ECE 355 Lab Section B01
Final Report

Due Date: **November 29, 2021**

Group Members: **Joel Meuleman (V00919945)**

Michael Pillon (V00973605)

Marking

Problem Description/Specs	/ 5
Design/Solution	/ 15
Testing/Results	/ 10
Discussion	/ 15
Code Design/Documentation	/ 15
Total	**/ 60**

Table of Contents

Project Specifications

Objective

The objective of this project is to develop an embedded system based on the STM32F0 Discovery board which has the ability to measure the frequency of a PWM signal from either a function generator or a 555 timer. The purpose of this project is to gain an understanding for how to configure bare-metal microcontrollers and interface them with a variety of electronics.

Design Specifications

The system should be able to switch between measuring a PWM signal from a function generator and one generated by a 555 timer using the NE555 IC. A 4N35 optocoupler will be used to bias the 555 timer and will interface with the STM32F0 through a DAC output pin. The output from the STM320 into the optocoupler will be controlled by a potentiometer and an ADC pin on the STM320. Switching between the 2 signals will be done using a pushbutton and the interrupt functionality of the STM32F0. The currently active mode will be visible on the board based on whether the green or blue LED is on and the LEDs will swap when a pushbutton is pressed and the active mode is changed. The measured resistance value as well as the measured frequency of the active mode will be displayed on an LCD. A full functional block diagram of this system is shown in *Figure 1*.

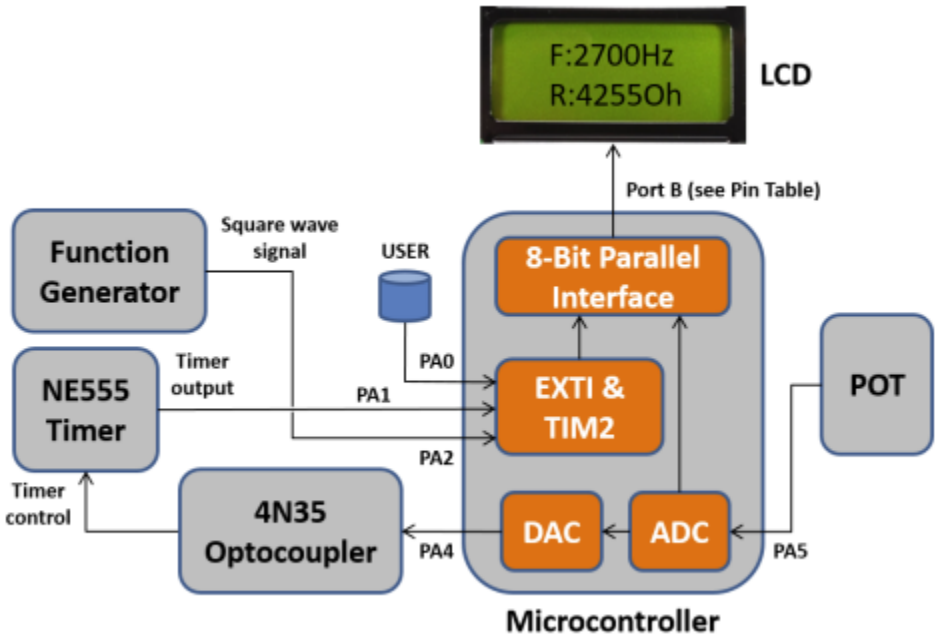


Figure 1: Overall System Diagram [1]

To simplify development and testing of this system, an independent program will be used to simulate and control the system in *Figure 1*. The application called "555 Timer Monitor" interfaces with the STM32F0 Discovery board and provides a virtual controller for the frequency generator, a monitor of the LCD pins, a simulation of the board, and a virtual LCD. The application is a standin for testing and if the discovery board is wired as per the table in *Figure 2*, a fully hardware based configuration should operate identically.

STM32F0	SIGNAL	DIRECTION
PA0	USER PUSH BUTTON	INPUT
PC8	BLUE LED	OUTPUT
PC9	GREEN LED	OUTPUT
PA1	555 TIMER	INPUT
PA2	FUNCTION GENERATOR	INPUT
PA4	DAC	OUTPUT (Analog)
PA5	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

Figure 2: Pin Assignments and functionality [1]

The software will be designed around the external interrupt functionality or EXTI of the STM32F0. After initializing the hardware, the main function of the program will consist of an empty infinite loop which exists to keep the program running while waiting for interrupt signals. 3 devices will be configured to create interrupts: a pushbutton, a 555 timer, and a function generator.

The pushbutton and 555 timer will be configured using the native *EXTI0_1_IRQHandler()* function which is a part of the Common Microcontroller Software

Interface Standard or CMSIS. When the STM32F0 detects an interrupt on EXTI0 or EXTI1, it enters the IRQ handler and the program checks which device triggered the interrupt. If the pushbutton is pressed, the system switches the active mode and resets necessary flags and registers. If the 555 timer triggers the interrupt, the system will call the *calculateFrequencyFromTIM2()* subroutine. Similarly, the function generator will be configured on EXTI2 and will use the *EXTI2_3_IRQHandler()* function to follow the same flow as the 555 timer.

A more in depth description of the software configuration and methods is given in the design section of this manual and a block diagram describing the overall program architecture is available in *Figure 3*.

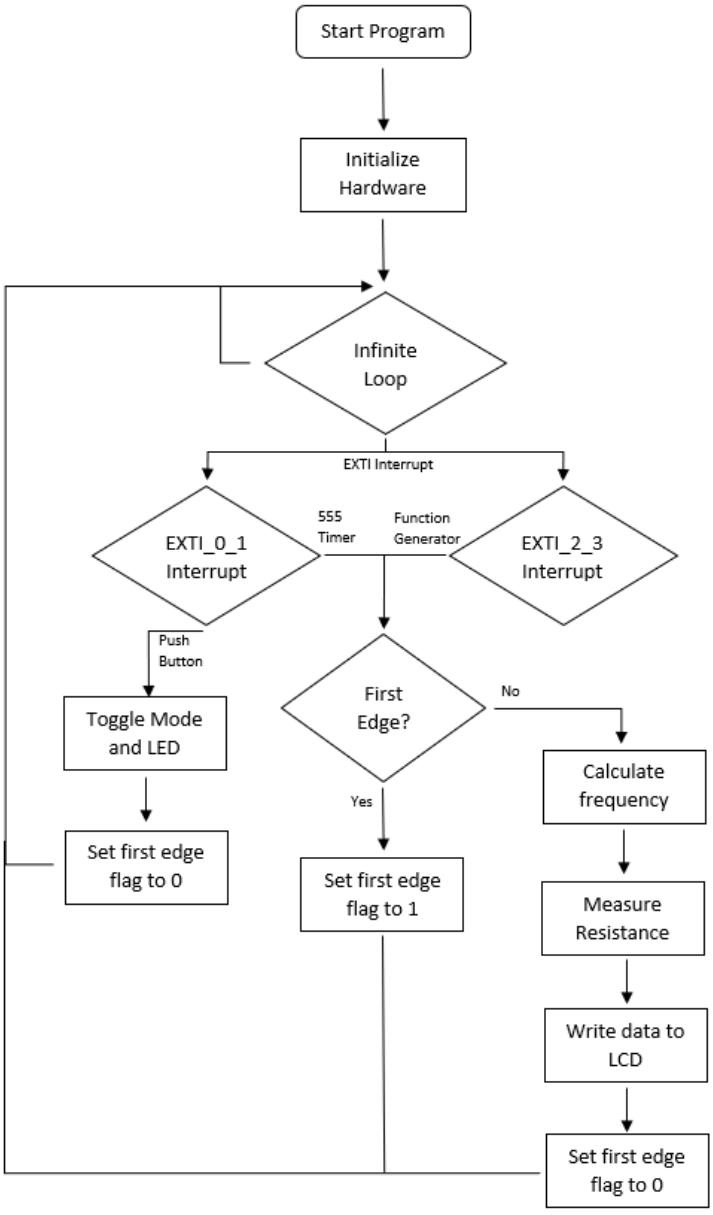


Figure 3 - Program flow block diagram describing the software of the system

Design

From the start, it was recognized that identifying and separating the project into its functional components would be the best way to proceed. By modularizing the project, code could be worked on independently before being integrated into the system and provides reusability for future projects. This modularization of the C code was done using header files. *Figure 1* shows the functional components of the project.

The design was divided into the following modules:

- The input push button that toggles the active external interrupt and the corresponding indicator LEDs
- The External Interrupt Controller and Timer to measure input frequency
- The parallel port I/O to write measured data to the LCD
- The Analog to Digital Converter to read the potentiometer resistance.
- The Digital to Analog Converter to drive the optocoupler to set the frequency of the 555 timer.

Push Button / LEDs / Interrupts

The core of this project is based around external interrupts by the push button, frequency generator, and 555 timer. The current mode of operation is stored in the global variable, *activeMode*, and uses the pin defines *MODE_FUNCTION_GENERATOR* and *MODE_555_TIMER* for its states and describes whether the PWM frequency will be calculated for either the function generator or 555 timer.

Configuring the Push Button

The push button was configured to be connected to pin PA2 and the configuration was handled by the *GPIOA_init()* function. First, the clock signal was enabled for port A by turning on the bit stored in *RCC_AHBENR_GPIOAEN* in the *RCC->AHBENR* register. Next, PA2 is configured as an input by ensuring the *GPIO_MODER_MODER2* bits of the *GPIOA->MODER* register are turned off. Lastly, the pushbutton is configured with internal no pull up or pull down resistors by turning off the *GPIO_PUPDR_PUPDR2* bits of the *GPIOA->PUPDR* register.

Configuring the LEDs

The LEDs were configured on pins PC8 and PC9 on the STM32F0 Discovery board and consist of a blue and a green surface mount resistor. Both LEDs are configured with physical resistors as active high, in other words the LEDs turn on when a 1 is written to the corresponding pin.

The first step in configuring the LEDs is to ensure the clock for port C is turned on by enabling the *RCC_AHBENR_GPIOCEN* bit of the *RCC->AHBENR* register. Next, the 2 pins are set as outputs by setting the corresponding *GPIO_MODER_MODERx_0* bit of the GPIOC->MODER register. The output type for both LEDs was set to push-pull through the *GPIOC->OTYPER* register, the pins were set to high-speed mode through the *GPIOC->OSPEEDR* register, and the pins were configured with no pull-up or pull-down resistors through the *GPIOC->PUPDR* register. Further details about the LED configuration can be seen in Appendix A or in the STM reference manual [2].

Configuring the Interrupts

To map a pin to an EXTI line, the appropriate bits of the *SYSCFG->EXTICR* register. The EXTI0 line was mapped to PA0 for the pushbutton, the EXTI1 line was mapped to PA1 for the 555 timer, and the EXTI2 line was mapped to PA2 for the function generator. All the interrupts were set to a rising-edge trigger by turning on the *EXTI_RTSR_TRx* bit of the *EXTI->RTSR* register.

To initialize the program, the *EXTI->IMR* register is used to enable the push button interrupts and the function generator through the constants mapped to the *EXTI_IMR_MR0* and *EXTI_IMR_MR2* bits. *EXTI_CONTROL_BIT_x* constants are used to make the code more readable. The *NVIC_SetPriority()* function is called to set *EXTI0_1_IRQn* to a priority of 0, and *EXTI2_3_IRQn* to a priority of 1 so that the pushbutton will always have the highest priority regardless of which signal is being measured and triggering interrupts. The *EXTI0_1_IRQn* and *EXTI2_3_IRQn* lines are then enabled using the *NVIC_EnableIRQ()* function.

Interrupt Handling Routines

The *EXTI0_1_IRQHandler()* and *EXTI2_3_IRQHandler()* functions are called whenever an interrupt is called on either EXTI0/EXTI1 or EXTI2/EXTI3 respectively. Since EXTI0 and EXTI1 are both used in this system, the first step of the subroutine is to identify which line triggered the interrupt and this is done by checking which bits are set in the *EXTI->PR*. The *EXTI_PR_PRx* bits are stored in global constants as *PENDING_REGISTER_x* with the appropriate hardware name that triggered the interrupt. If a pending register bit is found to be set, after handling the interrupt the pending register is reset to 0 by writing a 1 to the *PENDING_REGISTER_x* bit to the *EXTI_PR_PRx register.*

The *PENDING_REGISTER_PUSH_BUTTON* bit is checked first to ensure that if the push button is pressed, it takes priority over the 555 timer in the *EXTI0_1_IRQHandler()* function. If the push button had triggered the interrupt, the program waits for the button to be released before moving on. The timer is disabled and the first edge flag is reset to 0 to ensure that after the mode of the program is changed, the first reading is accurate. The currently active LED is turned off using the *GPIOC->BRR* register, the *activeMode* register is switched to the new mode, and the appropriate LED for the new mode is turned on using the *GPIOC->BSRR* register.

The *EXTI->IMR* register is then used to enable interrupts on the pushbutton and the new mode while disabling interrupts for the previously active mode.

When an interrupt is detected in the *EXTI_PR_PRx* register for either the 555 timer or the function generator, the *calculateFrequencyFromTIM2()* function is called and handles the appropriate signal. This function uses the *activeMode* register to measure frequency on the appropriate pin.

Frequency Measuring with TIM2

To measure the frequency of input waveforms, general purpose timer TIM2 was used. By measuring the time between rising edges, frequency can be calculated. When a rising pulse edge was detected, TIM2 was started. When the next rising pulse edge was detected, the value TIM2 had counted to was recorded, and the frequency conversion made.

Configuring TIM2

The configuration of TIM2 involved many registers. First, the clock for TIM2 was enabled. Second, TIM2 was configured for count up mode by setting the appropriate *Center-Aligned Mode Selection* bits in the *TIM2 Control Register 1*. Third, the *Timer2 Event Generation Register* was configured to correctly reinitialize itself by setting the *Update Generation* bit. Fourth, the priority of the TIM2 interrupt was configured as highest priority in the Nested Vector Interrupt Controller (NVIC) by calling the corresponding function. Also in the NVIC, the TIM2 interrupt was enabled. Lastly, interrupts were enabled by setting the *Interrupt Enable* bit in the *TIM2 DIER* register.

Measuring Frequency

There are 2 states of the system when measuring frequency: the timer is not running and the timer is running. When the *calculateFrequencyFromTIM2()* function is called, the *firstEdgeFlag* is checked to determine which state the system is in. If the *firstEdgeFlag* is equal to 0, it is the first time the function is being called so the timer count is reset to 0, the *firstEdgeFlag* is set to 1, and the timer is enabled. When the system returns to the *calculateFrequencyFromTIM2()* method, the timer is immediately stopped to maximize accuracy and the *TIM2->CNT* register is read. This register is incremented once for every rising edge of the system clock which is available in the *SystemCoreClock* global variable. The count divided by the clock frequency gives the measured frequency of the signal. This value is then passed to the *writeToLCD()* function which measures the potentiometer and outputs both data points to the LCD as formatted strings. The *firstEdgeFlag* is then reset to 0.

LCD

Measured data was displayed using a simulated LCD modelled after the Hitachi HD44780. The simulated LCD was used to display measured frequency from a selectable source in addition to measured potentiometer resistance. The GPIOs of port B were used to interface the LCD to the microcontroller.

Configuring the STM32F0 for LCD Interfacing

Before the LCD can be directly configured, the interfacing GPIOs must be set up. Port B pins 4-6 and 8-15 were configured as output pins by setting the corresponding bits in the port B *Port Mode Register*. Pins 4-6 were designated as control pins, while Pins 8-15 were designated as data pins. These pin designations are seen in the LCD.h header file in Appendix E.

The output mode bits of the *Port Mode Register* were set for each pin using the *GPIO_MODER_MODERx_0* define as found in the STM32F0 header file. All port B I/O pins were configured for the highest speed by setting all bits in the *GPIO Port Output Speed Register*.

Configuring the LCD

To send commands and data to the LCD, a procedure is defined in the HD44780 datasheet and LCD interfacing slides provided in class[4][6]. This procedure is abstracted into a function lcd_write(uint16_t data) to make for simple writing of a data byte with relevant control bits to the LCD. To simplify the sending of commands, GPIO pins corresponding to the data and control bits of the LCD were defined as seen in Appendix E.

1. The LCD data bits DB0-DB7 are set with the data to be sent.
2. The LCD enable bit EN is set and the LCD reads the data lines.
3. The LCD busy/handshake bit HANDSHAKE is polled until it is asserted, meaning the LCD has begun instruction execution[6].
4. The LCD enable bit EN is deasserted.
5. The LCD busy/handshake bit HANDSHAKE is polled until it is deasserted, meaning the LCD has completed the last instruction[6].
6. The data bits are reset

```c
void lcd_write(uint16_t data)
{
  GPIOB->BSRR |= data; /* Write data to pins */
  GPIOB->BSRR |= EN;
  while(!(GPIOB->IDR & HANDSHAKE));
  GPIOB->BSRR |= EN << 16;
  while((GPIOB->IDR & HANDSHAKE));
  GPIOB->BSRR |= data << 16;
}
```

Figure 4: LCD Write Function

Toggling of the command and data bits are done using the *GPIO Bit Set/Reset Register*. A write to bits in the range [15:0] sets the corresponding output pin high. A write to bits in the range [31:16] sets the corresponding bit in the range [15:0] low. Handshaking is performed by polling input pin 7 of port B to check if it is set (instruction execution in progress) or unset(no instruction in progress). With a function to transfer data to the LCD, commands can be given and configuration can proceed.

To display data correctly, several commands are given to the LCD. All commands are given as some combination of set data bits, as defined in the HT44780U datasheet. First, the LCD was configured for 8-bit mode (as it is interfaced by 8 GPIOs), both lines of the LCD were enabled, and a character size of 5x8 pixels was selected. The data bits corresponding to this command were DB3, DB4, and DB5. Next, the display was enabled, the cursor disabled, and blink-mode disabled. The data bits corresponding to this command were DB3 and DB2. Next, auto-increment was enabled, and the direction of the increment was set (this makes accessing DDRAM more convenient when attempting to display characters). The data bits corresponding to this command were DB1 and DB2. Finally, the display was cleared by setting DB0. Summarized, the display is configured for: 8-bit interfacing, two-line display, no cursor nor character blinking, and auto increment DDRAM address when writing successive characters.

Instruction	RS	R/$\overline{W}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Clear display	0	0	0	0	0	0	0	0	0	1
Return home	0	0	0	0	0	0	0	0	1	—
Entry mode set	0	0	0	0	0	0	0	1	I/D	S
Display on/off control	0	0	0	0	0	0	1	D	C	B
Cursor or display shift	0	0	0	0	0	1	S/C	R/L	—	—
Function set	0	0	0	0	1	DL	N	F	—	—
Set CGRAM address	0	0	0	1	ACG	ACG	ACG	ACG	ACG	ACG
Set DDRAM address	0	0	1	ADD	ADD	ADD	ADD	ADD	ADD	ADD
Read busy flag & address	0	1	BF	AC	AC	AC	AC	AC	AC	AC

Figure 5: Table of LCD Commands[6]

Writing Data to the LCD

Once the LCD display formatting was complete, data could be written to it. The LCD displays data by receiving a given address from its data bus, and displaying the character stored in internal DDRAM at that address. Conveniently, character addresses in DDRAM are the same as the ASCII codes of those same characters. This means that to display any ASCII character all that needs to be done is to set the corresponding data bus pins to the ASCII code of that character. As pins 8-15 (DB0-DB7) were designated as the data bus pins for interfacing the LCD, to put an ASCII character on the bus was a simple bit shift when sending the 16-bit data integer to lcd_write(uint16_t data). In addition to the data bits, the data indicator bit (RS) must also be set to indicate a data write rather than a command. The line of the LCD to write also had to be selected by setting DB7 for line 1 and DB6 and DB7 for line 2.

Being able to efficiently write changing strings of data to the display was an important aspect of this project. To accomplish this, two functions were written, the one used depending on the line of the display to be written. These functions, lcd_display_line1(char *data) lcd_display_line2(char *data) take a reference to a string, the data to be displayed, and iterate over the length of the string, calling lcd_write(uint16_t data) on each character iterated over. The remaining spaces not filled by characters from the input data are written with blank space. Filling the unused spaces with empty spaces proved to be a preferable solution to clearing the entire display upon one new reading.

```
void lcd_display_line1(char *data)
{
  /* Select line 1 */
  lcd_write(LINE_1);
  unsigned int i = 0;
  /* Write data */
  for(; i < strlen(data); i++){
    lcd_write(RS | data[i]<<8);
  }
  /* Ensure the rest of the line is empty */
  for(; i < LCD_LINE_LENGTH; i++){
    lcd_write(RS | ' ' << 8);
  }
}
```

*Figure 6: lcd_display_line1(char *data)*

Analog to Digital Converter

The analog to digital converter (ADC) of the STM32F0 was used to measure the resistance of an external user-set potentiometer. The voltage measured by the ADC was then mirrored by the digital to analog converter (DAC) which in turn drove the 555 timer.

Configuring the ADC

Configuration of pin A5 for analog to digital conversion is identified as two separate tasks: configuration of the I/O port as an analog input, as well as configuration of relevant ADC registers. Beginning with the I/O port configuration, the peripheral clock is enabled on Port A. Next, the Port A *Port Mode Register* register is configured. To use GPIO pin A5 as an analog input, the corresponding *Analog Mode* bits must be set in said register. For pin 5, these bits in the

Port Mode Register register are set by *GPIO_MODER_MODER5*, as defined in the STM32F0 header file.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y+1:2y **MODERy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O mode.
00: Input mode (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

Figure 7: GPIO Mode Set Register

Next, configuration of the main ADC registers is considered. For this project, maximum accuracy in the reading of the potentiometer is desired. Accuracy in the ADC reading is directly proportional to the selected resolution, so the highest resolution of the STM32F0 was selected, 12 bits. 12 bits is the default resolution set in *ADC Configuration Register 1*. Additional bits set in the prior register are the *Overrun Management Mode* bit which allows old data to be overwritten if new data is available, and the *Continuous Conversion Mode* bit which forces ADC conversion to run continuously during program execution.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	AWDCH[4:0]					Res.	Res.	AWDEN	AWDSGL	Res.	Res.	Res.	Res.	Res.	DISCEN
	rw	rw	rw	rw	rw			rw	rw						rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AUTOFF	WAIT	CONT	OVRMOD	EXTEN[1:0]		Res.	EXTSEL[2:0]			ALIGN	RES[1:0]		SCANDIR	DMACFG	DMAEN
rw	rw	rw	rw	rw			rw			rw	rw		rw	rw	rw

Figure 8: ADC Configuration Register

Relevant to the resolution of the ADC, the *Sample Rate* register is configured for maximum sampling time (239.5 clock cycles)[2]. A lengthy sample time is desirable for an accurate reading as it gives the internal capacitor time to charge to and hold the input voltage [3]. Then, the *Channel 5* bit (as pin A5 is being read) of the *ADC Channel Selection Register* is set, which configures the channel for conversion. Finally, after configuration of all other registers are complete, the *ADC Enable* bit is set in the *ADC Control Register*, as well as the *Analog to Digital Conversion Start* bit starting conversion. The *Analog to Digital Conversion Start* bit must be set only once configuration is complete as specified in the user manual.

Reading the ADC

Reading raw data from the ADC is done using a polling approach. To check for a new valid reading, the *End of Conversion Flag* of the *ADC Interrupt and Status Register* is polled. The *End of Conversion Flag* bit is set when a conversion is finished and new data is available in the *ADC Data Register*[2]. This flag is automatically reset upon reading the data register contents. The data register contents are converted to a voltage reading using *Equation 1* [2].

$$V_{CHANNELx} = \frac{V_{DDA}}{FULL\_SCALE} \times ADC\_DATA_x$$

Equation 1: Channel Data to Channel Voltage

The reference voltage, V_{DDA}, of *Equation 1* is equal to ~3.3V as specified in the STM32F0 datasheet and user manual. The scale factor of *Equation 1* is equal to the configured resolution of the ADC, $2^{12} - 1$. To read the resistance set on the external potentiometer, a similar equation is used, where the resistance is calculated as the ratio of ADC input data to maximum resolution as seen in *Equation 2*.

$$Potentiometer\ Value = \frac{Maximum\ Potentiometer\ Resistance}{Full\ Scale\ Resolution} * ADC\ Data$$

Equation 2: Channel Data to Resistance Reading

The *Maximum Potentiometer Resistance* of *Equation 2* is equal to the maximum value of the potentiometer, 5kΩ.

Digital to Analog Converter

The Digital to Analog Converter (DAC) of the STM32F0 was used to control the frequency output of the 555 timer by providing a voltage varying from 0V to 3.3V as read from the ADC.

Configuring the DAC

Configuration of pin A4 for digital to analog conversion is identified as two separate tasks: configuration of the I/O port as an analog output, as well as configuration of relevant DAC registers. Beginning with I/O port configuration, the peripheral clock is enabled on port A; the DAC clock is also enabled. Pin A4 is set to *Analog Mode* in the Port A *Port Mode Register* register by setting the corresponding bits defined by *GPIO_MODER_MODER4* in the STM32F0 header file. Next, the DAC control registers are configured. The only configuration needed is to enable the DAC, by setting the *DAC Channel1 Enable* bit in the *DAC Control Register*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	DMAUDRIE2	DMAEN2	MAMP2[3:0]				WAVE2[1:0]		TSEL2[2:0]			TEN2	BOFF2	EN2
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	DMAUDRIE1	DMAEN1	MAMP1[3:0]				WAVE1[1:0]		TSEL1[2:0]			TEN1	BOFF1	EN1
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figure 9: DAC Control Register

Writing and Reading the DAC

The output voltage of the DAC is set by writing to an intermediate register, whose value is then shifted to the output register after a clock cycle. First, the *ADC Data Register* is read; this voltage reading is the voltage to be output from the DAC. Second, the *ADC Data Register* contents are written to the DAC intermediate data holding register. For simplicity, *DAC Channel 1 12-bit Right-Aligned Data Holding Register* is selected for writing to eliminate any bit shifting and scaling that would be needed if a 8 or 12 bit left-aligned register were selected. To check that the correct voltage is being output, the *DAC Channel 1 Data Output Register* is read and the data made readable by *Equation 3*. V_{DDA} is the same reference voltage as referenced in *Equation 1*; 4095 is the resolution of the DAC which is the same as the ADC.

$$\text{DACoutput} = V_{DDA} \times \frac{DOR}{4095}$$

Equation 3: Converting DAC Output Data to a Voltage Reading [2]

Testing and Results

Push button, LEDs, and interrupts

Testing for this part of the project was relatively straightforward as a button press should toggle the system between reading the function generator and reading the 555 timer while toggling the LED according to the specific mode of operation. The user button in the 555 timer application was pressed repeatedly and the status LED could be observed to be changing in both the application as well as on the physical board. When the user button was pressed it was also observed that the frequency measurements on the LCD would switch between the value of the function generator to the value of the 555 timer and vice versa. As a part of testing in the early stages of development, *trace_printf()* statements were used to indicate which mode was switched to and these print statements were left in the final version of the code.

Resistances, the ADC, and the DAC

The potentiometer readings were tested by adjusting the resistance in the 555 timer application from the minimum value of 0Ω to the maximum value of 5kΩ. These are the theoretical limits of the potentiometer however both are subject to practical limitations. The 0Ω minimum value is limited by any resistances in the terminals and traces however should likely be very close to 0Ω. The maximum reading of 5kΩ also would be increased by terminal and trace resistances however is also subject to variability during the manufacturing process. There was no way to confirm this variability since the potentiometer was already integrated into the system.

When the ADC read values from the potentiometer the actual resistance limits were roughly 4935Ω and 68Ω as seen in *Figure 10* and *Figure 11*.

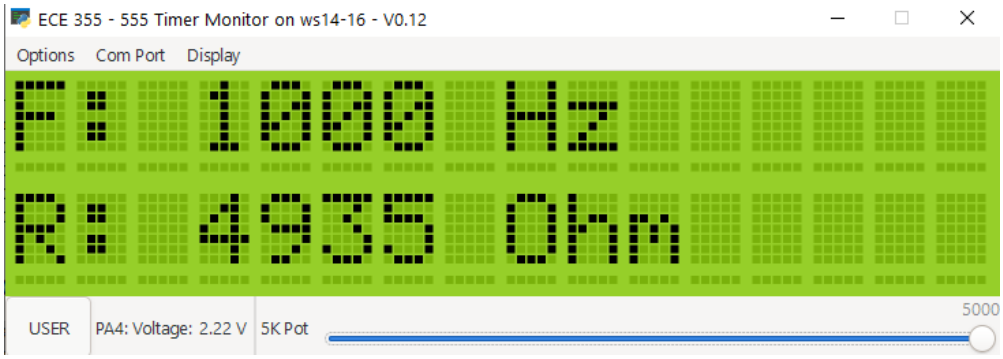


Figure 10 - Maximum measured resistance value

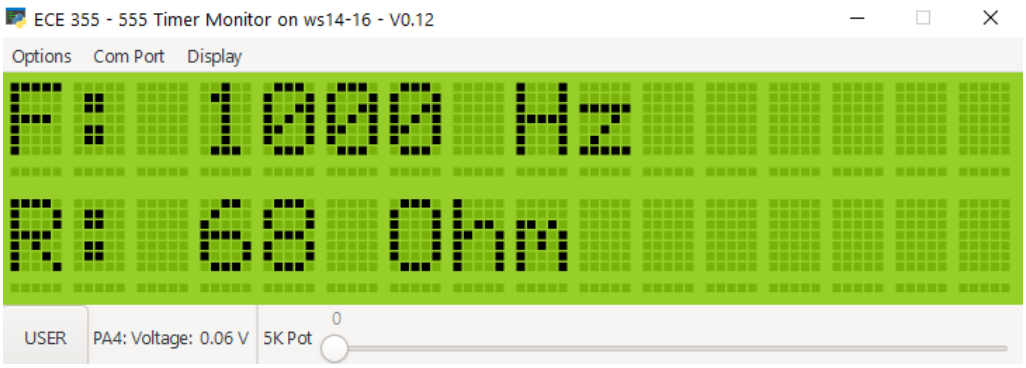


Figure 11 - Minimum measured resistance value

The minimum value resistance is attributed to additional resistances in the circuit as well as some practical limitations of the ADC. The resolution of resistance stems from the 12-bit ADC resolution and is equal to $\frac{5000\Omega}{2^{12}} = 1.22\Omega$ and is the minimum reading between 2 measurements. The accuracy of the system was a larger factor and the resistance measurements on the LCD could be seen varying by roughly ±50Ω which corresponds closely to the observed minimum and maximum readings.

During development, the *dac_read()* function was used to test the DAC and returns the value that is currently being output by the DAC. After the ADC value or any 12-bit value was written to the DAC, the read function was used to verify that the written value was being stored.

Frequency Measurements

Testing for this project consisted of measuring frequencies from 2 separate sources, a function generator and a 555 timer. These sources used separate input pins however both of them used TIM2 for calculation of frequency measurements. TIM2 was configured with no prescaler so the maximum theoretical frequency achievable was the period related to 1 clock pulse or 48MHz. Practically however the maximum readable frequency was much lower at just over 500KHz, see *Figure 12*.

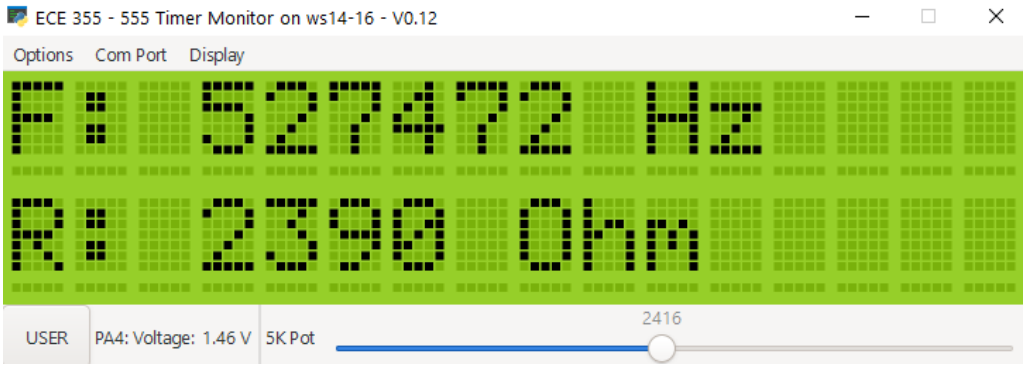


Figure 12 - Maximum practical frequency reading

This result was achieved by setting the frequency generator to 10MHz and observing the output on the LCD. Since the frequency was well above the practical limit of the system, the observed frequency was a result of the system responding as quickly as possible and was consistent as frequency was increased to any value above this practical limit.

This practical limit exists for a number of reasons but is mostly due to the overhead required to compute frequency. The interrupt routine as well as all the time required to process the code takes up time meaning that the interrupt routines, starting the timer, and stopping the timer are all accounted for when measuring frequency. Furthermore, all testing was done while using a debugger which has extra overhead compared to programming the STM Discovery board and having it run independently. These effects were noticeable in the practical limit of the frequency readings but also showed up for significantly lower frequency readings. *Figure 13* shows these effects as a 10kHz signal being generated by the function generator displays a value of 10012Hz and this discrepancy can be attributed to the same computation times which limit the maximum frequency readings. As frequencies are increased, these effects have a greater effect and at low frequencies these effects are negligible and don't show up for frequencies of a few kHz or lower.

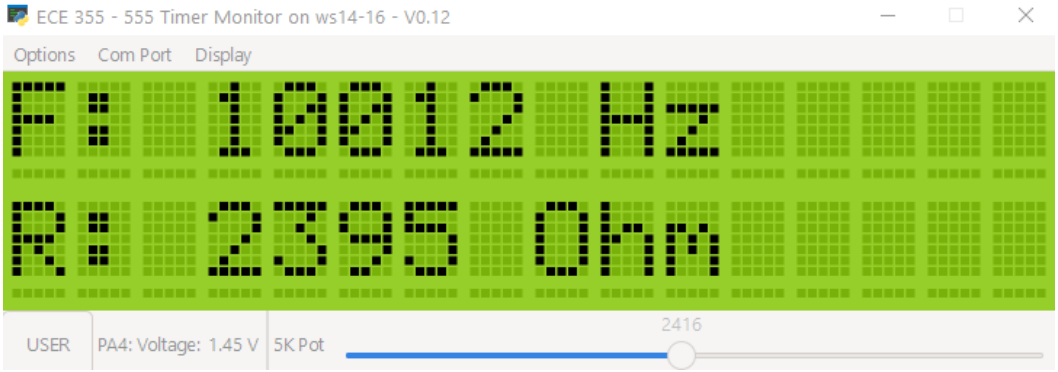


Figure 13 - A 10kHz frequency reading showing 12Hz of error due to computation time

The theoretical minimum for our frequency measurements was based on the resolution of TIM2. There was no prescaler value set and TIM32 is a 32-bit counter meaning the theoretical minimum frequency was $2^{32} = 233\ x\ 10^{-12}\ Hz$ which gives a period of roughly 90 seconds.

555 Timer vs Frequency Generator

The theoretical and practical limitations directly applied to the frequency generator which has a much wider range of frequencies it could produce, however the 555 timer has a significantly smaller range of operation. The 555 timer works based on the configuration of external resistances and capacitances which give reference voltages at some of its pins. This project had limited flexibility in the range of these voltages since the DAC and optocoupler were limited to operating between 0V and 3.3V. This configuration gave a practical limit to the

frequency output of the 555 timer to between roughly 1kHz and 1.5kHz. There was also a limited effective range for the potentiometer as only about one third of the resistance range was observed to have any effect on the frequency produced by the 555 timer. These limits are available in *Figure 14* and *Figure 15*.

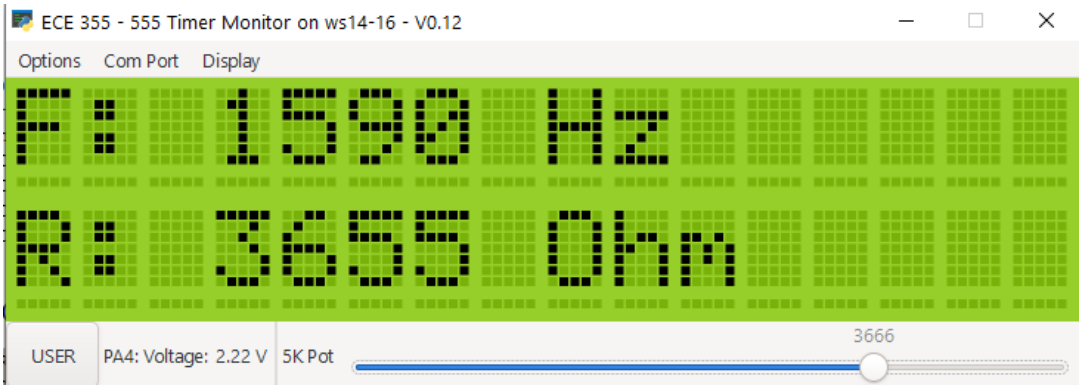


Figure 14 - Maximum frequency of the 555 timer achieved at the minimum resistance

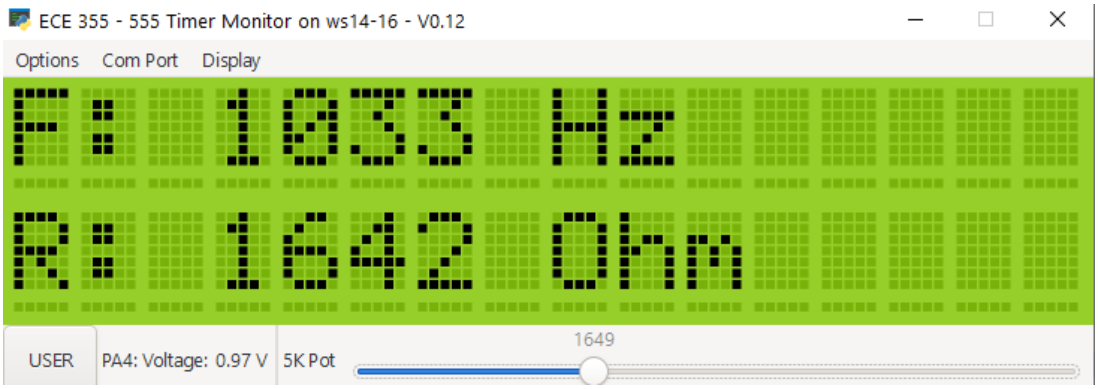


Figure 15 - Minimum frequency of the 555 timer achieved at the maximum resistance

When varying the potentiometer from 0Ω to 5kΩ, the 555 timer only varied from roughly the 1.6kΩ to 3.6kΩ range but remained relatively constant when adjusting the potentiometer outside of this range. The constant variations in the ADC readings were also observed as similar variations in the 555 timer frequency of less than 5% of the measured frequency.

Discussion

Design

Our design was able to meet all specified project requirements: external frequencies were accurately measured and able to be toggled between, the potentiometer value was accurately read at all values (excluding 0Ω) with a relative error of less than 1%, modifying the DAC output voltage successfully varied the frequency output by the 555 timer, and the LCD correctly displayed the measured data with seamless transitions between values.

Assumptions

A first assumption is that the PWM input frequency falls within the accurately measurable frequency range of the microcontroller, which was determined to be 1Hz to ~500 kHz. A second assumption was that the internal reference voltage, *VDDA*, was 3.3V. This value should be very close to correct and is spec for an operating temperature of 30℃[2]. As this reference voltage is used in calculations of input voltage from the ADC and output voltage of the DAC, it is possible it introduced a small source of error in input and output voltage readings. The precise value of *VDDA* can be calculated from the *VREFINT_CAL* and *VREFINT_DATA* registers but was deemed unnecessary. Per the datasheet, the precise value of *VDDA* should only differ from 3.3V by a factor of millivolts[2].

Shortcomings

The most notable shortcoming was the small bandwidth of frequencies output from the 555 timer, 1kHz to 1.5kHz. The frequency was observed to change over a DAC output voltage of approximately 1V to 2.2V. Without knowing the exact circuit values used in the construction of the 555 multivibrator circuit, its performance is qualitatively analyzed. The frequency of the waveform output from the 555 is dependent on the rate of charging of the capacitor C1, seen in Appendix F. The current output from the optocoupler, the current that charges C1, varies with the LED current (forward current) of the optocoupler. The LED current is dependent on the supplied voltage from the DAC, and its magnitude in relation to its forward voltage drop. The collector-emitter current (output current) of the optocoupler's phototransistor is dependent on the forward current of the LED, which is approximately constant above a certain voltage as seen in *Figure 17*. This would give a maximum to the rate C1 is charged at, and subsequently a maximum to the frequency of the output waveform.

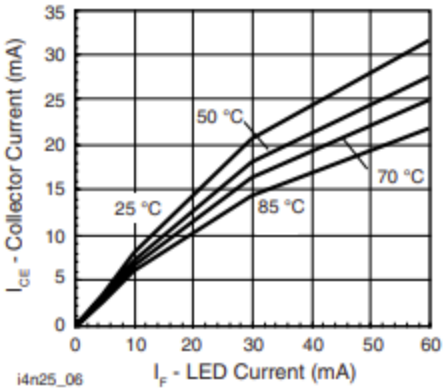


Fig. 6 - Collector Emitter Current vs. Temperature and LED Current

Figure 16: Optocoupler Output Current Dependence on LED Current[4]

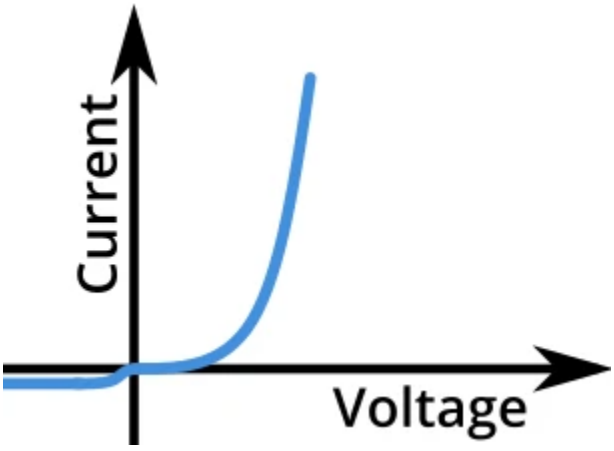


Figure 17: Diode IV Graph

Below 1V, no frequency change was observed. This was to be expected, as the forward voltage of the optocoupler LED is listed near this value, and no current will flow through the LED below this value[4].

The other shortcoming of the system was in its frequency measuring capability. The microcontroller was not able to measure frequencies over the entire calculated range. This is mainly due to time lost processing instructions as was discussed in **Testing: Frequency Measurements**.

Lessons Learned

The construction of this system was an awesome learning experience. Valuable experience was gained in embedded programming. Design of this system provided insight into the architecture of the STM32, an industry-standard microcontroller. This experience will surely prove useful in further embedded programming projects, whether that be on the STM32 or another microcontroller.

Appendix A - Main.c

```c
//
// 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".
 *  ECE 355 lab project.
 *  By:
 *    Joel Meuleman
 *    Michael Pillon
 *
 *  See "system/include/cmsis/stm32f0xx.h" for register/bit definitions.
 *  See "system/src/cmsis/vectors_stm32f0xx.c" for handler declarations.
 */

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

#include "lcd.h"
#include "timer.h"
#include "adc.h"
#include "dac.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"

/* Interrupt mode constants */
#define EXTI_CONTROL_BIT_PUSH_BUTTON      EXTI_IMR_MR0
#define EXTI_CONTROL_BIT_555_TIMER        EXTI_IMR_MR1
#define EXTI_CONTROL_BIT_FUNCTION_GENERATOR   EXTI_IMR_MR2
#define PENDING_REGISTER_PUSH_BUTTON      EXTI_PR_PR0
#define PENDING_REGISTER_555_TIMER        EXTI_PR_PR1
#define PENDING_REGISTER_FUNCTION_GENERATOR   EXTI_PR_PR2

/* Hardware initialization */
void GPIOA_init(void);
void GPIOC_init(void);
void EXTI_init(void);

/* Function which formats data for use with lcd.h */
void writeToLCD(uint32_t frequency);
void calculateFrequencyFromTIM2(void);

/* Global variable indicating which LED is blinking */
#define MODE_555_TIMER (uint16_t)0x0100
#define MODE_FUNCTION_GENERATOR (uint16_t)0x0200
volatile uint16_t activeMode = MODE_FUNCTION_GENERATOR; /* Start program in mode defined here */
volatile uint8_t firstEdgeFlag = 0;

int main()
{
  trace_printf("Initializing ECE355 project\n");
  trace_printf("By: Joel Meuleman and Michael Pillon\n");
  GPIOA_init();   /* Initialize push button on GPIOA */
  TIM2_init();    /* Initialize timer TIM2 */
  EXTI_init();    /* Initialize external interrupts */
  GPIOC_init();   /* Initialize LEDs on GPIOC */
  adc_init();     /* Initialize analog to digital converter */
  dac_init();     /* Initialize digital to analog converter */
  lcd_init();     /* Initialize LCD */
  trace_printf("Successfully initialized project\n");

  while (1); /* Keep program running with an infinite loop */
  return EXIT_SUCCESS;
}

```

```c
/* Initialize a pushbutton on GPIOA */
void GPIOA_init()
{
  /* Enable clock for GPIOA peripheral */
  RCC->AHBENR |= RCC_AHBENR_GPIOAEN;
  /* Configure PA2 as input */
  GPIOA->MODER &= ~(GPIO_MODER_MODER2);
  /* Ensure no pull-up/pull-down for PA2 */
  GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR2);
}

/* Initialize 2 LEDs on GPIOC */
void GPIOC_init()
{
  /* Enable clock for GPIOC peripheral */
  RCC->AHBENR |= RCC_AHBENR_GPIOCEN;
  /* Configure PC8 and PC9 as outputs for LEDs */
  GPIOC->MODER |= (GPIO_MODER_MODER8_0 | GPIO_MODER_MODER9_0);
  /* Ensure push-pull mode selected for PC8 and PC9 */
  GPIOC->OTYPER &= ~(GPIO_OTYPER_OT_8 | GPIO_OTYPER_OT_9);
  /* Ensure high-speed mode for PC8 and PC9 */
  GPIOC->OSPEEDR |= (GPIO_OSPEEDER_OSPEEDR8 | GPIO_OSPEEDER_OSPEEDR9);
  /* Ensure no pull-up/pull-down for PC8 and PC9 */
  GPIOC->PUPDR &= ~(GPIO_PUPDR_PUPDR8 | GPIO_PUPDR_PUPDR9);
  /* Initialize LED of current mode */
  GPIOC->BSRR = activeMode;
}

/* Initialize external interrupts */
void EXTI_init()
{
  /* Map EXTI0 line to PA0 for push button */
  SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI0_PA;
  /* Map EXTI1 line to PA1 for the 555 timer */
  SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI1_PA;
  /* Map EXTI2 line to PA2 for the function generator */
  SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI2_PA;
  /* EXTI line interrupts: set rising-edge trigger */
  EXTI->RTSR |= EXTI_RTSR_TR0 | EXTI_RTSR_TR1 | EXTI_RTSR_TR2;
  /* Unmask interrupts from EXTI line 0 and 2 to enable push button and function generator interrupts */
  EXTI->IMR = EXTI_CONTROL_BIT_PUSH_BUTTON  | EXTI_CONTROL_BIT_FUNCTION_GENERATOR;
  /* Assign EXTI2 interrupt priority = 0 in NVIC */
  NVIC_SetPriority(EXTI0_1_IRQn, 0);
  NVIC_SetPriority(EXTI2_3_IRQn, 1);
  /* Enable EXTI2 interrupts in NVIC */
  NVIC_EnableIRQ(EXTI0_1_IRQn);
  NVIC_EnableIRQ(EXTI2_3_IRQn);
}

/* Push button and 555 timer interrupt handler */
/* This handler is declared in system/src/cmsis/vectors_stm32f0xx.c */
void EXTI0_1_IRQHandler(){
  /* Check if PB on PA0 triggered interrupt */
  if ((EXTI->PR & PENDING_REGISTER_PUSH_BUTTON) != 0){
    /* Wait for button to be released */
    while((GPIOA->IDR & GPIO_IDR_0) != 0);
    /* Make sure timer is stopped and first interrupt with be the first edge */
    TIM2->CR1 &= ~(TIM_CR1_CEN);
    firstEdgeFlag = 0;
    /* Turn off currently mode indicator LED */
    GPIOC->BRR = activeMode;
    /* Switch blinking LED */
    activeMode ^= (MODE_FUNCTION_GENERATOR | MODE_555_TIMER);
    /* Turn on switched LED */
    GPIOC->BSRR = activeMode;

    /* Enable interrupts for new mode */
    if(activeMode == MODE_555_TIMER) {
      trace_printf("Switched to 555 Timer\n");
      EXTI->IMR = EXTI_CONTROL_BIT_PUSH_BUTTON | EXTI_CONTROL_BIT_555_TIMER;
    } else {
      trace_printf("Switched to Function Generator\n");
      EXTI->IMR = EXTI_CONTROL_BIT_PUSH_BUTTON | EXTI_CONTROL_BIT_FUNCTION_GENERATOR;
    }
    /* Clear EXTI0 interrupt pending flag (EXTI->PR) */
    EXTI->PR = PENDING_REGISTER_PUSH_BUTTON;
  }

```

```
153     /* Check if 555 timer triggered interrupt */
154     if ((EXTI->PR & PENDING_REGISTER_555_TIMER) != 0){
155       /* Call timer handler to calculate frequency */
156       calculateFrequencyFromTIM2();
157       /* Clear EXTI1 interrupt pending flag (EXTI->PR) */
158       EXTI->PR = PENDING_REGISTER_555_TIMER;
159     }
160
161   }
162
163   /* Function generator interrupt handler */
164   /* This handler is declared in system/src/cmsis/vectors_stm32f0xx.c */
165   void EXTI2_3_IRQHandler()
166   {
167     /* Check if EXTI2 interrupt pending flag is indeed set */
168     if ((EXTI->PR & PENDING_REGISTER_FUNCTION_GENERATOR) != 0)
169     {
170       /* Call timer handler to calculate frequency */
171       calculateFrequencyFromTIM2();
172       /* Clear EXTI2 interrupt pending flag (EXTI->PR) */
173       EXTI->PR = PENDING_REGISTER_FUNCTION_GENERATOR;
174     }
175   }
176
177   void calculateFrequencyFromTIM2() {
178     if(firstEdgeFlag == 0){ /* If this is the first edge */
179       /* Clear count register */
180       TIM2->CNT = 0;
181       /* Set firstEdgeFlag */
182       firstEdgeFlag = 1;
183       /* Start timer */
184       TIM2->CR1 |= TIM_CR1_CEN;
185     } else { /* Else this is the second edge */
186       /* Stop timer */
187       TIM2->CR1 &= ~(TIM_CR1_CEN);
188       /* Read out count register */
189       uint32_t currentCount = TIM2->CNT;
190       /* Calculate signal frequency */
191       uint32_t signalFrequency = SystemCoreClock / currentCount;
192       writeToLCD(signalFrequency);
193       /* Reset firstEdgeFlag */
194       firstEdgeFlag = 0;
195     }
196   }
197
198   /* Write formatted output to LCD screen */
199   void writeToLCD(uint32_t frequency) {
200     /* Write Frequency to LCD */
201     char lcd_line1[LCD_LINE_LENGTH + 1];
202     sprintf(lcd_line1, "F: %li Hz", frequency);
203     lcd_display_line1(lcd_line1);
204
205     /* Get ADC value, convert to resistance and send the raw value to the DAC */
206     uint32_t rawAdcValue = adc_readRaw();
207     dac_write(rawAdcValue);
208
209     /* Write Resistance to LCD */
210     uint32_t adcResistance = (uint32_t)adc_convertRawToResistance(rawAdcValue);
211     char lcd_line2[LCD_LINE_LENGTH + 1];
212     sprintf(lcd_line2, "R: %li Ohm", adcResistance);
213     lcd_display_line2(lcd_line2);
214   }
215
216   #pragma GCC diagnostic pop
217
```

Appendix B - adc.h and adc.c

```c
/*
 * adc.h
 *
 */

#include "diag/Trace.h"
#include "cmsis/cmsis_device.h"
#include "dac.h"

#ifndef ADC_H_
#define ADC_H_

#define VDDA 3.3 /* Assume VDDA = 3.3V */
#define MAX_RESISTANCE_VALUE 5000 /* in Ohms */
#define ADC_RESOLUTION 4095 /* 2^12 - 1 */

/* Configure relevant ADC registers, start ADC */
void adc_init();

/* Reads the raw binary ADC value from 0 to ADC_RESOLUTION */
uint32_t adc_readRaw();
/* Helper to convert raw value to voltage */
double adc_convertRawToVoltage(uint32_t rawValue);
/* Reads the raw ADC value and converts it to a voltage */
double adc_readVoltage();
/* Helper to convert raw value to resistance */
double adc_convertRawToResistance(uint32_t rawValue);
/* Reads the raw ADC value and converts it to a resistance */
double adc_readResistance();

#endif /* ADC_H_ */
```

```c
/*
 * adc.c
 *
 */

#include "adc.h"

void adc_init(){
  RCC -> APB2ENR |= RCC_APB2ENR_ADCEN;  /* Enable peripheral clock on ADC */
  GPIOA->MODER |= GPIO_MODER_MODER5;    /* Analog mode on pin 5 */
  ADC1->CFGR1 = ADC_CFGR1_CONT | ADC_CFGR1_OVRMOD;  /* ADC config 12 bit res continuous */
  ADC1->CHSELR |= ADC_CHSELR_CHSEL5;  //ADC on pin5
  ADC1->SMPR &= ~(0x00000007);
  ADC1->SMPR |= (0x00000007); /* Configured for highest sampling time */
  ADC1->CR |= ADC_CR_ADEN;  /* Enable ADC */
  while(!(ADC1->ISR & ADC_CR_ADEN));  /* Wait for ADC */
  ADC1->CR |= 0x00000004; /* Start ADC conversion */
}

double adc_readVoltage(){
  uint32_t rawAdcValue = adc_readRaw();
  return adc_convertRawToVoltage(rawAdcValue);
}

double adc_readResistance(){
  uint32_t rawAdcValue = adc_readRaw();
  return adc_convertRawToResistance(rawAdcValue);
}

uint32_t adc_readRaw() {
  while(!(ADC1->ISR & 0x00000004)); /* Poll for EOC (End of Conversion) */
  return ADC1->DR;
}

double adc_convertRawToVoltage(uint32_t rawValue){
  return VDDA * rawValue / ADC_RESOLUTION;
}

double adc_convertRawToResistance(uint32_t rawValue){
  return MAX_RESISTANCE_VALUE * rawValue / ADC_RESOLUTION;
}
```

Appendix C - dac.h and dac.c

```c
/*
 * dac.h
 *
 */

#include "diag/Trace.h"
#include "cmsis/cmsis_device.h"

#ifndef DAC_H_
#define DAC_H_

/* Initialize DAC */
void dac_init();

/* Read DAC value */
uint32_t dac_read();
/* Write DAC register */
void dac_write(uint32_t value);

#endif /* DAC_H_ */
```

```c
/*
 * dac.c
 *
 */

#include "dac.h"

void dac_init(){
  /* Enable PA peripheral clock */
  RCC->AHBENR |= RCC_AHBENR_GPIOAEN;
  /* Enable DAC clock */
  RCC->APB1ENR |= RCC_APB1ENR_DACEN;
  /* GPIO for analog output of DAC */
  GPIOA->MODER |= GPIO_MODER_MODER4;
  /* Initialize and enable DAC channel */
  DAC->CR |= 0x00000001;
}

uint32_t dac_read(){
  return DAC->DOR1;
}

void dac_write(uint32_t value) {
  DAC->DHR12R1 = value;
}
```

Appendix D - timer.h and timer.c

```c
/*
 * timer.h
 *
 */

#include "diag/Trace.h"
#include "cmsis/cmsis_device.h"

#ifndef TIMER_H_
#define TIMER_H_

/* Clock prescaler for TIM2 timer: no prescaling */
#define TIM2_PRESCALER ((uint16_t)0x0000)
/* Default auto reload for TIM2 of maximum value */
#define TIM2_PERIOD ((uint32_t)0xFFFFFFFF)

/* Initialize TIM2 */
void TIM2_init();

#endif /* TIMER_H_ */
```

```c
/*
 * timer.c
 *
 */

#include "timer.h"

void TIM2_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 */
  TIM2->CR1 = ((uint16_t)0x008C);
  /* Set clock prescaler value */
  TIM2->PSC = TIM2_PRESCALER;
  /* Set auto-reloaded delay */
  TIM2->ARR = TIM2_PERIOD;
  /* Update timer registers */
  TIM2->EGR = ((uint16_t)0x0001);
  /* Assign TIM2 interrupt priority = 0 in NVIC */
  NVIC_SetPriority(TIM2_IRQn, 0);
  /* Enable TIM2 interrupts in NVIC */
  NVIC_EnableIRQ(TIM2_IRQn);
  /* Enable update interrupt generation */
  TIM2->DIER |= TIM_DIER_UIE;
}

/* Handles overflow of TIM2 */
/* 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 */
    TIM2->SR &= ~(TIM_SR_UIF);

    /* Restart stopped timer */
    TIM2->CR1 |= TIM_CR1_CEN;
  }
}
```

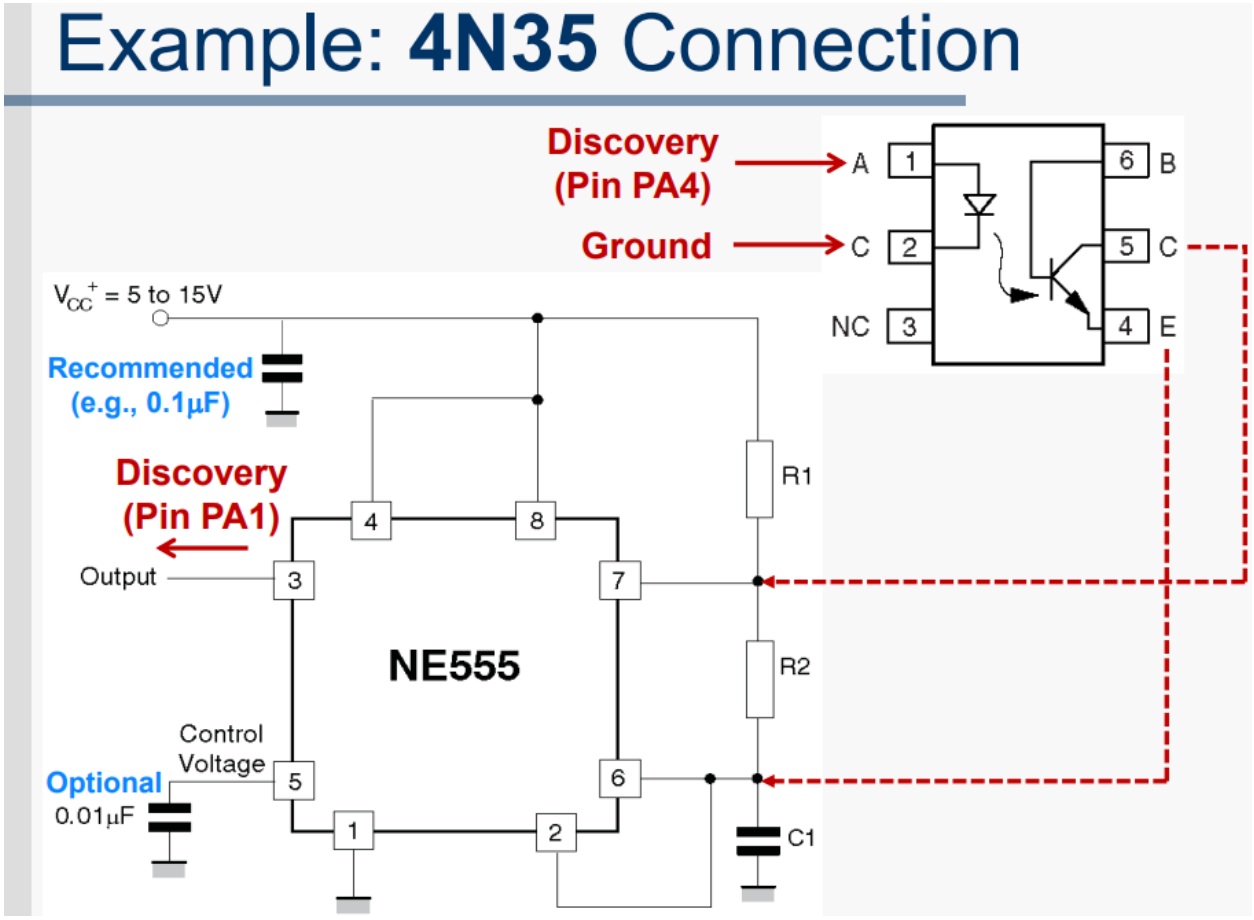
Appendix E - lcd.h and lcd.c

```c
/*
 * lcd.h
 *
 */
#include "stdint.h"
#include "string.h"

#include "diag/Trace.h"
#include "cmsis/cmsis_device.h"

#ifndef LCD_H_
#define LCD_H_
/* LCD Pin defines */
#define HANDSHAKE   GPIO_IDR_7
#define DB0     0x0100
#define DB1     0x0200
#define DB2     0x0400
#define DB3     0x0800
#define DB4     0x1000
#define DB5     0x2000
#define DB6     0x4000
#define DB7     0x8000
#define RW      0x0040
#define RS      0x0020
#define EN      0x0010

/* LCD constants */
#define LINE_1    DB7
#define LINE_2    DB7 | DB6
#define LCD_LINE_LENGTH 16

void lcd_gpio_init();

/* Initialize LCD hardware */
void lcd_init();

void lcd_display_line1(char *data);
void lcd_display_line2(char *data);
void lcd_clearDisplay();

/*
 * Write a single character or instruction
 * Must be shifted to data pins ( << 8 )
 * Must include bits for register select and read/write
 */
void lcd_write(uint16_t data);

#endif /* LCD_H_ */
```

```c
/*
 * lcd.c
 *
 */

#include "lcd.h"

void lcd_gpio_init(){
  /* Enable clock for GPIOB */
  RCC->AHBENR |= RCC_AHBENR_GPIOBEN;
  /* Set all needed pins as outputs */
  GPIOB->MODER |= GPIO_MODER_MODER4_0
      | GPIO_MODER_MODER5_0
      | GPIO_MODER_MODER6_0
      | GPIO_MODER_MODER8_0
      | GPIO_MODER_MODER9_0
      | GPIO_MODER_MODER10_0
      | GPIO_MODER_MODER11_0
      | GPIO_MODER_MODER12_0
      | GPIO_MODER_MODER13_0
      | GPIO_MODER_MODER14_0
      | GPIO_MODER_MODER15_0;
  /* Set high speed mode */
  GPIOB->OSPEEDR |= 0xFFFFFFFF;
}

```

```c
void lcd_init()
{
  lcd_gpio_init(); /* Initialize pins for LCD */
  lcd_write(DB5 | DB4 | DB3); /* 8-bit mode, 2 line, 5x7 charcters */
  lcd_write(DB3 | DB2); /* Display on, cursor off, blink off */
  lcd_write(DB2 | DB1); /* Auto-increment enabled, increment to the right */
  lcd_clearDisplay();
}

void lcd_write(uint16_t data)
{
  GPIOB->BSRR |= data; /* Write data to pins */
  GPIOB->BSRR |= EN;
  while(!(GPIOB->IDR & HANDSHAKE));
  GPIOB->BSRR |= EN << 16;
  while((GPIOB->IDR & HANDSHAKE));
  GPIOB->BSRR |= data << 16;
}

void lcd_clearDisplay()
{
  lcd_write(DB0);
}

void lcd_display_line1(char *data)
{
  /* Select line 1 */
  lcd_write(LINE_1);
  unsigned int i = 0;
  /* Write data */
  for(; i < strlen(data); i++){
    lcd_write(RS | data[i]<<8);
  }
  /* Ensure the rest of the line is empty */
  for(; i < LCD_LINE_LENGTH; i++){
    lcd_write(RS | ' ' << 8);
  }
}

void lcd_display_line2(char *data)
{
  /* Select line 2 */
  lcd_write(LINE_2);
  unsigned int i = 0;
  /* Write data */
  for(; i < strlen(data); i++){
    lcd_write(RS | data[i]<<8);
  }
  /* Ensure the rest of the line is empty */
  for(; i < LCD_LINE_LENGTH; i++){
    lcd_write(RS | ' ' << 8);
  }
}
```

Appendix F - Multivibrator Circuit Schematic [4]



References

[1] B. Sirna and D. Rakhamatov, “ECE 355: Microprocessor-Based Systems Laboratory Manual”, 2021, https://ece.engr.uvic.ca/~ece355/lab/ECE355-LabManual-2021.pdf (accessed Nov 29, 2021)

[2] STMicroelectronics, “RM0091 Reference Manual”, 018940 Rev 5 datasheet, Jan. 2014, https://www.ece.uvic.ca/~ece355/lab/supplement/stm32f0RefManual.pdf (accessed Nov. 29, 21)

[3] STMicroelectronics, “STM32F051xx”, 022265 Rev 4 datasheet, Jan. 2014, https://www.ece.uvic.ca/~ece355/lab/supplement/stm32f051r8t6_datatsheet_DM00039193.pdf (accessed Nov. 29, 21)

[4] D.N. Rakhmatov. (2021). Interfacing [Online]. Available: https://www.ece.uvic.ca/~daler/courses/ece355/interface.pdf

[5] Vishay Semiconductors, “Optocoupler, Phototransistor Output, with Base Connection”, 81181 Rev 1.2 datasheet, Jan. 2010, https://www.ece.uvic.ca/~ece355/lab/supplement/4n35.pdf (accessed Nov. 29, 21)

[6] Hitachi, “HD44780U (LCD-II)”, ADE-207-272(Z) Rev 0 datasheet, 1998, https://www.ece.uvic.ca/~ece355/lab/supplement/HD44780.pdf (accessed Nov. 29, 21)