signal and notice the corresponding change in the console output. The resistance of the system was calculated by reading in the value of the ADC, dividing it by the max ADC value 4095 to get a ratio, then multiply it by the maximum potentiometer value, which was 5000. This gave values between 0 and 5000 depending on the potentiometers position. The resistance calculated was printed to the console to confirm it was reacting correctly to live changes.

Next, the ADC's input analog signal was checked with an oscilloscope to further understanding of the system. A picture of the oscilloscope output can be seen in figure 5. After that was confirmed to be working, it was simply moved the ADC's output value in ADC1->DR to the DAC->RHR12R1 which is the DAC data holding register in right aligned mode. This registers contents are moved internally and converted and outputted to DAC-OUT1 which is connected to pin PA4.

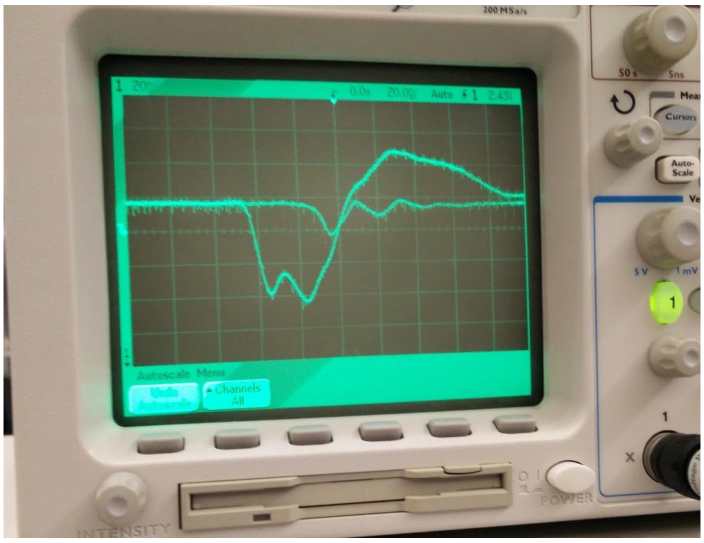
Figure 5. Oscilloscope Reading For Input Analog Signal

Once the ADC and DAC were confirmed to be working, the optocoupler and NE555 timer were added to the system. To evaluated if they were working as intended, and oscilloscope was added to read the output of the timer. The output of the timer should be a square wave, and the frequency should fluctuate depending on the surrounding resistances in the system. A second potentiometer was connected in as one of the resistors, which meant it was possible to change it on the fly to see that the waveform changes as the potentiometer changed. An output of the timer can be seen below in figure 6.

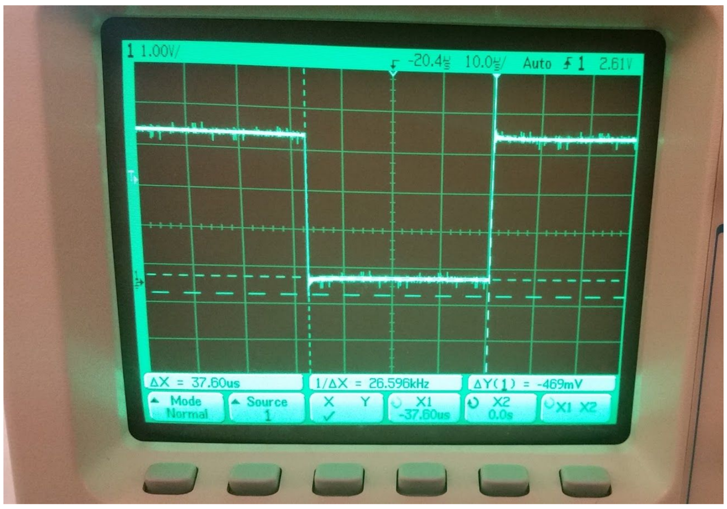

Figure 6. Oscilloscope Reading for NE555 Timer Output

Since it was known that the frequency calculations should work since it was the same code as lab 1 part 2 and the timer was outputting a square wave, it was trivial to see that it was working correctly. The calculated frequency value was outputted to the console and was confirmed to match the oscilloscopes reading. Now that the resistance and frequency were correctly being printed to the console, the final step was to get the LCD display working.

The LCD displays testing was more of a trial-and-error kind of method. Initially it was noticed that the screen was not clearing at all or giving any updates so it became clear that the initialization was likely not working correctly. After verifying that the commands being sent should be the correct ones, it was later realized that the GPIOB pins were not being enabled and initialized for use. After adding it to the myGPIO_Init function, the LCD then started to react to the code. Initially the screen would only flash, but have nothing persist on the screen. Even though it was just flashing, the fact that it initially cleared was a good sign since it indicated that it was at least partially working. Since being warned multiple times about the LCD speed, it was clear that before diving too deep into tearing apart the functions to send data to the screen that a delay between commands to the LCD should be added. Since the LCD screen is quite slow, if a delay is not used the LCD will get updates faster than it can process which will then break the program giving undesired results. After a delay function had been created using the TIM3 timer, a small delay was added to LCD updates and the screen began to function.

By this point a couple letters were being printed to the screen, and quite slowly. Logically at this point the next step was to decrease the delay to find the minimum value. After decrementing, then testing again a delay of 1ms was determined to still work. With a 1ms display, the output of the screen was now working, with only one small hitch being it was backwards, resistance on top, frequency on the bottom. This was an easy fix, the command to move the cursor to the first/second lines were just backwards. After setting this the LCD was now functioning correctly as can be seen figure 7. Further into testing we realized that occasionally after stopping the program and restarting it that the LCD screen would not function correctly only showing a line of gibberish text until the board was restarted. This was likely caused by the LCD being stopped while being in a state where it is waiting for the second half of a command, but never getting it. This would mean next time the program ran each command would be sort of “off-by-one”. This problem was determined to be acceptable since the program worked on a fresh board, and only happened sporadically, not every time. It was guessed that initialization changes may fix it, but the time to investigate that was not deemed worth it.

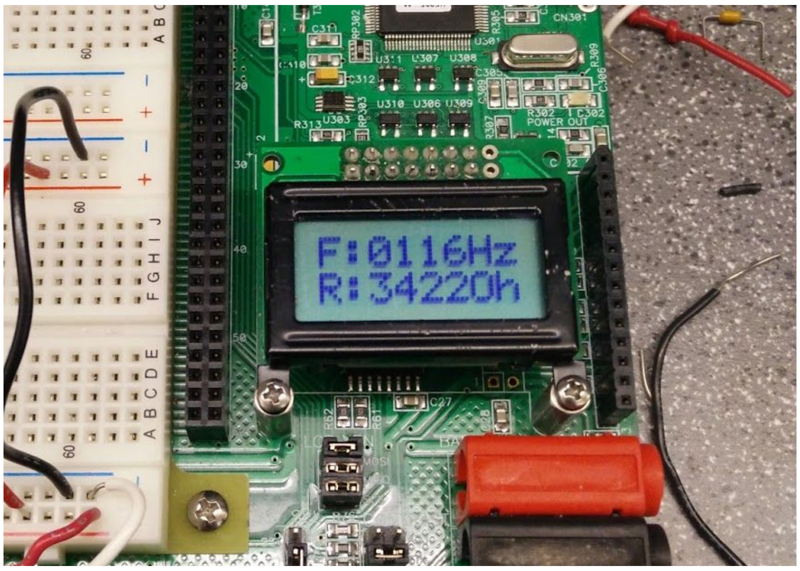

Figure 7. LCD Screen Output

Discussion

Work on the project was best done in long sessions. It was found that if you just stopped by for a short period of time that it was very inefficient since you had to spend a chunk of time setting up the lab environment and wiring it all. Some stations turned out to operate differently as well. One station I used the LCD screen did not appear to be functioning at all. I thought it may have been my code or wiring, but after a thorough investigation I found nothing wrong. After switching stations it began to work immediately. One other thing that varied largely between stations was the voltage output of the Microcontrollers 3.3V pin. Some stations had ~2.9V, some read around 3.1V, and some had the full 3.3V. In order to have consistent results, it was changed to use the 3.3V pin on the PBMCUSLK board since it consistently had a 3.3V output.

The LCD screens turned out to be a bit tedious when attempting to find an appropriate delay as a guess and check method was used. In the end, the final delay value chosen was the same across boards at least which was nice. Sometimes the LCD screen would fail to work if the code had been previously running and the board has not been restarted. The root of the cause is not known, but it can be guessed to be related to the state of the LCD. If the LCD was stopped in a half-state where its received half a command and is expecting the other half, then the future commands when the code resumes may be off-by-one in an essence, making it not work.

All in all, I was very pleased with the outcome of the project. Resistance values were read between a full range of 0-5000 Ohms, and our frequency varied between 107Hz and 118Hz depending on potentiometer values. The screen was seen to update multiple times per second, giving adequate feedback to changes. The updates could be sped up by shortening the delay on the LCD commands even further, and by removing printing to console as that eats up clock cycles.

Conclusion

This project, while time consuming, still proved to be an effective learning process. Microprocessor programming and embedded systems are something not heavily previously covered in the software engineering program, so this was a fresh challenge. The project would have to be considered a success, meeting the requirements in the lab manual. The system successfully measures the resistance and frequency of a pulse width modulated signal and displays the output on to the 4-bit LCD display. The system performs as expected, with the only further optimization being to correct the LCD problem with pausing and restarting the application that happened infrequently.

References

[1] D. Rakhmatov, “CENG 355 Lab Home Page“, [Online].
Available: http://www.ece.uvic.ca/~ceng355/lab/.

[2] D. Rakhmatov, “CENG 355 Microprocessor-Based Systems,” [Online].
Available: http://www.ece.uvic.ca/~daler/courses/ceng355/.

[3] “STM32F0 Reference Manual“, STMicroelectronics, [Online].
Available: http://www.ece.uvic.ca/~ceng355/lab/supplement/stm32f0RefManual.pdf

[4] T. van Roon, “555 Timer Tutorial”, [Online].
Available: http://www.sentex.ca/~mec1995/gadgets/555/555.html

[5] “Hitachi HD44780 LCD Display Controller Reference Manual”, Hitachi, Ltd., [Online].
Available: http://www.ece.uvic.ca/~ceng355/lab/supplement/HD44780.pdf

Code

Project code also attached to project report submission for viewing in preferred text editor.

```
// ----------------------------------------------------------
//  School: University of Victoria, Canada.
//  Course: CENG 355 "Microprocessor-Based Systems".
//  Developed by: Justin Richard
//  Project: CENG 355 Lab Project
// ----------------------------------------------------------
//
// ----------------------------------------------------------

#include <stdio.h>
#include "diag/Trace.h"
#include "cmsis/cmsis_device.h"

// 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"

// Defines
#define myTIM2_PRESCALER ((uint16_t) 0x0000)        // Clock prescaler for TIM2 timer: no prescaling
#define myTIM2_PERIOD ((uint32_t) 0xFFFFFFFF)       // Maximum possible setting for overflow
#define myTIM3_PRESCALER ((uint16_t) 0xBB80)        // Clock prescaler for TIM3 timer: 48Mhz/48K (0xBB80 = 48000) = 1Khz p
#define myTIM3_PERIOD ((uint32_t) 0xFFFFFFFF)       // Maximum possible setting for overflow

// SPI
SPI_InitTypeDef SPI_InitStructInfo;
SPI_InitTypeDef* SPI_InitStruct = &SPI_InitStructInfo;

// Declare Functions
void myGPIO_Init(void);
void myTIM2_Init(void);
void myTIM3_Init(void);
void myTIM3_Wait(int);
void myEXTI_Init(void);
void myADC_Init(void);
void myDAC_Init(void);
void myLCD_Init(void);

void mySendToLCD(int, int);
void myLCD_SendCommand(char);
void myLCD_SendData(char);
void mySPI_SendData(uint8_t);

// Declare global vars
float timerEnabled = 0;
float timerPulses = 0;
float frequency = 0;

int main(int argc, char* argv[]){
    trace_printf("Welcome to the project demo.\n");
    myGPIO_Init();     // Initialize I/O port PA
    myTIM2_Init();  // Initialize timer TIM2
    myTIM3_Init();  // Initialize timer TIM3
    myEXTI_Init();  // Initialize EXTI
    myADC_Init();   // Initialize ADC
    myDAC_Init();   // Initialize DAC
    myLCD_Init();   // Initialize LCD
    while (1){
        ADC1 -> CR = ((uint32_t) 0x00000004);        // Start of ADC conversion
        while((ADC1 -> ISR & 0x00000004) ==0);       // Wait for end of conversion
        DAC->DHR12R1 = ADC1->DR;                     // DAC input from ADC last conversion result
        mySendToLCD(frequency, ((ADC1->DR)*5000/4095)); // Resistance = ADC value * 5000 / 4095 (Max adc value)
    }
    return 0;
}

// Initialize General Purpose I/O pins
void myGPIO_Init(){
    RCC->AHBENR |= 0x00060000;                  // Enable clock for GPIOA and GPIOB
    // PA1 - Input
    GPIOA->MODER &= ~((uint32_t) 0x0000000C);   // Set as input
    GPIOA->PUPDR &= ~((uint32_t) 0x0000000C);   // Ensure no pull-up/pull-down
    // PA2 - ADC
    GPIOA->MODER |= ((uint32_t) 0x00000030);    // Set as analog
    GPIOA->PUPDR &= ~((uint32_t) 0x00000030);   // Ensure no pull-up/pull-down
    // PA4 - DAC
    GPIOA->MODER |= ((uint32_t) 0x00000300);    // Set as analog
    GPIOA->PUPDR &= ~((uint32_t) 0x00000300);   // Ensure no pull-up/pull-down
    // PB3 - SPI SCK
    GPIOB->MODER |= (GPIO_MODER_MODER3_1);   // Set as output
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR3);            // Ensure no pull-up/pull-down
    // PB5 - SPI MOSI
    GPIOB->MODER |= (GPIO_MODER_MODER5_1);   // Set as output
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR5);            // Ensure no pull-up/pull-down
    // PB4 - SPI LCK
    GPIOB->MODER |= (GPIO_MODER_MODER4_0);   // Alternating function
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR4);            // Ensure no pull-up/pull-down
}

// Initialize TIM2 timer
void myTIM2_Init(){
    RCC->APB1ENR |= ((uint32_t) 0x00000001);// Enable clock for TIM2 peripheral - Relevant register: RCC->APB1ENR
    TIM2->CR1 = ((uint16_t) 0x006C);        // Configure TIM2: buffer auto-reload, count up, stop on overflow, enable updat
    TIM2->PSC = myTIM2_PRESCALER;           // Set clock prescaler value
    TIM2->ARR = myTIM2_PERIOD;              // Set auto-reloaded delay
    TIM2->EGR = ((uint16_t) 0x0001);        // Update timer registers - Relevant register: TIM2->EGR
    NVIC_SetPriority(TIM2_IRQn, 0);         // Assign TIM2 interrupt priority = 0 in NVIC - Relevant register: NVIC->IP[3],
    NVIC_EnableIRQ(TIM2_IRQn);              // Enable TIM2 interrupts in NVIC - Relevant register: NVIC->ISER[0], or use NV
    TIM2->DIER |= ((uint16_t) 0x0001);      // Enable update interrupt generation - Relevant register: TIM2->DIER
}

// Initialize TIM3 timer
void myTIM3_Init(){
    RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;     // Enable clock for TIM3 peripheral - Relevant register: RCC->APB1ENR
    TIM3->CR1 = ((uint16_t) 0x006C);        // Configure TIM3: buffer auto-reload, count up, stop on overflow, enable updat
    TIM3->PSC = myTIM3_PRESCALER;           // Set clock prescaler value
    TIM3->EGR |= ((uint16_t) 0x0001);       // Update timer registers - Relevant register: TIM3->EGR
}

// Use tim3 to wait for a certain amount of clock cycles
void myTIM3_Wait(int ms){
    TIM3->CNT = ((uint32_t) 0x00000000);     // Clear the timer
    TIM3->ARR =  ms;                         // Set the timeout value from parameter
    TIM3->EGR |= ((uint16_t) 0x0001);    // Update registers
    TIM3->CR1 |= ((uint16_t) 0x0001);       // Start the timer
    while((TIM3->SR & 0x0001) == 0);            // Wait until timer hits the desired count
    TIM3->SR &= ~((uint16_t) 0x0001);       // Clear update interrupt flag
    TIM3->CR1 &= ~0x0001;                   // Stop timer (TIM3->CR1)
}

// Initialize extended interrupt and events controller
void myEXTI_Init(){
    SYSCFG->EXTICR[0] = ((uint32_t) 0x0000);// Map EXTI1 line to PA1 - Relevant register: SYSCFG->EXTICR[0] (PA2 = ADC, PA4
    EXTI->RTSR =        ((uint16_t) 0x0002);// EXTI1 line interrupts: set rising-edge trigger - Relevant register: EXTI->RT
    EXTI->IMR =         ((uint16_t) 0x0002);// Unmask interrupts from EXTI1 line - Relevant register: EXTI->IMR
    NVIC_SetPriority(EXTI0_1_IRQn, 0);      // Assign EXTI1 interrupt priority = 0 in NVIC
    NVIC_EnableIRQ(EXTI0_1_IRQn);           // Enable EXTI1 interrupts in NVIC
}

// Notes: ADC supply requirements: 2.4 V to 3.6 V
// Initalize analog to digital converter
void myADC_Init(){
    RCC->APB2ENR |= ((uint16_t) 0x0200);     // Enabled ADC clock
    ADC1->CFGR1 |= ((uint32_t) 0x00002000);  // Enable continuous conversion
    ADC1->CHSELR |= ((uint32_t) 0x00000004);    // Select channel to use to 2 - PA2 (PA1 = EXT1 line, PA4 = DAC)
    ADC1->SMPR |= ((uint32_t) 0x00000007);   // Set sampling time
    ADC1->CR |= ((uint32_t) 0x00000002);     // Ensure ADC disabled before calibration
    ADC1->CR |= ((uint32_t) 0x80000000);     // Start calibration
    while((ADC1->CR & 0x8000000) != 0x0);    // Wait for ADC calibration to complete
    ADC1->CR |= ((uint32_t) 0x00000001);            // Enable ADC
}

// Initialize digital to analog converter - Outputs to PA4 (PA1 = EXT1, PA4 = DAC)
void myDAC_Init(){
    RCC->APB1ENR |= ((uint32_t) 0x20000000);    // Enabled DAC clock
    DAC->CR = ((uint32_t) 0x00000001);               // Enable DAC on channel 1
    DAC->SWTRIGR |= ((uint32_t)0x1);         // DAC channel1 software trigger
}

// Initialize the LCD screen
void myLCD_Init(){
    trace_printf("Initializing LCD...\n");
    // Initialize SPI
    RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;
    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; // Test for appropriate values
    SPI_InitStruct->SPI_FirstBit = SPI_FirstBit_MSB;
    SPI_InitStruct->SPI_CRCPolynomial = 7;
    SPI_Init(SPI1, SPI_InitStruct);
    SPI_Cmd(SPI1, ENABLE);

    // Initialize the LCD screen step 1
    mySPI_SendData(0x02);  // 0000 0010 = 0x02 -> LCD Initialization 1 - H = 0010, RS = 0, EN = 0
    mySPI_SendData(0x82);     // 1000 0010 = 0x82 -> LCD Initialization 1 - H = 0010, RS = 0, EN = 1
    mySPI_SendData(0x02);     // 0000 0010 = 0x02 -> LCD Initialization 1 - H = 0010, RS = 0, EN = 0

    // Initialize steps 2-5 from slides
    mySPI_SendCommand(0x28); // 00 0010 1000 = 0x -> 4-bit interface = true, Two-lines = true, 5x8 Dots = true
    myLCD_SendCommand(0x0C); // 00 0000 1100 = 0x -> Display on = true, Cursor on = false, Cursor blink = false
    myLCD_SendCommand(0x06); // 00 0000 0110 = 0x -> Increment cursor position = true, display shift = false
    myLCD_SendCommand(0x01); // 00 0000 0001 = 0x -> Clear display
    trace_printf("Initialized LCD.\n");
}

// Print a given frequency and resistance to the screen
void mySendToLCD(int frequency, int resistance){
    // For debug:
    trace_printf("Frequency: %d hz, Resistance: %d Ohms\n", frequency, resistance);

    // Output to actual screen:
    myLCD_SendCommand(0x80);        // Select first line
    myLCD_SendData(0x46);           // Print F
    myLCD_SendData(0x3A);           // Print :
    myLCD_SendData((uint8_t) 48+((frequency/1000)%10));
    myLCD_SendData((uint8_t) 48+((frequency/100)%10));
    myLCD_SendData((uint8_t) 48+((frequency/10)%10));
    myLCD_SendData((uint8_t) 48+((frequency)%10));
    myLCD_SendData(0x48);           // Print O
    myLCD_SendData(0x7A);           // Print h

    myLCD_SendCommand(0xC0);        // Select second line
    myLCD_SendData(0x52);           // Print R
    myLCD_SendData(0x3A);           // Print :
    myLCD_SendData((uint8_t) 48+((resistance/1000)%10));
    myLCD_SendData((uint8_t) 48+((resistance/100)%10));
    myLCD_SendData((uint8_t) 48+((resistance/10)%10));
    myLCD_SendData((uint8_t) 48+((resistance)%10));
    myLCD_SendData(0x4F);           // Print H
    myLCD_SendData(0x68);           // Print z
}

// Takes a char and sends it to the LCD as a command for you - (RS = 0, EN 0/1/0)
void myLCD_SendCommand(char command){
    char h = ((command >> 4) & 0x0f);        // command = xxxx yyyy >> 4 = 0000 xxxxx, & 0x0f = 0000 xxxx = h
    char l = (command & 0x0f);               // command = xxxx yyyy & 0x0f                        = 0000 yyyy = l
    mySPI_SendData(0x00 | h); // 0000 0000 | h = 0000 xxxx
    mySPI_SendData(0x80 | h); // 1000 0000 | h = 1000 xxxx
    mySPI_SendData(0x00 | h); // 0000 0000 | h = 0000 xxxx
    mySPI_SendData(0x00 | l); // 0000 0000 | l = 0000 yyyy
    mySPI_SendData(0x80 | l); // 1000 0000 | l = 1000 yyyy
    mySPI_SendData(0x00 | l); // 0000 0000 | l = 0000 yyyy
}

// Takes a char and sends it to the LCD as data (RS = 1, EN 0/1/0)
void myLCD_SendData(char data){
    char h = ((data >> 4) & 0x0f);    // data = xxxx yyyy >> 4 = 0000 xxxxx, & 0x0f = 0000 xxxx = h
    char l = (data & 0x0f);           // data = xxxx yyyy & 0x0f                        = 0000 yyyy = l
    mySPI_SendData(0x40 | h); // 0100 0000 | h = 0100 xxxx
    mySPI_SendData(0xC0 | h); // 1100 0000 | h = 1100 xxxx
    mySPI_SendData(0x40 | h); // 0100 0000 | h = 0100 xxxx
    mySPI_SendData(0x40 | l); // 0100 0000 | l = 0100 yyyy
    mySPI_SendData(0xC0 | l); // 1100 0000 | l = 1100 yyyy
    mySPI_SendData(0x40 | l); // 0100 0000 | l = 0100 yyyy
}

// Sends data via SPI to LCD as per slide 24 of interfaces slides
void mySPI_SendData(uint8_t Data) {
    GPIOB->BRR |= GPIO_BRR_BR_4;        // Force LCK signal to 0
    while((SPI1->SR & SPI_SR_BSY) !=0); // Check if BSY is 0/SPI ready
    SPI_SendData8(SPI1, Data);              // Send the data via SPI
    while((SPI1->SR & SPI_SR_BSY) !=0); // Check if BSY is 0/SPI ready
    GPIOB->BSRR = GPIO_BSRR_BS_4;       // Force LCK signal to 1
    myTIM3_Wait(1);                         // Wait between LCD accesses since software too fast
}

// This handler is declared in system/src/cmsis/vectors_stm32f0xx.c - From template code
void TIM2_IRQHandler(){
    // Check if update interrupt flag is indeed set
    if ((TIM2->SR & TIM_SR_UIF) != 0){
        trace_printf("\n*** Overflow! ***\n");       // Print message to console
        TIM2->SR &= ~(0x0001);                       // Clear update interrupt flag
        TIM2->CR1 |= 0x0001;                         // Restart stopped timer
    }
}

// This handler is declared in system/src/cmsis/vectors_stm32f0xx.c - From lab 1 part 2
void EXTI0_1_IRQHandler(){
    if ((EXTI->PR & EXTI_PR_PR1) != 0){
        if (timerEnabled == 0){                     //  If this is the first edge:
            TIM2->CNT = 0x0000;                     //  Clear count register (TIM2->CNT)
            TIM2->CR1 |= 0x0001;                    //  Start timer (TIM2->CR1)
            timerPulses = 0;                        //  Clear pulse count
            timerEnabled = 1;                       //  Set triggered for next interrupt
        } else {                                    //  Else (this is the second edge):
            TIM2->CR1 &= ~0x0001;                   //  Stop timer (TIM2->CR1)
            timerEnabled = 0;                       //  Toggle triggered variable
            timerPulses = TIM2->CNT;                //  Read out count register (TIM2->CNT)
            frequency = SystemCoreClock / timerPulses;  //  Calculate signal frequency
        }
        EXTI->PR |= EXTI_PR_PR1;                    // Clear EXTI1 interrupt pending flag (EXTI->PR)
    }
}

#pragma GCC diagnostic pop
// ----------------------------------------------------------------------------
```