University of Victoria

Department of Electrical & Computer Engineering

Fall 2020

ECE 355 Project Report:
PWM Signal Generation and Monitoring System

Nolan Caissie

[redacted]

December 4, 2020

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**) _____

1. Specifications

The purpose of this project is to implement an embedded system using an STM32F051 microcontroller. The microcontroller must control a (pulse-width modulation) PWM signal generated by a 555 timer; the STM32F051 microcontroller will drive an optocoupler which will, in turn, control the frequency generated by the 555 timer [1]. A potentiometer will be used to enable human analog input (the microcontroller will measure the voltage across this potentiometer) for controlling the voltage going to the optocoupler (this will in turn control the frequency of the 555 timer) [1]. The microcontroller will also measure the frequency of the 555 timer. The measurements of frequency and resistance taken by the microcontroller will be displayed on an LCD screen; it is important to note that the potentiometer and LCD are controlled through an "ECE 355 Emulation board", which provides a graphical user interface (GUI) to view the LCD screen and its signals, view the value of the voltage used to drive the optocoupler, and control the potentiometers resistance [1].

General purpose Input/Output (GPIO) pins provide the interface in which the microcontroller will communicate with external peripherals [2] (such as the optocoupler, 555 timer, and potentiometer). The voltage used to drive the optocoupler is controlled by the STMF0 Discovery board which implements a series of internal peripherals [2, 3, 4] made configurable through C programming. The internal peripherals that are of importance to this project are: a digital-to-analog converter (DAC); a analog-to-digital converter (ADC); Port A, B, and C GPIO ports; the EXTI external interrupts and events controller; and the TIM2 general purpose timer. A diagram of the interfacing connections is shown in figure 1.

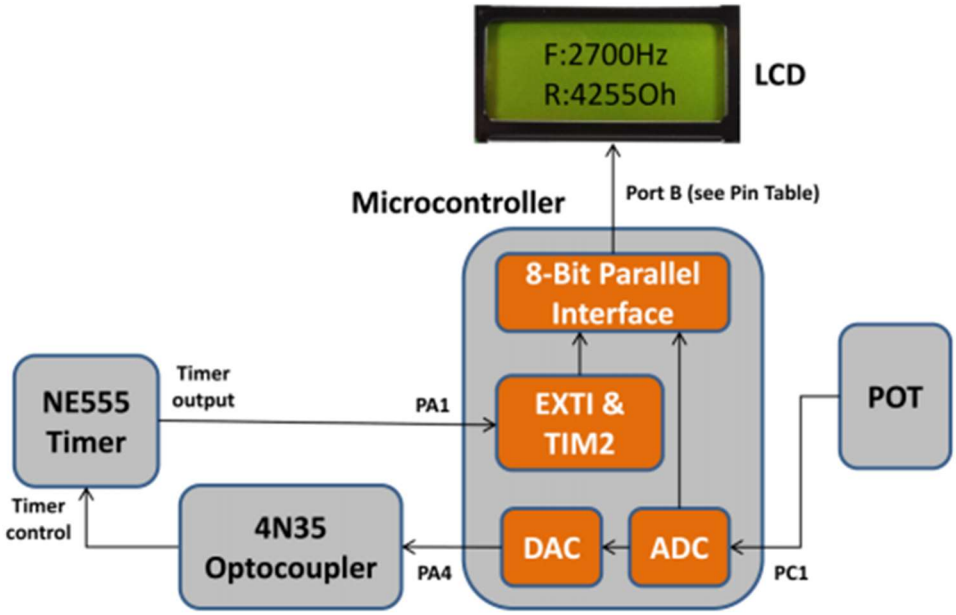

Figure 1: Project Design Specifications [1]

2. Design & Solution

The project was divided into a series of modular steps (performed in the order listed): the configuration/implementation of the LCD, the ADC, the DAC, the TIM2 general-purpose timer and the EXTI controller. All of the configurations are done in software using the C programming language. A header file, provided by the MCD Application Team [5] contains all the bit definitions, register declarations, and peripheral addresses needed to implement the project. "The DAC will be used to drive the optocoupler to adjust the signal frequency of the 555 timer, based on the potentiometer voltage read by the ADC" [1]. The TIM2 general-purpose timer will measure the frequency of the PWM signal coming from the 555 timer; it will do this by making using the EXTI's rising-edge trigger (when a rising edge is detected at the GPIO input, the TIM2 timer will begin counting) [1]. GPIO port B will be configured for use with the LCD; GPIO port C, pin 1 (PC1) will be

configured to measure resistance from the potentiometer; and GPIO port A, pin 1 (PA1) will measure the 555 timer's frequency while pin 4 of GPIO port A (PA4) will be used to drive the optocoupler.

2.1 The GPIO Ports

In order to initialize the microcontroller GPIO ports, the enable register for the advanced high-performance bus's (AHB) clock must be set to 1, providing a clock signal to the specified GPIO ports; this register is contained within the reset and clock control (RCC) module on the microcontroller board [2, 4]. All GPIO ports require a clock signal to operate with the microcontroller; these clock signals allow instructions to be sent/received by the microcontroller. The other registers that are of importance are the GPIO port mode registers, pull-up/pull-down registers, output data registers, and input data registers. The mode registers are used to configure the direction of the GPIO port; 00 is for input mode, 01 is for general purpose output mode, and 11 is for analog mode [2]. For the purpose of this project, all GPIO ports will be configured as 00 for no pull-up/pull-down [2]. Input data registers are used to read data from GPIO pins sent by peripherals such as the LCD, while output data registers are used to write data to the GPIO pins to be retrieved by peripherals such as the LCD. Figure 2 shows the structure of the GPIO ports; data sent to output data registers goes to the I/O pins, while data taken in by the I/O pins can be read from the input data registers.

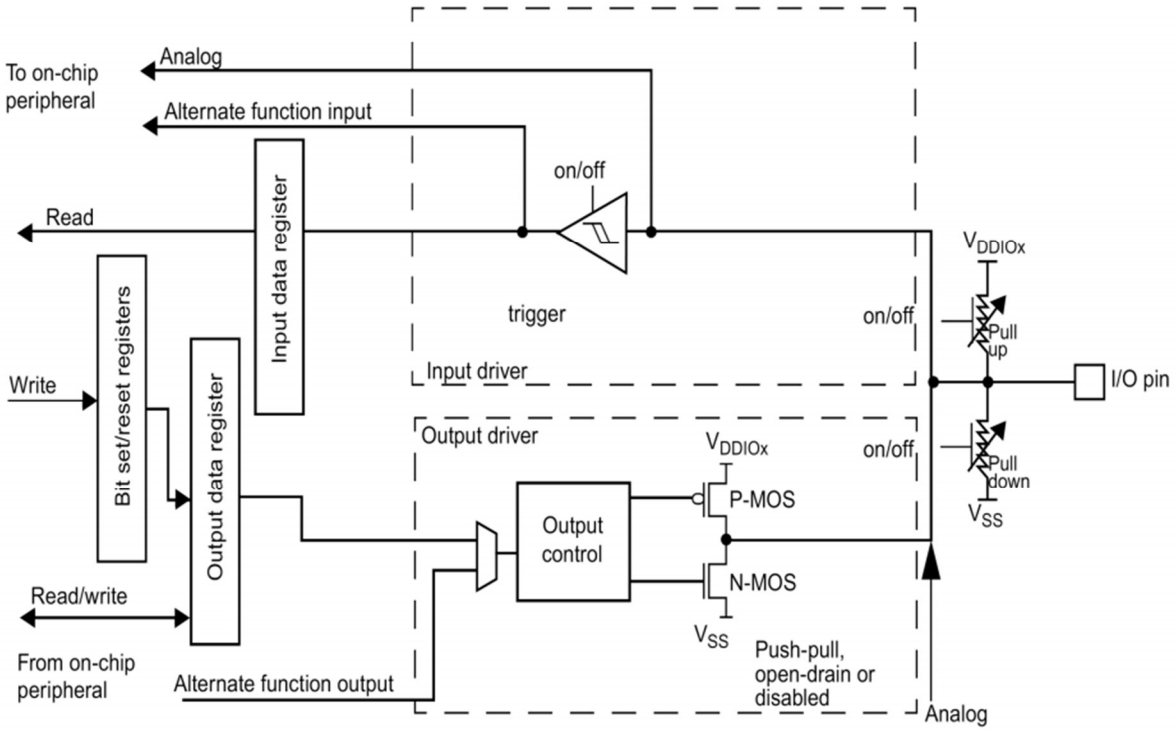

Figure 2: I/O Port Structure [2]

2.2 The LCD

The LCD is an external device that must receive sets of instructions to configure it for use. The diagram shown in figure 3 contains the relevant sets of instructions required to configure the LCD (of which four instructions will be used). The instructions relevant to the configuration of the LCD for this project's purposes (listed in order) are as follows:

1. Function set: "Sets to 8-bit operation and selects 2-line display and 5x8 dot character font" [6]
2. Display on/off control: "Turns on display [no cursor]" [6]
3. Entry mode set: "Sets mode to increment the address by one and shift the cursor to the right" [6]
4. Clear display: "Clears entire display and sets DDRAM address 0 in address counter" [6]

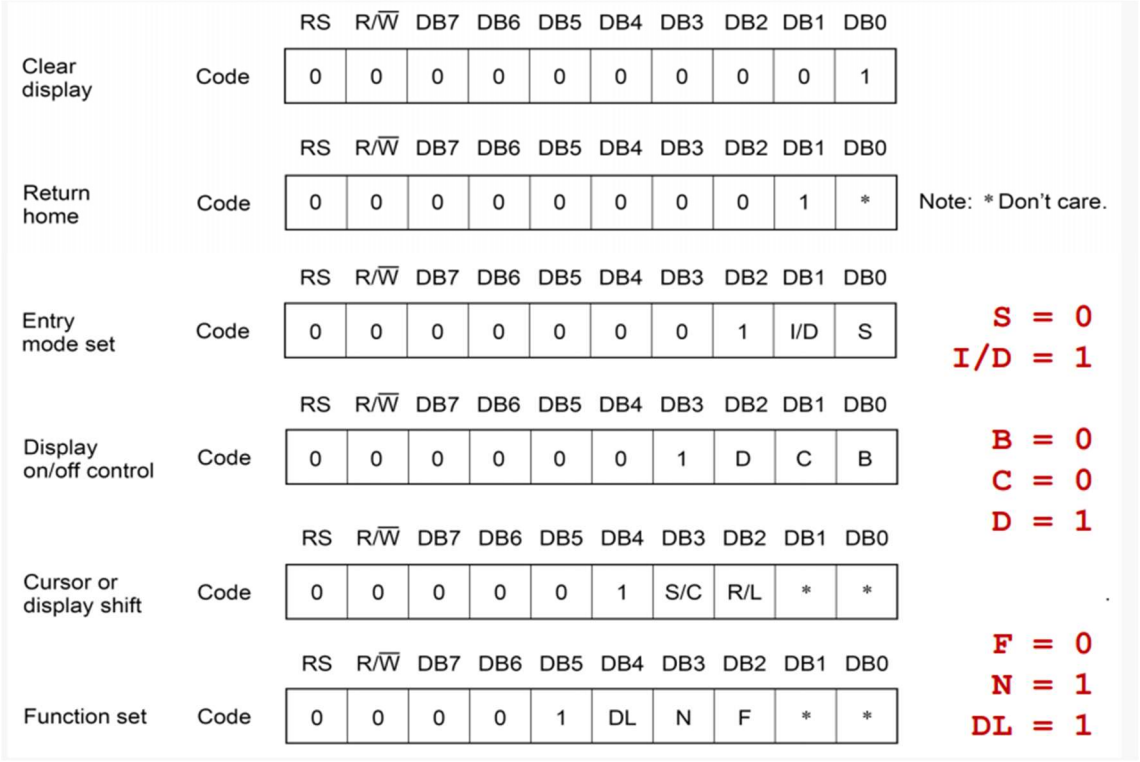

Figure 3: LCD Configuration Instructions [3]

Once the latter four configuration instructions have been successfully received by the LCD, the display will be ready for data in the form of ASCII characters. The entry mode instruction allows the display to increment the address in the LCD's DDRAM, while the function set instruction allows the display to utilize 8-bit operation (for 8-bit ASCII characters) and the ability to address two separate lines on the display (one to display frequency, one to display resistance as shown in figure 1). Figure 4 depicts the access instructions to implement this process; first, the DDRAM address for the first line must be set (data bit 7 must be set to 1, while the "A's" represent address bits), then 8-bit ASCII characters will be written to the DDRAM for display (RS must be set to 1, while the "D's" represent each bit of the 8-bit ASCII characters). For this project, 8 ASCII characters will be displayed on each line in the form, "F:xxxxHz" and "R:xxxxOh", on the first and second lines, respectively. It is important to emphasize that, first, the DDRAM address for the line must be set, then 8 separate instructions each containing 1 of the 8 ASCII characters must be sent to the LCD'S DDRAM for a total of 9 instructions per line of the LCD display.

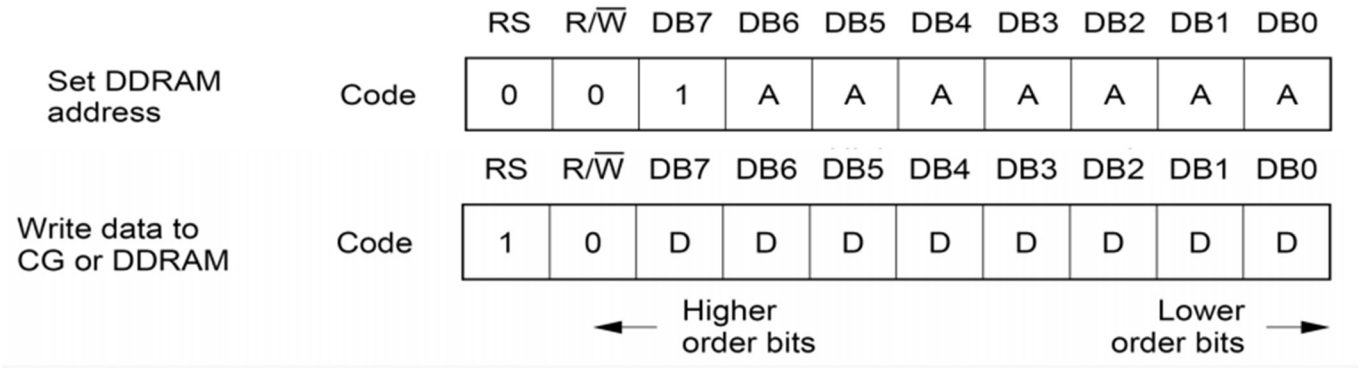

Figure 4: DDRAM Access Instructions [3]

To implement the aforementioned LCD instructions, the GPIO ports and pins hardwired to the LCD's pins must be configured as described in section 2.1. Figure 5 shows the hardwired connections between the STM32F0 and the LCD, and the configuration "direction" (i.e., whether the GPIO port will act as an output or an input).

STM32F0	SIGNAL	DIRECTION
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 5: The LCD's Interface [3]

Using the bit definitions for specified registers in the header file [5], provided by STMicroelectronics, the code for the GPIO configurations was written as shown in figure 6. As previously mentioned, the AHB clock enable register "AHBENR" must be set to 1 and this is implemented in the first line of the "myGPIOB_Init" function. The other lines in this function simply configure the mode registers as input or output as described in figure 5 and ensure that there is no pull-up/pull-down for any of the pins.

```
//Here I need to initialize GPIOB for use with the LCD
void myGPIOB_Init()
{
   /*Enable clock for GPIOB peripheral */
   //Relevant register: RCC->AHBENR
   RCC->AHBENR |= RCC_AHBENR_GPIOBEN;
   //Configure PB7 as an input
   GPIOB->MODER &= ~(GPIO_MODER_MODER7);
   /* Ensure no pull-up/pull-down for PB7 */
   GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR7);
   //Configure PB4-6 and PB8-15 as output
   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);
   /* Ensure no pull-up/pull-down for PB4-6 and PB8-15 - I think this is correct for this */
   GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR4 | GPIO_PUPDR_PUPDR5 | GPIO_PUPDR_PUPDR6 | GPIO_PUPDR_PUPDR8 |
            GPIO_PUPDR_PUPDR9 | GPIO_PUPDR_PUPDR10 | GPIO_PUPDR_PUPDR11 | GPIO_PUPDR_PUPDR12 | GPIO_PUPDR_PUPDR13 |
            GPIO_PUPDR_PUPDR14 | GPIO_PUPDR_PUPDR15);
   /* Ensure high-speed mode for PC8 and PC9 */
   //GPIOB->OSPEEDR |= (GPIO_OSPEEDER_OSPEEDR4 | GPIO_OSPEEDER_OSPEEDR5 | GPIO_OSPEEDER_OSPEEDR6 | GPIO_OSPEEDER_OSPEEDR8 |
   //   GPIO_OSPEEDER_OSPEEDR9 | GPIO_OSPEEDER_OSPEEDR10 | GPIO_OSPEEDER_OSPEEDR11 | GPIO_OSPEEDER_OSPEEDR12 |
   //          GPIO_OSPEEDER_OSPEEDR13 | GPIO_OSPEEDER_OSPEEDR14 | GPIO_OSPEEDER_OSPEEDR15);
   /* Ensure push-pull mode selected for data pins*/
   //GPIOB->OTYPER &= ~(GPIO_OTYPER_OT_4 | GPIO_OTYPER_OT_5 | GPIO_OTYPER_OT_6 | GPIO_OTYPER_OT_8 | GPIO_OTYPER_OT_9 |
   //GPIO_OTYPER_OT_10 | GPIO_OTYPER_OT_11 | GPIO_OTYPER_OT_12 | GPIO_OTYPER_OT_13 | GPIO_OTYPER_OT_14 | GPIO_OTYPER_OT_15);
}
```

Figure 6: The GPIOB Initialization Code

A "handshaking" process must be used to ensure that write operations to the LCD are performed at the correct time; the LCD and the GPIO pins of the microcontroller communicate asynchronously, each having their own timeline and process for communicating. The handshaking process consists of two important signals: Master-ready and Slave-ready [7]. For this implementation, the "master" is the microcontroller, and the "slave" is the LCD. The idea, is that the microcontroller "places the address and command on the bus and asserts Master-ready" [7]; the LCD then "decodes the address [and] performs the requested operation and asserts Slave-ready" [7]. The microcontroller then waits for the for the LCD to retrieve the instructions and assert "Slave-ready" [7]; once the LCD has retrieved the instructions and confirmed the retrieval via the slave-ready signal, the microcontroller can remove the signals from the GPIO ports and de-assert the master-ready signal; the LCD can then remove its signals from the bus last. This process is repeated every time

instructions are sent to the LCD. Due to the need for handshaking to occur after every interaction with the LCD, a function called “myLCD_Handshake” was written; any time an interaction with the LCD happens this function will be called immediately afterwards. The code for the function is shown in figure 7. Instructions are sent to the output data registers “ODR” and read from the input data registers “IDR”. Again, as shown in figure 5, the LCD will assert slave-ready by setting PB7 to 1 (which is read through IDR) and the microcontroller can assert master-ready by setting PB4 to 1 (by writing to the ODR).

```
//Master-Slave Handshake
void myLCD_Handshake(){

    //Note that: #define GPIO_ODR_4        ((uint32_t)0x00000010)
    //and #define GPIO_ODR_7               ((uint32_t)0x00000080) and thus,
    //trace_printf will output 9
    //Assert master enable (PB[4]=1)
    GPIOB->ODR |= GPIO_ODR_4;
    //trace_printf("GPIOB I/O Data Registers Write Enable: %x \n", (uint16_t)GPIOB->IDR );
    //Wait for LCD (slave) to assert done (PB[7]=1)
    while((GPIOB->IDR & GPIO_IDR_7) == 0);
    //trace_printf("GPIOB I/O Data Registers LCD Slave assert done : %x \n", (uint16_t)GPIOB->IDR );
    //De-assert master enable (PB[4]=0)
    GPIOB->ODR &= ~(GPIO_ODR_4);
    //trace_printf("GPIOB I/O Data Registers De-assert Enable: %x \n", (uint16_t)GPIOB->IDR );
    //Wait for LCD (slave) to de-assert done (PB[7]=0)
    while((GPIOB->IDR & GPIO_IDR_7) !=0);
    //trace_printf("GPIOB I/O Data Registers LCD Slave De-assert Done : %x \n", (uint16_t)GPIOB->IDR );

}
```

Figure 7: The LCD Handshaking Code

To initialize the LCD, following the configuration instructions described above (and shown in figure 3), a function called “myLCD_Init” was written. This function makes use of the handshaking function/code in figure 7 and, using the bit definitions found in the header file [5], the instructions can be sent to the ODR registers to be retrieved by the LCD. Every instruction sent to the LCD is followed by calling the handshaking function (initiating the handshaking process ensuring that instructions will not get “jumbled” together). The code is shown in Figure 8.

```
void myLCD_Init()
{

  //Function set - PB13=DB5=1, PB12=DB4=1, PB11=DB3=1, PB10=DB2=0
  GPIOB->ODR = ((GPIO_ODR_13 | GPIO_ODR_12 | GPIO_ODR_11)&(~(GPIO_ODR_10)));
  //trace_printf("GPIO -> ODR LCD Function Set : %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  //Display on - PB11=DB3=1, PB10=D2=1, PB9=DB1=0, PB8=DB0=0
  GPIOB->ODR = ((GPIO_ODR_11 | GPIO_ODR_10 )&(~(GPIO_ODR_9 | GPIO_ODR_8 )));
  //trace_printf("GPIO -> ODR LCD Display On: %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  //Entry mode set - PB10=DB=1, PB9=DB1=1, PB8=DB0=0
  GPIOB->ODR = ((GPIO_ODR_10 | GPIO_ODR_9)&(~(GPIO_ODR_8)));
  //trace_printf("GPIO -> ODR LCD Entry Mode Set: %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  //Clear display - PB8=DB0=1
  GPIOB->ODR = GPIO_ODR_8;
  //trace_printf("GPIO -> ODR LCD Clear Display: %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();

}
```

Figure 8: The LCD Initialization Code

Some simple math was implemented to extract the digits in the thousands, hundreds, tens, and ones places of the resistance and frequency values; once each of these digits is extracted, code can be implemented to convert these values into ASCII characters. The conversion process is quite simple and consists of simply adding 48 to the value (when expressed as an unsigned integer). The idea for this process was obtained from slide 2 of Professor Rakhmatov’s “Interface Examples” [3] in conjunction with Professor Rakhmatov’s lectures. The process of “digit extraction” is shown in the first 10 lines of code of the “myLCD_Update” function shown in figure 9. The conversion to ASCII process is implemented in 8 lines of code: 4 lines of code for line 1 of the display, and 4 lines of code for line 2 of the display (the extracted digits are incremented by 48 and then shifted left by 8 bits using the “<<” bit shifting operator as seen in figure 9). It is important to note that the ASCII characters were shifted by 8 bits due to the LCD’s data registers (D0-D7 as

shown in figure 5) correspond to GPIO output ports PB8-PB15, and thus, ODR bit definitions 8-15. ODR bits 5 and 6 correspond to “RS” and “R/W” (figure 5) and are set following the requirements shown in figure 4. The address for the first line of the LCD is “0x00”, and the address for the second line of the LCD is “0x40”; thus, the GPIO pin connected to D6 of the LCD (PB14) is represented by ODR bit 14 and is 0 for line 1 and 1 for line 2.

```
//Refresh the LCD with new values
void myLCD_Update()
{
    //Divide the ADC data into ones, tens, hundreds, thousands for display
    uint16_t resistance = ADC_Data*ADC_CONV_FACTOR;
    uint16_t resistance_thousands = (resistance)/1000;
    uint16_t resistance_hundreds = ((resistance)-(resistance_thousands*1000))/100;
    uint16_t resistance_tens = ((resistance)-(resistance_thousands*1000)-(resistance_hundreds*100))/10;
    uint16_t resistance_ones = ((resistance)-(resistance_thousands*1000)-(resistance_hundreds*100)-(resistance_tens*10))/1;
    //Divid the DAC data into ones, tens, hundreds, thousands for display
    unsigned int frequency = TIM2_Frequency;
    unsigned int frequency_thousands = (frequency)/1000;
    unsigned int frequency_hundreds = ((frequency)-(frequency_thousands*1000))/100;
    unsigned int frequency_tens = ((frequency)-(frequency_thousands*1000)-(frequency_hundreds*100))/10;
    unsigned int frequency_ones = ((frequency)-(frequency_thousands*1000)-(frequency_hundreds*100)-(frequency_tens*10))/1;


   //Set the LCD's DDRAM 7 bit address you will write to (here RS=0 and R/W=0, DB7=1 and DB6-0=A[6:0] - A for address)

   //Write 8-bits of data D[7:0] into LCD's DDRAM (here RS=1 and R/W=0 and DB7-0=D[7:0](ASCII code)) - each ASCII is written seperately
   //Note: cannot read from DDRAM (R/W is always 0)
   //Note: DDRAM address is incremented for you every time - only have to set the address twice (once for each line)

    //Set DDRAM address for line 1 PB5=RS=0, PB6=R/W=0, PB15=DB7=1, PB8,PB9,...,PB14=D0,D1,...,D6=0
  GPIOB->ODR = (~(GPIO_ODR_5 | GPIO_ODR_6 )&(GPIO_ODR_15));
  myLCD_Handshake();
  //trace_printf("GPIO -> ODR Line 1 Address: %x \n", (uint16_t)GPIOB->ODR);

  //Write data to DDRAM
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|(ASCII_F<<8));
  //trace_printf("GPIO -> ODR Write H : %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|(ASCII_COL<<8));
  //trace_printf("GPIO -> ODR Write F : %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|((frequency_thousands+48)<<8));
  //trace_printf("GPIO -> ODR Write 0: %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|((frequency_hundreds+48)<<8));
  //trace_printf("GPIO -> ODR Write 1: %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|((frequency_tens+48)<<8));
  //trace_printf("GPIO -> ODR Write 2: %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|((frequency_ones+48)<<8));
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|(ASCII_H<<8));
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|(ASCII_z<<8));
  myLCD_Handshake();

  //Set DDRAM address for line 2 PB5=RS=0, PB6=R/W=0, PB15=DB7=1, PB14=DB6=1 PB8,PB9,...,PB14=D0,D1,...,D6=0
  GPIOB->ODR = (~(GPIO_ODR_5 | GPIO_ODR_6 )&(GPIO_ODR_15|GPIO_ODR_14));
  myLCD_Handshake();
  //trace_printf("GPIO -> ODR Line 2 Address: %x \n", (uint16_t)GPIOB->ODR);
  //Write data to DDRAM
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|(ASCII_R<<8));
  //trace_printf("GPIO -> ODR Write H : %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|(ASCII_COL<<8));
  //trace_printf("GPIO -> ODR Write F : %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|((resistance_thousands+48)<<8));
  //trace_printf("GPIO -> ODR Write 0: %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|((resistance_hundreds+48)<<8));
  //trace_printf("GPIO -> ODR Write 1: %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|((resistance_tens+48)<<8));
  //trace_printf("GPIO -> ODR Write 2: %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|((resistance_ones+48)<<8));
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|(ASCII_O<<8));
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|(ASCII_h<<8));
  myLCD_Handshake();
}
```

Figure 9: The LCD Update Code

2.3 The ADC

The ADC has 16 channels which are internally connected to GPIO pins on the microcontroller board [3]. Pin 1 of port C is connected internally to the ADC input channel 11 [3]; thus, when configured properly the, the ADC can read the voltage drop across the potentiometer (and in turn, the resistance) being input to GPIO Port C, Pin 1. The ADC will convert this analog signal into a digital signal to be passed to the DAC (discussed in subsequent section 2.4).

Analogous to the GPIOB initialization shown in figure 6 for the LCD (see section 2.1 for GPIO initialization), pin 1 of port C (PC1) must also be configured for use. The clock must be enabled for the GPIOC peripheral, PC1 must have its direction configured by accessing the mode register ("MODER" in figure 10), and the pull-up/pull-down register ("PUPDR" in figure 10) must be set to ensure no pull-up/pull-down. The main difference in the configuration of GPIOC versus GPIOB is that PC1's mode must be set to "11" (represented by the bit definition"GPIO_MODER_MODER1" in figure 10) to act as an analog input; this will allow PC1 to read the analog voltage drop over the potentiometer resistance. Figure 10 shows the function "myGPIOC_Init" which is implemented to accomplish the aforementioned tasks in three lines of code.

```
//Here I need to initialize GPIOC for use with the ADC and POT
void myGPIOC_Init()
{
   /*Enable clock for GPIOC peripheral */
   //Relevant register: RCC->AHBENR
   RCC->AHBENR |= RCC_AHBENR_GPIOCEN;
   //Configure PC1 as analog
   GPIOC->MODER |= GPIO_MODER_MODER1;
   /*Ensure no pull-up/pull-down for PC1 - this should be correct for this I think */
   GPIOC->PUPDR &= ~(GPIO_PUPDR_PUPDR1);
}
```

Figure 10: The GPIOC Initialization Code

In the same manner that GPIO's need a clock signal enabled by the AHB, the ADC's clock must be enabled by setting bit 9 of the advanced peripheral bus register "ABP2ENR" [3] (shown in figure 11), which is also located in the RCC module. Enabling this bit enables communication between the microcontroller processor and this internal peripheral. This is the first step in initializing the ADC.

RCC->APB2ENR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	DBGM CUEN	Res.	Res.	Res.	TIM17 EN	TIM16 EN	TIM15 EN
									rw				rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	USART1 EN	Res.	SPI1 EN	TIM1 EN	Res.	ADC EN	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	SYSCFG COMP EN
	rw		rw	rw		rw									rw

Figure 11: The ADC Clock Enable Bit [3]

The ADC itself, has several internal registers that need to be configured for this project: the control register, the sampling time register, the channel selection register, and the configuration register; other internal registers such as the ADC's interrupt service routine (ISR) register and the ADC's 12-bit data register (to pass data to the LCD, and the DAC described in the next section) must be read from [3]. The important bits in the latter mentioned registers required for initialization are as follows:

1) The control register:
   - "ADEN": setting ADEN=1 enables the ADC (essentially supplying it with power) [3, 8]
   - "ADSTART": setting ADSTART=1 starts ADC conversions [3, 8]
2) The ISR register:
   - "ADRDY": this bit is automatically set to 1 by hardware once the ADC is ready for operation [3, 8]
   - "EOC": this bit is automatically set to 1 by hardware once a conversion is complete [3, 8]
3) The sampling time register:

- “SMP[2:0]”: these bits determine the sampling time through ADC clock cycles [3, 8]

4) The configuration register:
   - “CONT”: setting CONT=1 puts the ADC into continuous conversion mode, allowing the ADC to keep convert when triggered by hardware
   - “OVRMOD”: setting OVRMOD=1 allows the data register in the ADC to be overwritten so that it always contains the latest conversion [3, 8]

3) 5) The channel selection register: The control register:
   - “ADEN”: setting ADEN=1 enables the ADC (essentially supplying it with power) [3, 8]
   - “ADSTART”: setting ADSTART=1 starts ADC conversions [3, 8]

4) The ISR register:
   - “ADRDY”: this bit is automatically set to 1 by hardware once the ADC is ready for operation [3, 8]
   - “EOC”: this bit is automatically set to 1 by hardware once a conversion is complete [3, 8]

3) The sampling time register:
   - “SMP[2:0]”: these bits determine the sampling time through ADC clock cycles [3, 8]

4) The configuration register:
   - “CONT”: setting CONT=1 puts the ADC into continuous conversion mode, allowing the ADC to keep convert when triggered by hardware
   - “OVRMOD”: setting OVRMOD=1 allows the data register in the ADC to be overwritten so that it always contains the latest conversion [3, 8]

5) The channel selection register:
   - “CHSEL11”: setting CHSEL11=1 enables channel 11 to be used for conversion since it is internally connected to PC1 [3, 8]
   - “CHSEL11”: setting CHSEL11=1 enables channel 11 to be used for conversion since it is internally connected to PC1 [3, 8]

A function was written called “myADC_Init” to initialize the ADC and configure the correct bits. The function first enables the peripheral clock for the ADC; the function then configures the sampling time, channel, overrun management mode (OVRMOD), and continuous conversion (CONT); the function must then wait for ADRDY to become 1, indicating that the ADC is configured; once the ADC is fully configured, the conversion process can be started by making ADSTART=1.

```
void myADC_Init()
{
   /*Enable clock for the ADC*/
   RCC->APB2ENR |= RCC_APB2ENR_ADC1EN;
   //Make sure all the bits in the control register are zero (Bit [0]: ADEN can only be set if all bits of ADC_CR are 0)
   ADC1->CR &= 0x00000000;
   //Enable the ADC - Bit [0]: ADEN set to 1 (Note: this bit is cleared by hardware after ADDIS bit is set - ADDIS is a disable bit - i.e ADDIS=1=DISABLE)
   //Another note on ADDIS - once set to 1, the ADC will begin to shut down... once shut down it will clear ADEN and ADDIS will be cleared by hardware as a result
   ADC1->CR |= ADC_CR_ADEN;

   //Set the SMP[2:0] all to 1 - a sampling time to an intermediate value of 239 1/2 ADC clock cycles - Again, software can write these bits only when ADSTART=0
   ADC1->SMPR |= ADC_SMPR_SMP;
   //Set the channel selection register to channel 11 (since it is internally connected to PC1)
   ADC1->CHSELR |= ADC_CHSELR_CHSEL11;

   //Bit alignment is by default set to 0 which is right data alignment (this is what we want so do nothing)
   //Resolution is by default set to RES=00 which is exactly what we want so no need to change (do nothing)

   //Set the overrun management mode to 1: ADC_DR contents are preserved/overwritten when an overrun is detected and thus we will always read the freshest value
   ADC1->CFGR1 |= ADC_CFGR1_OVRMOD;

   //Set Bit[13]: CONT (continuous conversion mode - everytime it finishes converting it will continue running)
   ADC1->CFGR1 |= ADC_CFGR1_CONT;

   //Now, the ADC will continue converting due to the configurations made. The EOC flag will become 1 when a channel conversion is complete
   //thus EOC is the bit that needs to be polled by software - harware will clear this bit when ADC_DR is read and thus we will always read the freshest value
   //with no confusion overrun stuff (since overrun mode is set) ADC_ISR_EOC

   //Wait for the ADC ready flag (Bit[0]: ADRDY) - this will tell us that initialization is complete
   while((ADC1->ISR & ADC_ISR_ADRDY) == 0);
   //trace_printf("ADRDY is now 1 and the ADC IS READY!\n");

   //Once ADRDY is set 1, the ADC is ready to start converting
   /*should maybe include this stuff in the main?
   //Make sure ADDIS set to zero to allow ADSTART to be set and start the conversion process (most likely an unessecary step due to previous AND operation)
   ADC1->CR &= ~(ADC_CR_ADDIS);
   //Start the conversion process - this will stay 1 since we will just let it keep converting
   //Note: ADSTART is cleared by hardware when the conversion process is finished or stopped (but again.. we will let it keep converting)*/
   ADC1->CR |= ADC_CR_ADSTART;
}
```

Figure 12: The ADC Initialization Code

Inside the main function, an infinite loop is implemented to continuously poll data the data stored in the ADC from the PC1 potentiometer readings. The EOC bit will become 1 when a conversion is complete; once EOC=1, the ADC's data register can be read from, as this is where all conversions performed by the ADC are stored; the EOC bit will be cleared when the data register is read from. This infinite loop is shown in figure 13; an if statement checks the EOC bit, and then the data register is read into a global variable called "ADC_Data". This global variable is seen in the "myLCD_Update" function shown (figure 9), where it is used to calculate the resistance (with a scaling factor), extract the digits, and then convert to ASCII and display on the LCD. Figure 13 also shows this global variable being read into the DAC, which will in turn control PA4's analog voltage output. Again, all bit definitions needed for the ADC are in the header file [5].

```
while (1)
{
    //If the ADC conversion is complete, read the data register
    if((ADC1->ISR & ADC_ISR_EOC) != 0 ){
        //Read data register ADC->DR bits [11:0]
        ADC_Data = (ADC1->DR & ADC_DR_DATA);
        //trace_printf("The data reads: %u \n", (uint16_t)((ADC_Data)*ADC_CONV_FACTOR));
        //DAC register in use DAC->DHR12R1 (12 bit register for right-aligned) DAC_DHR12R1_DACC1DHR
        //Read from ADC1_DR and then send to DAC_DHR12R1 (i.e. DAC->DH12R1 = ADC1->DR)
        //trace_printf("The DAC Register: %x \n", (uint16_t)(DAC->DHR12R1));
        //ADC_Data = ADC_Data*1.9;

        //DAC->DHR12R1 = (ADC_Data & DAC_DHR12R1_DACC1DHR); -
        DAC->DHR12R1 = (ADC_Data & DAC_DHR12R1_DACC1DHR);
        DAC_Data = (DAC->DHR12R1);
        //DAC_Data =
        //trace_printf("The data passed from ADC1 to DHR12R1 is: %u \n", (uint16_t)(DAC_Data));
        myLCD_Update();

    }
}
```

Figure 13: The Infinite "while" Loop

2.4 Port A: The DAC, EXTI &TIM2

The initialization for GPIOA follows the same suit as the previous GPIOA initializations in sections 2.1-2.3. The clock is enabled for GPIOA, PA1 is configured as an input to read the frequency output from the 555 timer, PA4 is configured as an analog output to be used for driving the optocoupler, and both PA1 and PA4 are set to ensure no pull-up/pull-down. The initialization function "myGPIOA_Init" is shown in figure 14.

```
//this will be used to measure the frequency of the timer output
//will also need to initialize PA4 for the DAC
void myGPIOA_Init()
{
    /* Enable clock for GPIOA peripheral */
    // Relevant register: RCC->AHBENR
    RCC->AHBENR |= RCC_AHBENR_GPIOAEN;

    /* Configure PA1 as input */
    // Relevant register: GPIOA->MODER
    GPIOA->MODER &= ~(GPIO_MODER_MODER1);
    /* Ensure no pull-up/pull-down for PA1 */
    // Relevant register: GPIOA->PUPDR
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR1);

    /*Configure PA4 as analog */
    GPIOA->MODER |= GPIO_MODER_MODER4;
    /*Ensure no pull-up/pull-down for PA4*/
    //GPIOA->PUPDR |= GPIO_PUPDR_PUPDR4;
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR4);
    //Ensure push-pull
    //GPIOA->OTYPER &= ~(GPIO_OTYPER_OT_4);
    //Here, you need to initialize pin 4 for the DAC
    //GPIOA->OSPEEDR |= GPIO_OSPEEDER_OSPEEDR4;
}
```

Figure 14: The GPIOA Initialization Code

Initialization of the DAC is similar (yet simpler) to the initialization of the ADC. The DAC (like any peripheral) must have the clock enabled for it; in this case, the clock enable for the DAC is bit 29 of the APB1ENR register contained within the RCC module (shown in figure 15).

RCC->APB1ENR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	CEC EN	DAC EN	PWR EN	CRS EN	Res.	CAN EN	Res.	USB EN	I2C2 EN	I2C1 EN	Res.	USART 4EN	USART 3EN	USART 2EN	Res.
	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.	SPI2 EN	Res.	Res.	WWDG EN	Res.	Res.	TIM14 EN	Res.	Res.	TIM7 EN	TIM6 EN	Res.	Res.	TIM3 EN	TIM2 EN
	rw			rw			rw			rw	rw			rw	rw

Figure 15: The DAC Clock Enable Bit [3]

The DAC peripheral itself, is simpler in design when compared to the ADC. There are only two internal DAC registers of importance: the DAC control register, and the 12-bit DAC data register. Both registers are shown in the figure 16 block diagram; the control register is labelled “DAC control register”, and the data register is labelled “DHRx”. The DAC only has 1 channel unlike the 16 channel ADC [3, 8]; when this channel is enabled, it “becomes connected to pin PA4 automatically” [3]. The bits of interest for configuring the DAC are as follows:

1) The control register:
    - “EN1”: setting EN1=1 enables the DAC channel (supplying it with power) [3, 8]
    - “BOFF1”: setting BOFF1=0 enables the buffer and is “used to reduce the output impedance, and to drive external loads directly without having to add an external operational amplifier” [8] (ideal for driving the optocoupler)
    - “TEN1”: setting TEN1=0 disables the channel1 trigger [3, 8], which ensures that “data written into the DAC_DHRx register are transferred on APB clock cycle to the DAC_DOR1 register” [8] (the DAC_DOR1 register is where data is sent to be output by the DAC and is handled by the DAC hardware – see figure 16)

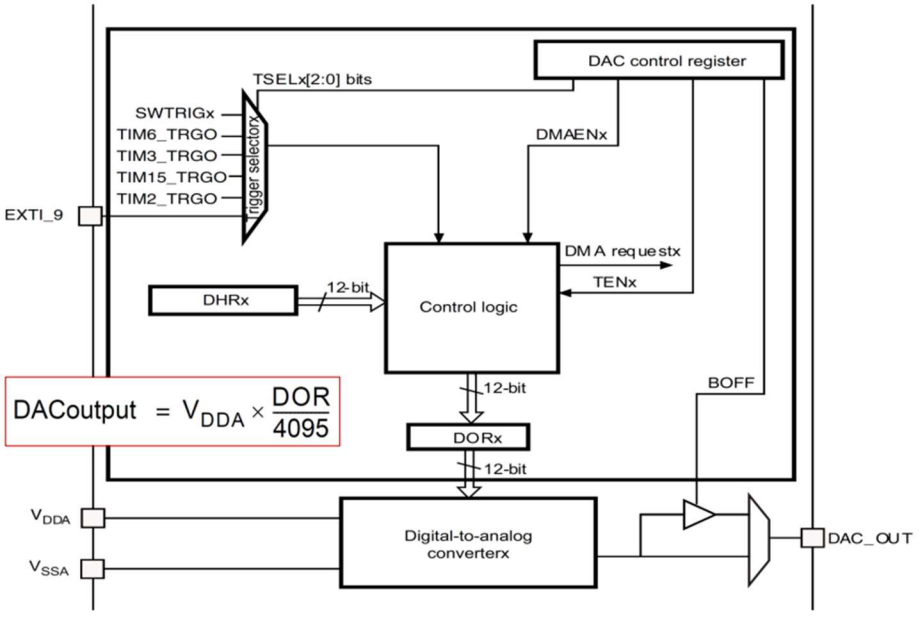

Figure 16: DAC Block Diagram [3]

The function initializing the DAC, “myDAC_Init”, is written before the EXTI and TIM2 functions. The function is shown in figure 17; the peripheral clock for the DAC is enabled, the DAC is enabled (powered on), BOFF1 and TEN1 are both set to 0 to reduce output impedance and disable the channel1 trigger respectively.

```
void myDAC_Init()
{
    /* Enable clock for the DAC*/
    RCC->APB1ENR |= RCC_APB1ENR_DACEN;
    //DACoutput=VDDA*DOR/4096
    //DAC register in use DAC->DHR12R1 (12 bit register for right-aligned) DAC_DHR12R1_DACC1DHR
    //Read from ADC1_DR and then send to DAC_DHR12R1 (i.e. DAC->DH12R1 = ADC1->DR)
    //DAC control register (DAC->CR) - Bit[0]:EN1 (Channel1 enable) - must configure PA4 as analog before you enable (RCC too)
    DAC->CR &= 0x00000000;
    DAC->CR |= DAC_CR_EN1;
    //Bit[1]:BBOFF1=0/1 (channel1 tri-state buffer is enabled/disabled) - need this set to 0 (should already be 0 by default)
    DAC->CR &= ~(DAC_CR_BOFF1);
    //Bit[2]:TEN1 (channel1 trigger enable)=0/1: - need this set to 0 (should already be 0 by default)
    DAC-> CR &= ~(DAC_CR_TEN1);
}
```

Figure 17: The DAC Initialization Code

Referring back to figure 13 (the infinite loop located in the main routine), it is seen that data is read from the "ADC_Data" global variable and placed in the DAC's 12-bit right-aligned [3, 5, 8] data register. Data sent into this register will be automatically converted from a digital signal to an analog voltage seen at PA4 (the voltage that drives the optocoupler). The data contained in the DAC's data register is then transferred to a global variable "DAC_Data" to be used for debugging purposes (to be discussed).

The next step is to configure the TIM2 general-purpose timer (see figure 18 for the "myTIM2_Init" function). Again, the peripheral clock (provided by the APB), is the first step to configuring the timer (the bit of importance here is TIM2EN). The control register is configured such that the buffer is set to auto-reload; TIM2 will count up, stop on overflow, interrupt on overflow only (for which the enable update events bit is configured) [2]. The relevant registers and bits to initialize TIM2 (both internal and external) are as follows:

1) The control register (TIM2):
    - "DIR": setting DIR =1 configures the counter for up-counting [3, 8]
    - "ARPE": setting ARPE=1 sets the TIM2_ARR such that it is buffered [3,8]
    - "UDIS": setting UDIS=0 ensures even requests are enabled [3,8]
    - "URS": setting URS=1 ensures update events are caused by counter overflow [3,8]
    - "OPM": setting OPM=1 ensures that the counter stops counting at the next update event (this will happen on counter overflow since we are up-counting and URS=1) [3, 8]
    - "CKD": setting CKD=00 sets clock division to an intermediate sampling frequency [3, 8]
    - "CEN": setting CEN=1 will enable the counter to start counting [3, 8]
2) The event generation register (TIM2):
    - "UG": setting UG=1 updates TIM2's registers (cleared by hardware) [3, 8]
3) The pre-scaler register (TIM2):
    - "PSC[15:0]": setting PSC[15:0]=0 ensures that that no pre-scaling happens [3, 8]
4) The auto-reload register (TIM2):
    - "ARR[32:0]": setting ARR[32:0]=1 sets the maximum possible setting for overflow [3, 8]
5) The DMA/interrupt enable register (TIM2):
    - "UIE": setting UIE=1 enables interrupts [3, 8]
6) The interrupt priority register (NVIC controller):
    - "IPR": setting IP=0 configures the interrupt priority as zero [3, 8]
    - "ISER": setting SETENA=1 enables the interrupt request line [3, 8]

```c
void myTIM2_Init()
{
    /* 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 = ((uint16_t)0x0001);

    /* Assign TIM2 interrupt priority = 0 in NVIC */
    // Relevant register: NVIC->IP[3], or use NVIC_SetPriority
    NVIC_SetPriority(TIM2_IRQn, 0);

    /* Enable TIM2 interrupts in NVIC */
    // Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
    NVIC_EnableIRQ(TIM2_IRQn);

    /* Enable update interrupt generation */
    // Relevant register: TIM2->DIER
    TIM2->DIER |= TIM_DIER_UIE;
    /* Start counting timer pulses */
    TIM2->CR1 |= TIM_CR1_CEN;

}
```

Figure 18: The TIM2 Initialization Code

Next to be configured is the EXTI controller (see figure 19 for the "myEXTI_Init" function). First, through the system configuration controller (SYSFG) EXTI1 must be mapped to PA1 using the EXTI1 configuration register (since EXTI line 1 is hardwired to PA1). The EXTI controller must then be set to trigger interrupts when a rising edge is seen on PA1 (enabling the period of the 555 timer to be calculated and thus the frequency)

1) The Rising trigger selection register (EXTI):
    - "TR": setting TR[1]=1 enables rising-edge trigger for EXTI line 1 [3, 8]
3) The interrupt mask register (EXTI):
    - "MR": setting MR[1]=1 ensures that that interrupt requests from line 1 are not masked [3, 8]
4) The DMA/interrupt enable register (NVIC controller):
    - "UIE": setting UIE=1 enables interrupts [3, 8]
5) The interrupt priority register (NVIC controller):
    - "IPR": setting IP=0 configures the interrupt priority as zero [3, 8]
    - "ISER": setting SETENA=1 enables the interrupt request line [3, 8]

```c
void myEXTI_Init()
{
    /* Map EXTI1 line to PA1 */
    // Relevant register: SYSCFG->EXTICR[0]
    //** I dont know if this is how we fill EXTI1 with zeros
    //SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI1_PA;
    SYSCFG->EXTICR[0]|= SYSCFG_EXTICR1_EXTI1_PA;

    /* EXTI1 line interrupts: set rising-edge trigger */
    // Relevant register: EXTI->RTSR
    EXTI->RTSR |= EXTI_RTSR_TR1;
    /* Unmask interrupts from EXTI1 line */
    // Relevant register: EXTI->IMR
    EXTI->IMR |= EXTI_IMR_MR1;
    /* Assign EXTI1 interrupt priority = 0 in NVIC */
    // Relevant register: NVIC->IP[2], or use NVIC_SetPriority
    NVIC_SetPriority(EXTI0_1_IRQn, 0);
    /* Enable EXTI1 interrupts in NVIC */
    // Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
    NVIC_EnableIRQ(EXTI0_1_IRQn);
}
```

Figure 19: The EXTI Initialization Code

The last two function blocks are the interrupt handlers (blocks of code that the processor will point its program counter at when an interrupt is generated); since both TIM2 and EXTI1 have been configured to generate interrupts, they both need a handler. Bit [0] of the status register in TIM2 will generate an interrupt flag (UIF=1) when an update even has happened (this bit is set by hardware, but must be cleared by writing UIF=0) [3, 8]. Earlier, the "myTIM2_Init" function (figure 18), TIM2 was configured to generate interrupts when the counter overflows so that the timer can be restarted (again setting CEN=1). The EXTI1 handler, handles events triggered by a rising edge from the 555 timer (as configured in figure 19, the "myEXTI_Init" function); when an interrupt has been generated the EXTI handler will confirm whether the rising edge that has been detected is the first, or second, edge (this will be used to measure the 555 signals period); if it is the first edge, the count register will be cleared and then begin counting; if the second edge has been detected, then the count register will be read and period will be calculated; the period is then used to calculate the frequency of the signal, store the frequency in a global variable called TIM2_Frequency seen in the "myLCD_Update" function (figure 9). After the interrupt has been handled in EXTI the pending flag must be set back to 0 by writing 1 into bit PR1 (associated with line 1) of the pending register. It is important to note that it is possible for interrupt flags to be set by other interrupt sources; the solution to confirming whether or not the right source generated the flag are the "if statements" used to check TIM2's status register and EXTI's pending regester before running the executing the rest of the handler's code.

```
/* 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;
    }
}
/* This handler is declared in system/src/cmsis/vectors_stm32f0xx.c */
void EXTI0_1_IRQHandler()
{
    // Declare/initialize your local variables here...
    //timerTriggered += (int) EXTI_PR_PR1;
    uint16_t T = 0;
    uint16_t f = 0;
    /* Check if EXTI1 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 &= ~(myTIM2_PERIOD);
            //  - Start timer (TIM2->CR1).
            TIM2->CR1 |= TIM_CR1_CEN;
            //    Else (this is the second edge):
            timerTriggered++;
        } else{
            //  - Stop timer (TIM2->CR1).
            TIM2->CR1 &= ~(TIM_CR1_CEN);
            //  - Read out count register (TIM2->CNT).
            T = (TIM2->CNT-1)/48;
            //  - Calculate signal period and frequency.
            f = 1000000/T -1;
            //  - 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.
            //TIM2_Frequency = 0x0000;
            TIM2_Frequency = f;
            trace_printf("Period: %u microseconds, Frequency:%u Hz\n", (uint16_t)T, (uint16_t)f);
            //
            // 2. Clear EXTI1 interrupt pending flag (EXTI->PR).
            // NOTE: A pending register (PR) bit is cleared
            // by writing 1 to it.
            timerTriggered--;
        }

        EXTI->PR |= (EXTI_PR_PR1);


    }
}
```

Figure 20: The TIM2 & EXTI Interrupt Handler Code

3. Testing & Results

During the testing period many possible configurations were used in a modular fashion; this enabled each block of code to be tested seperately until it was confirmed to be working. Issues initially thought to be strange and bewildering were eventually corrected.

3.1 The LCD Testing and Results

To test the LCD display ability, the ASCII code for each symbol was defined at the beginning of the project source code see figure 21). These values were used to test the LCD before any actual signals were being measured. These values were used by the “myLCD_Update” function in the early stages of its development (when it was being called directly from main without waiting for the ADC polling operation or EXTI interrupts).

```
/*ASCII codes*/
#define ASCII_F ((uint8_t)0x46)
#define ASCII_H ((uint8_t)0x48)
#define ASCII_R ((uint8_t)(0x52))
#define ASCII_COL ((uint8_t)0x3A)
#define ASCII_z ((uint8_t)0x7A)
#define ASCII_O ((uint8_t)0x4F)
#define ASCII_h ((uint8_t)0x68)
#define ASCII_0 ((uint8_t)0x30)
#define ASCII_1 ((uint8_t)0x31)
#define ASCII_2 ((uint8_t)0x32)
#define ASCII_3 ((uint8_t)0x33)
#define ASCII_4 ((uint8_t)0x34)
#define ASCII_5 ((uint8_t)0x35)
#define ASCII_6 ((uint8_t)0x36)
#define ASCII_7 ((uint8_t)0x37)
#define ASCII_8 ((uint8_t)0x38)
#define ASCII_9 ((uint8_t)0x39)
```

Figure 21: The ASCII Code Defines

At one point, when values were sent to the LCD (on the emulation board), the characters were showing up on the far-right side of the LCD (the result of which is shown in figure 22). In response to this issue, “trace_printf” functions were places after each line of code to print the values of the registers being configured and confirm that the proper bitwise operations were being conducted using the header file bit definitions [5]. This issue was resolved by correcting an improperly implemented handshake routine. This was, however, not the last issue with the LCD.

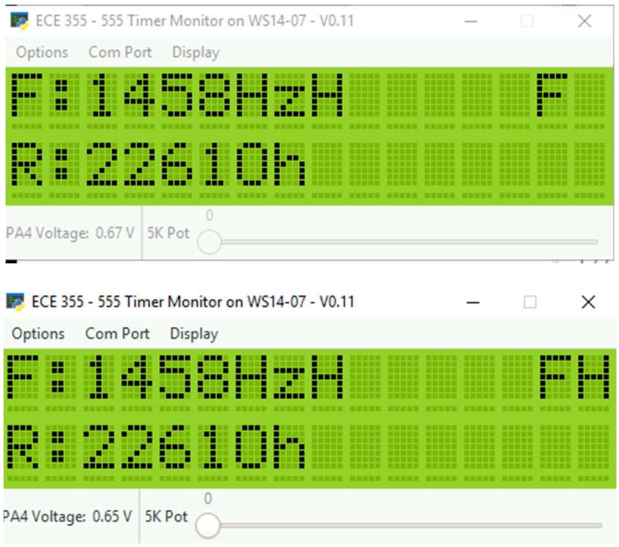

Figure 22: Improper LCD Display Configuration

The next issue revolving around the LCD was an ineffeciently slow incrementation from one character address to the next when updating the LCD. After attemps to configure the port B GPIO in highspeed mode, push-pull mode, different pull-up/pull-down variations (as shown by the “commented out” code in figure 6), more “trace_printf’s” were placed after each line in the LCD initialization, the LCD handshake, and the GPIO initialization; the code was then run in “debug” mode, and then paused at different stages so that the console could be checked for proper results. An example of some of the results are shown in figure 23. Eventually, the issue was found through process of elimination; the issue was determined to be some impropery configured pull-up/pull-down registers (PUPDR); an early version of the GPIOB initialization code is shown in figure 24 in which the PUPDR meant to be used for PB4-PB6 and PB8-PB15 were all set using the “GPIO_PUPDR_PUPDR5” bit declarations from the header file [5].

```
This is The Final Project!...
System clock: 48000000 Hz
GPIO -> ODR LCD Function Set : 3800
GPIOB I/O Data Registers Write Enable: 381f
Period: 17815 microseconds, Frequency:55 Hz
GPIOB I/O Data Registers LCD Slave assert done : 389f
GPIOB I/O Data Registers De-assert Enable: 388f
Period: 11812 microseconds, Frequency:83 Hz
GPIOB I/O Data Registers LCD Slave De-assert Done : 380f
GPIO -> ODR LCD Display On: c00
Period: 3971 microseconds, Frequency:250 Hz
GPIOB I/O Data Registers Write Enable: c1f
GPIOB I/O Data Registers LCD Slave assert done : c9f
Period: 13442 microseconds, Frequency:73 Hz
GPIOB I/O Data Registers De-assert Enable: c8f
GPIOB I/O Data Registers LCD Slave De-assert Done : c0f
Period: 20304 microseconds, Frequency:48 Hz
GPIO -> ODR LCD Entry Mode Set: 600
GPIOB I/O Data Registers Write Enable: 61f
Period: 10555 microseconds, Frequency:93 Hz
GPIOB I/O Data Registers LCD Slave assert done : 69f
GPIOB I/O Data Registers De-assert Enable: 68f
Period: 22059 microseconds, Frequency:44 Hz
GPIOB I/O Data Registers LCD Slave De-assert Done : 60f
GPIO -> ODR LCD Clear Display: 100
Period: 6727 microseconds, Frequency:147 Hz
GPIOB I/O Data Registers Write Enable: 11f
GPIOB I/O Data Registers LCD Slave assert done : 19f
Period: 14471 microseconds, Frequency:68 Hz
GPIOB I/O Data Registers De-assert Enable: 18f
GPIOB I/O Data Registers LCD Slave De-assert Done : 10f
Period: 14856 microseconds, Frequency:66 Hz
GPIOB I/O Data Registers Write Enable: 801f
GPIOB I/O Data Registers LCD Slave assert done : 809f
Period: 18001 microseconds, Frequency:54 Hz
GPIOB I/O Data Registers De-assert Enable: 808f
GPIOB I/O Data Registers LCD Slave De-assert Done : 800f
Period: 17350 microseconds, Frequency:56 Hz
GPIO -> ODR Line 1 Address: 8000
GPIOB I/O Data Registers Write Enable: 483f
```

Figure 23: Debugging with “trace_printf” Function

```
//Here I need to initialize GPIOB for use with the LCD
void myGPIOB_Init()
{
   /*Enable clock for GPIOB peripheral */
   //Relevant register: RCC->AHBENR
   RCC->AHBENR |= RCC_AHBENR_GPIOBEN;
   //Configure PB7 as an input
   GPIOB->MODER &= ~(GPIO_MODER_MODER7);
   /* Ensure no pull-up/pull-down for PB7 */
   GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR7);
   //Configure PB4-6 and PB8-15 as output
   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);
   /* Ensure no pull-up/pull-down for PB4-6 and PB8-15 - I think this is correct for this */
   GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR4 | GPIO_PUPDR_PUPDR5 | GPIO_PUPDR_PUPDR5 | GPIO_PUPDR_PUPDR5 |
         GPIO_PUPDR_PUPDR5 | GPIO_PUPDR_PUPDR5 | GPIO_PUPDR_PUPDR5 | GPIO_PUPDR_PUPDR5 | GPIO_PUPDR_PUPDR5 |
         GPIO_PUPDR_PUPDR5 | GPIO_PUPDR_PUPDR5 | GPIO_PUPDR_PUPDR5);
   /* Ensure push-pull mode selected for data pins*/
   //GPIOB->OTYPER &= ~(GPIO_OTYPER_OT_4 | GPIO_OTYPER_OT_5 | GPIO_OTYPER_OT_6 | GPIO_OTYPER_OT_8 | GPIO_OTYPER_OT_9 |
   //GPIO_OTYPER_OT_10 | GPIO_OTYPER_OT_11 | GPIO_OTYPER_OT_12 | GPIO_OTYPER_OT_13 | GPIO_OTYPER_OT_14 | GPIO_OTYPER_OT_15);
}
```

Figure 24: Early Version of The GPIOB Initialization Code

3.2 The ADC Testing and Results

This time around, a slightly more thorough research was conducted prior to writing the code. “Pseudo-code”, written in the form of comments, was used during the research phase prior to any actual code implementation. A “trace_printf” was used to print the values read from the ADC data values (this can be seen commented out in figure 13). The resulting ADC testing proved that successful code was written on the first attempt; this code was adjusted “here-and-there” to make sure it was robust, which it turned out to be. After this, a scaling factor of 1.25 (shown in figure 25) was used to adjust the resistance for display on the LCD (in order to remain consistent with that seen on the GUI’s POT slider).

```
/*ADC Scaling Factor*/
#define ADC_CONV_FACTOR 1.25
```

Figure 25: The ADC Scaling Factor

At this point in time, a hand calculator was used as a means to re-learn the math and logic behind the digit extraction implemented in figure 9 (“myLCD_Update”). This code was then implemented and the global “ADC_Data” variable was multiplied by the “ADC_CONV_FACTOR” (figure 25). The rest of the figure 9 code was adjusted to use the extracted digits and add 48 to each of them, as opposed to the ASCII “defines” shown in figure 21. The resulting resistance values were then displayed on the LCD and updated continuously in the loop shown in figure 13 (and located in the main routine). The resistance test results displayed on the GUI are shown in figure 26 in steps of approximately 1000 Ohms with an error typically between 50-90 Ohms (this value changed depending on the lab machine being used).

Figure 26: Incremental ADC Resistance Test

3.3 The DAC, EXTI & TIM2 Testing and Results

Similarly to the ADC, the configuration of GPIOA and the DAC went well. The same method of writing "psuedo-code" while doing research on the DAC were of great benefit and allowed the DAC to be implemented correctly on the first attempt (the code is seen in figures 14 and 17). A simple read and transfer of the ADC's data register into the DAC's data register in figure 13's loop resulted in PA4 voltage level changes displayed on the GUI as shown in figure 26 and 27. The voltage was observed to vary linearly from about 0.06-2.2 volts.

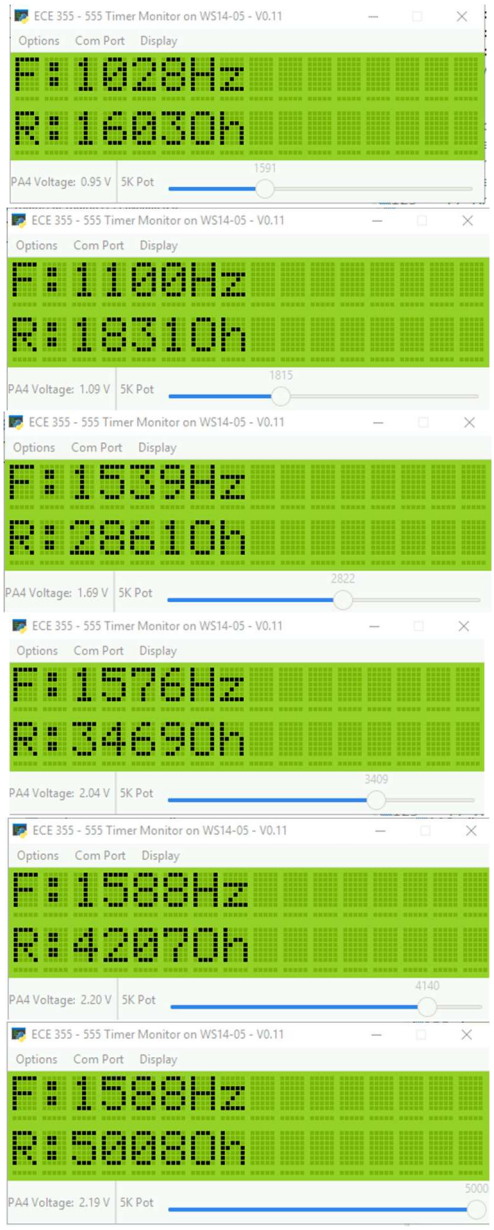

Figure 27: Incremental PA4 Voltage Test

The voltage of PA4 became clipped around 2.2 volts when the resistance value was only around the 4100 ohms range (a factor of approximately 1.25, which is on the order the factor used in the ADC). For experimental purposes, the data being read into the DAC's data register was divided by the the "ADC_CONV_FACTOR", as shown in an adjusted version of the infinite while loop (figure 28). The results of this test are shown in figure 29 where it can be seen that the voltage no longer becomes clipped around 4100 ohms; the result, is a PA4 voltage value that instead increases in a linear fashion when adjusting the POT slider from 0-5000 ohms.

```
while (1)
{
    //If the ADC conversion is complete, read the data register
    if((ADC1->ISR & ADC_ISR_EOC) != 0 ){
        //Read data register ADC->DR bits [11:0]
        ADC_Data = (ADC1->DR & ADC_DR_DATA);
        //trace_printf("The data reads: %u \n", (uint16_t)((ADC_Data)*ADC_CONV_FACTOR));
        //DAC register in use DAC->DHR12R1 (12 bit register for right-aligned) DAC_DHR12R1_DACC1DHR
        //Read from ADC1_DR and then send to DAC_DHR12R1 (i.e. DAC->DH12R1 = ADC1->DR)
        //trace_printf("The DAC Register: %x \n", (uint16_t)(DAC->DHR12R1));
        //ADC_Data = ADC_Data*1.9;


        //DAC->DHR12R1 = (ADC_Data & DAC_DHR12R1_DACC1DHR); -
        DAC->DHR12R1 = (ADC_Data & DAC_DHR12R1_DACC1DHR)/ADC_CONV_FACTOR;
        DAC_Data = (DAC->DHR12R1);
        //DAC_Data =
        //trace_printf("The data passed from ADC1 to DHR12R1 is: %u \n", (uint16_t)(DAC_Data));
        myLCD_Update();

    }

}
```

Figure 28: The Adjusted Infinite “while” Loop

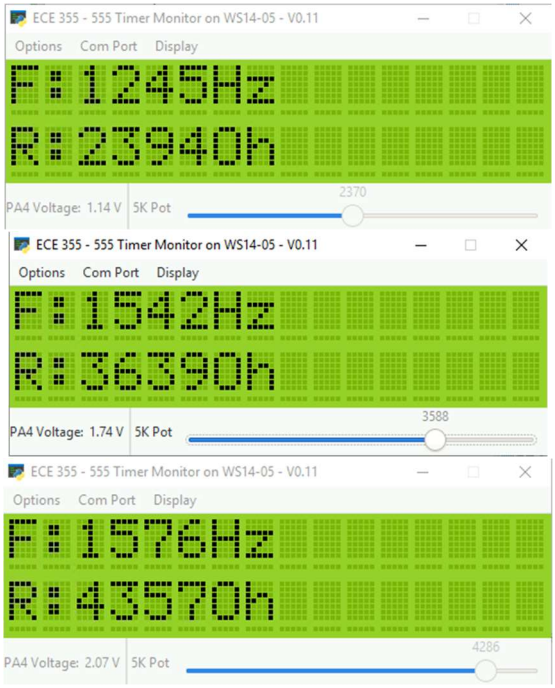

Figure 29: Scaled PA4 Voltage Test

The images in figures 27-29 were obviously taken once the whole project was working together; however, the frequency measurement component of the project had some issues worth discussing. Many hours were spent observing a low frequency in the range of 40-60 Hz with sporatic and extreme spikes. The entire project was pulled apart from the very beginning and blocks of code were added back one at a time. The EXTI and TIM2 initializations and interrupt handlers were changed back to previously used versions of the code that measured the frequency from a function generator instead of the 555 timer; this version of the code appeared to be working and was able to be configured to display on the LCD. This was until a block of code containing “trace_printf” functions was added back into the code; thus, the code was fully reassembled with all of the “trace_printf” functions removed except for one located in the EXTI interrupt handler. The resulting code ran with no issues and the frequency results seen in figures 27-29 were observed.

The resulting frequencies measured (figures 27-29) by the TIM2 counter varied (approximately) between 1000-1600 Hz. The more interesting observation occurs when viewing figure 28; the frequency remains approximately the same while PA4 is between 0-0.95 volts, then increases from dramatically between 0.95-1.7 volts, and then remains approximately the same again between 1.7-2.2 volts.

4. Discussion

The project proved that it can be accomplished, whilst meeting the design specifications, in a reasonable amount of time. An important lesson was learned in regards to doing the necessary research and writing “pseudo-code” before trying to implement and run any hastily written code. It is of great importance to note that when implementing and testing the frequency measuring component of the project, that all “trace_printf” functions be commented out of the source code.

It is believed that the observations made in regards to the frequency changes only taking affect when the PA4 voltage is in between 0.9-1.7 volts is due to the forward voltage of the optocouplers input [9] as seen in figure 30. The images in figure 27 were deliberately taken around these voltages to demonstrate this.

ELECTRICAL CHARACTERISTICS (1)

PARAMETER	TEST CONDITION	PART	SYMBOL	MIN.	TYP.	MAX.	UNIT
INPUT							
Junction capacitance	V_R = 0 V, f = 1 MHz		C_j		50		pF
Forward voltage (2)	I_F = 10 mA		V_F		1.3	1.5	V
	I_F = 10 mA, T_{amb} = - 55 °C		V_F	0.9	1.3	1.7	V
Reverse current (2)	V_R = 6 V		I_R		0.1	10	µA
Capacitance	V_R = 0 V, f = 1 MHz		C_O		25		pF

Figure 30: Electrical Characteristics from 4N35 Optocoupler Datasheet [9]

The clipping affect (at 2.2 volts) seen on the PA4 voltage may be the product of the DAC electrical specifications (located in section 6.3.17 of [4]). The electrical specifications point to a required minimum value of 2.4 volts being applied to VDDA [4]; VDDA is shown in the block diagram of figure 16 along with the equation to calculate the “DACoutput”. The electrical specifications of the DAC also state that the maximum output of DAC (“DAC_OUT max”) is VDDA-0.2 volts; this appears to correspond with an operational VDDA voltage of the minimum 2.4 volts (subtracting 0.2 volts from this gives 2.2 volts, which is exactly what appeared on the PA4 voltage).

The last observation I made, which I would like to answer is in regards to the frequency characteristics of the 555 timer. A solid conclusion as to why the timer supplies a 1000 Hz frequency when the voltage at PA4 is below the optocouplers forward voltage minimum (and hence has no effect on the optocouplers output). This infers that the timer supplies a 1000 Hz frequency even when the optocoupler is not driving the timer. Simarly, a conclusion was not reached in regards to the variation of frequency between the approximate 1000-1600 Hz. Based on the information in the datasheet for the 555 timer [10], and the information in Professor Rakhmatov’s “Interface Examples” [3], it appears that the 555 timer is configured in “astable operation [and] free runs as a multi-vibrator” [10]; yet, the datasheet appears to claim that when operated in this mode, the frequency is a function of the resistor values (which are not changing via adjustments from the optocoupler).

In conclusion, I believe a lot was learned during this project. I have gained much more proficiency in embedded programming, reading datasheets, and searching through manuals. I enjoy doing home projects with single-board computers such as the Raspberry Pi, as well as, microcontrollers such as the Arduino; maybe the next project I try should use an STM microcontroller!

5. Appendix

```
//
// 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)
/*LCD Function Set */
//#define myLCD_FUNCTION_SET ((uint16_t)0x0038)
/*LCD Display On */
//#define myLCD_DISPLAY_ON ((uint16_t)0x000C)
/*LCD Entry Mode Set */
//#define myLCD_ENTRY_MODE_SET ((uint16_t)0x0006)
/*LCD Clear Display */
//#define myLCD_CLEAR_DISPLAY ((uint16_t)0x0001)
/*ASCII codes*/
#define ASCII_F ((uint8_t)0x46)
#define ASCII_H ((uint8_t)0x48)
#define ASCII_R ((uint8_t)(0x52))
#define ASCII_COL ((uint8_t)0x3A)
#define ASCII_z ((uint8_t)0x7A)
#define ASCII_O ((uint8_t)0x4F)
#define ASCII_h ((uint8_t)0x68)
#define ASCII_0 ((uint8_t)0x30)
#define ASCII_1 ((uint8_t)0x31)
#define ASCII_2 ((uint8_t)0x32)
#define ASCII_3 ((uint8_t)0x33)
#define ASCII_4 ((uint8_t)0x34)
#define ASCII_5 ((uint8_t)0x35)
#define ASCII_6 ((uint8_t)0x36)
#define ASCII_7 ((uint8_t)0x37)
#define ASCII_8 ((uint8_t)0x38)
#define ASCII_9 ((uint8_t)0x39)
/*LCD DDRAM Addresses*/
#define LCD_LINE_1 ((uint8_t)0x80)
#define LCD_LINE_2 ((uint8_t)0x40)
/*ADC Scaling Factor*/
#define ADC_CONV_FACTOR 1.25
```

```
75
76
77  void myGPIOB_Init(void);
78  void myLCD_Init(void);
79  void myLCD_Handshake(void);
80  void myLCD_Update(void);
81  void myDAC_Init(void);
82  void myGPIOC_Init(void);
83  void myADC_Init(void);
84  void myGPIOA_Init(void);
85  void myTIM2_Init(void);
86  //can use this for a delay for the LCD or whatever delay he said we may need to introduce
87  //void myTIM3_Init(void);
88  void myEXTI_Init(void);
89
90  //Global variable to hold what is read from ADC
91  volatile uint16_t ADC_Data = ((uint16_t)0x0000);
92  //Global variable to hold what is read from DAC
93  volatile uint16_t DAC_Data = ((uint16_t)0x0000);
94  //Global variable to hold calculated frequency from TIM2
95  volatile uint16_t TIM2_Frequency = ((uint16_t)0x0000);
96
97  //will most likely need a global variable here to store the data from the ADC to be sent to the LCD
98  //this global variable will tell us what numbers we need to send to the LCD and we can implement this in
99  //an if/else type manner
100 //this will be in the LCD handshake or something like that
101
102 // Declare/initialize your global variables here...
103 // NOTE: You'll need at least one global variable
104 // (say, timerTriggered = 0 or 1) to indicate
105 // whether TIM2 has started counting or not.
106 int timerTriggered = 0;
107
108 int
109 main(int argc, char* argv[])
110 {
111
112     trace_printf("This is The Final Project!...\n");
113     trace_printf("System clock: %u Hz\n", SystemCoreClock);
114
115     myGPIOA_Init();     /* Initialize I/O port PA (for Timer Output and Optocoupler)*/
116     myGPIOB_Init();     /* Initialize I/O port PB (for LCD) */
117     myGPIOC_Init();     /* Initialize I/O port PC (for ADC and POT)  */
118     myTIM2_Init();      /* Initialize timer TIM2 */
119     //myTIM3_Init();        /* Initialize timer TIM3 */
120     myEXTI_Init();      /* Initialize EXTI */
121     myLCD_Init();       /* Initialize LCD */
122     myADC_Init();
123     myDAC_Init();
124
125     //Will need to use this for the handshake with the LCD (for the LCD though I think that this can just be including
126     //into the ADC part - the ADC can then just wait to make sure the handshake is complete and then it can send to the LCD)
127     //as well as for polling the ADC (which will wait for an input from the POT on PC1) and for the DAC (which will wait
128     //for an input from the ADC)
129
130
131     while (1)
132     {
133         //If the ADC conversion is complete, read the data register
134         if((ADC1->ISR & ADC_ISR_EOC) != 0 ){
135             //Read data register ADC->DR bits [11:0]
136             ADC_Data = (ADC1->DR & ADC_DR_DATA);
137             //trace_printf("The data reads: %u \n", (uint16_t)((ADC_Data)*ADC_CONV_FACTOR));
138             //DAC register in use DAC->DHR12R1 (12 bit register for right-aligned) DAC_DHR12R1_DACC1DHR
139             //Read from ADC1_DR and then send to DAC_DHR12R1 (i.e. DAC->DH12R1 = ADC1->DR)
140             //trace_printf("The DAC Register: %x \n", (uint16_t)(DAC->DHR12R1));
141             //ADC_Data = ADC_Data*1.9;
142
143             //DAC->DHR12R1 = (ADC_Data & DAC_DHR12R1_DACC1DHR); -
144             DAC->DHR12R1 = (ADC_Data & DAC_DHR12R1_DACC1DHR);
145             DAC_Data = (DAC->DHR12R1);
146             //DAC_Data =
147             //trace_printf("The data passed from ADC1 to DHR12R1 is: %u \n", (uint16_t)(DAC_Data));
148             myLCD_Update();
149
150         }
151
152     }
153
154     return 0;
155
156 }
```

```
157
158 void myADC_Init()
159 {
160     /*Enable clock for the ADC*/
161     RCC->APB2ENR |= RCC_APB2ENR_ADC1EN;
162     //Make sure all the bits in the control register are zero (Bit [0]: ADEN can only be set if all bits of ADC_CR are 0)
163     ADC1->CR &= 0x00000000;
164     //Enable the ADC - Bit [0]: ADEN set to 1 (Note: this bit is cleared by hardware after ADDIS bit is set - ADDIS is a disable bit - i.e ADDIS=1=DISABLE)
165     //Another note on ADDIS - once set to 1, the ADC will begin to shut down... once shut down it will clear ADEN and ADDIS will be cleared by hardware as a result
166     ADC1->CR |= ADC_CR_ADEN;
167
168     //Set the SMP[2:0] all to 1 - a sampling time to an intermediate value of 239 1/2 ADC clock cycles - Again, software can write these bits only when ADSTART=0
169     ADC1->SMPR |= ADC_SMPR_SMP;
170     //Set the channel selection register to channel 11 (since it is internally connected to PC1)
171     ADC1->CHSELR |= ADC_CHSELR_CHSEL11;
172
173     //Bit alignment is by default set to 0 which is right data alignment (this is what we want so do nothing)
174     //Resolution is by default set to RES=00 which is exactly what we want so no need to change (do nothing)
175
176     //Set the overrun management mode to 1: ADC_DR contents are preserved/overwritten when an overrun is detected and thus we will always read the freshest value
177     ADC1->CFGR1 |= ADC_CFGR1_OVRMOD;
178
179     //Set Bit[13]: CONT (continuous conversion mode - everytime it finishes converting it will continue running)
180     ADC1->CFGR1 |= ADC_CFGR1_CONT;
181
182     //Now, the ADC will continue converting due to the configurations made. The EOC flag will become 1 when a channel conversion is complete
183     //thus EOC is the bit that needs to be polled by software - harware will clear this bit when ADC_DR is read and thus we will always read the freshest value
184     //with no confusion overrun stuff (since overrun mode is set) ADC_ISR_EOC
185
186     //Wait for the ADC ready flag (Bit[0]: ADRDY) - this will tell us that initialization is complete
187     while((ADC1->ISR & ADC_ISR_ADRDY) == 0);
188     //trace_printf("ADRDY is now 1 and the ADC IS READY!\n");
189
190     //Once ADRDY is set 1, the ADC is ready to start converting
191     /*should maybe include this stuff in the main?
192     //Make sure ADDIS set to zero to allow ADSTART to be set and start the conversion process (most likely an unessecary step due to previous AND operation)
193     ADC1->CR &= ~(ADC_CR_ADDIS);
194     //Start the conversion process - this will stay 1 since we will just let it keep converting
195     //Note: ADSTART is cleared by hardware when the conversion process is finished or stopped (but again.. we will let it keep converting)*/
196     ADC1->CR |= ADC_CR_ADSTART;
197 }
198
199 //Here I need to initialize GPIOB for use with the LCD
200 void myGPIOB_Init()
201 {
202     /*Enable clock for GPIOB peripheral */
203     //Relevant register: RCC->AHBENR
204     RCC->AHBENR |= RCC_AHBENR_GPIOBEN;
205     //Configure PB7 as an input
206     GPIOB->MODER &= ~(GPIO_MODER_MODER7);
207     /* Ensure no pull-up/pull-down for PB7 */
208     GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR7);
209     //Configure PB4-6 and PB8-15 as output
210     GPIOB->MODER |= (GPIO_MODER_MODER4_0 | GPIO_MODER_MODER5_0 | GPIO_MODER_MODER6_0 | GPIO_MODER_MODER8_0 |
211             GPIO_MODER_MODER9_0 | GPIO_MODER_MODER10_0 | GPIO_MODER_MODER11_0 | GPIO_MODER_MODER12_0 |
212             GPIO_MODER_MODER13_0 | GPIO_MODER_MODER14_0 | GPIO_MODER_MODER15_0);
213     /* Ensure no pull-up/pull-down for PB4-6 and PB8-15 - I think this is correct for this */
214     GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR4 | GPIO_PUPDR_PUPDR5 | GPIO_PUPDR_PUPDR6 | GPIO_PUPDR_PUPDR8 |
215             GPIO_PUPDR_PUPDR9 | GPIO_PUPDR_PUPDR10 | GPIO_PUPDR_PUPDR11 | GPIO_PUPDR_PUPDR12 | GPIO_PUPDR_PUPDR13 |
216             GPIO_PUPDR_PUPDR14 | GPIO_PUPDR_PUPDR15);
217     /* Ensure high-speed mode for PC8 and PC9 */
218     //GPIOB->OSPEEDR |= (GPIO_OSPEEDER_OSPEEDR4 | GPIO_OSPEEDER_OSPEEDR5 | GPIO_OSPEEDER_OSPEEDR6 | GPIO_OSPEEDER_OSPEEDR8 |
219     //   GPIO_OSPEEDER_OSPEEDR9 | GPIO_OSPEEDER_OSPEEDR10 | GPIO_OSPEEDER_OSPEEDR11 | GPIO_OSPEEDER_OSPEEDR12 |
220     //          GPIO_OSPEEDER_OSPEEDR13 | GPIO_OSPEEDER_OSPEEDR14 | GPIO_OSPEEDER_OSPEEDR15);
221     /* Ensure push-pull mode selected for data pins*/
222     //GPIOB->OTYPER &= ~(GPIO_OTYPER_OT_4 | GPIO_OTYPER_OT_5 | GPIO_OTYPER_OT_6 | GPIO_OTYPER_OT_8 | GPIO_OTYPER_OT_9 |
223     //GPIO_OTYPER_OT_10 | GPIO_OTYPER_OT_11 | GPIO_OTYPER_OT_12 | GPIO_OTYPER_OT_13 | GPIO_OTYPER_OT_14 | GPIO_OTYPER_OT_15);
224 }
225
```

```
225
226  //Here I need to initialize GPIOC for use with the ADC and POT
227  void myGPIOC_Init()
228  {
229     /*Enable clock for GPIOC peripheral */
230     //Relevant register: RCC->AHBENR
231     RCC->AHBENR |= RCC_AHBENR_GPIOCEN;
232     //Configure PC1 as analog
233     GPIOC->MODER |= GPIO_MODER_MODER1;
234     /*Ensure no pull-up/pull-down for PC1 - this should be correct for this I think */
235     GPIOC->PUPDR &= ~(GPIO_PUPDR_PUPDR1);
236  }
237
238  //Master-Slave Handshake
239  void myLCD_Handshake(){
240
241      //Note that: #define GPIO_ODR_4        ((uint32_t)0x00000010)
242      //and #define GPIO_ODR_7               ((uint32_t)0x00000080) and thus,
243      //trace_printf will output 9
244      //Assert master enable (PB[4]=1)
245      GPIOB->ODR |= GPIO_ODR_4;
246      //trace_printf("GPIOB I/O Data Registers Write Enable: %x \n", (uint16_t)GPIOB->IDR );
247      //Wait for LCD (slave) to assert done (PB[7]=1)
248      while((GPIOB->IDR & GPIO_IDR_7) == 0);
249      //trace_printf("GPIOB I/O Data Registers LCD Slave assert done : %x \n", (uint16_t)GPIOB->IDR );
250      //De-assert master enable (PB[4]=0)
251      GPIOB->ODR &= ~(GPIO_ODR_4);
252      //trace_printf("GPIOB I/O Data Registers De-assert Enable: %x \n", (uint16_t)GPIOB->IDR );
253      //Wait for LCD (slave) to de-assert done (PB[7]=0)
254      while((GPIOB->IDR & GPIO_IDR_7) !=0);
255      //trace_printf("GPIOB I/O Data Registers LCD Slave De-assert Done : %x \n", (uint16_t)GPIOB->IDR );
256
257  }
258
```

```
//Refresh the LCD with new values
void myLCD_Update()
{
    //Divide the ADC data into ones, tens, hundreds, thousands for display
    uint16_t resistance = ADC_Data*ADC_CONV_FACTOR;
    uint16_t resistance_thousands = (resistance)/1000;
    uint16_t resistance_hundreds = ((resistance)-(resistance_thousands*1000))/100;
    uint16_t resistance_tens = ((resistance)-(resistance_thousands*1000)-(resistance_hundreds*100))/10;
    uint16_t resistance_ones = ((resistance)-(resistance_thousands*1000)-(resistance_hundreds*100)-(resistance_tens*10))/1;
    //Divid the DAC data into ones, tens, hundreds, thousands for display
    unsigned int frequency = TIM2_Frequency;
    unsigned int frequency_thousands = (frequency)/1000;
    unsigned int frequency_hundreds = ((frequency)-(frequency_thousands*1000))/100;
    unsigned int frequency_tens = ((frequency)-(frequency_thousands*1000)-(frequency_hundreds*100))/10;
    unsigned int frequency_ones = ((frequency)-(frequency_thousands*1000)-(frequency_hundreds*100)-(frequency_tens*10))/1;


   //Set the LCD's DDRAM 7 bit address you will write to (here RS=0 and R/W=0, DB7=1 and DB6-0=A[6:0] - A for address)

   //Write 8-bits of data D[7:0] into LCD's DDRAM (here RS=1 and R/W=0 and DB7-0=D[7:0](ASCII code)) - each ASCII is written seperately
   //Note: cannot read from DDRAM (R/W is always 0)
   //Note: DDRAM address is incremented for you every time - only have to set the address twice (once for each line)

    //Set DDRAM address for line 1 PB5=RS=0, PB6=R/W=0, PB15=DB7=1, PB8,PB9,...,PB14=D0,D1,...,D6=0
  GPIOB->ODR = (~(GPIO_ODR_5 | GPIO_ODR_6 )&(GPIO_ODR_15));
  myLCD_Handshake();
  //trace_printf("GPIO -> ODR Line 1 Address: %x \n", (uint16_t)GPIOB->ODR);

  //Write data to DDRAM
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|(ASCII_F<<8));
  //trace_printf("GPIO -> ODR Write H : %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|(ASCII_COL<<8));
  //trace_printf("GPIO -> ODR Write F : %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|((frequency_thousands+48)<<8));
  //trace_printf("GPIO -> ODR Write 0: %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|((frequency_hundreds+48)<<8));
  //trace_printf("GPIO -> ODR Write 1: %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|((frequency_tens+48)<<8));
  //trace_printf("GPIO -> ODR Write 2: %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|((frequency_ones+48)<<8));
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|(ASCII_H<<8));
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|(ASCII_z<<8));
  myLCD_Handshake();

  //Set DDRAM address for line 2 PB5=RS=0, PB6=R/W=0, PB15=DB7=1, PB14=DB6=1 PB8,PB9,...,PB14=D0,D1,...,D6=0
  GPIOB->ODR = (~(GPIO_ODR_5 | GPIO_ODR_6 )&(GPIO_ODR_15|GPIO_ODR_14));
  myLCD_Handshake();
  //trace_printf("GPIO -> ODR Line 2 Address: %x \n", (uint16_t)GPIOB->ODR);
  //Write data to DDRAM
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|(ASCII_R<<8));
  //trace_printf("GPIO -> ODR Write H : %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|(ASCII_COL<<8));
  //trace_printf("GPIO -> ODR Write F : %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|((resistance_thousands+48)<<8));
  //trace_printf("GPIO -> ODR Write 0: %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|((resistance_hundreds+48)<<8));
  //trace_printf("GPIO -> ODR Write 1: %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|((resistance_tens+48)<<8));
  //trace_printf("GPIO -> ODR Write 2: %x \n", (uint16_t)GPIOB->ODR);
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|((resistance_ones+48)<<8));
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|(ASCII_0<<8));
  myLCD_Handshake();
  GPIOB->ODR = (((GPIO_ODR_5)&(~GPIO_ODR_6))|(ASCII_h<<8));
  myLCD_Handshake();
}
```

```
337
338 void myLCD_Init()
339 {
340
341   //Function set - PB13=DB5=1, PB12=DB4=1, PB11=DB3=1, PB10=DB2=0
342   GPIOB->ODR = ((GPIO_ODR_13 | GPIO_ODR_12 | GPIO_ODR_11)&(~(GPIO_ODR_10)));
343   //trace_printf("GPIO -> ODR LCD Function Set : %x \n", (uint16_t)GPIOB->ODR);
344   myLCD_Handshake();
345   //Display on - PB11=DB3=1, PB10=D2=1, PB9=DB1=0, PB8=DB0=0
346   GPIOB->ODR = ((GPIO_ODR_11 | GPIO_ODR_10 )&(~(GPIO_ODR_9 | GPIO_ODR_8 )));
347   //trace_printf("GPIO -> ODR LCD Display On: %x \n", (uint16_t)GPIOB->ODR);
348   myLCD_Handshake();
349   //Entry mode set - PB10=DB=1, PB9=DB1=1, PB8=DB0=0
350   GPIOB->ODR = ((GPIO_ODR_10 | GPIO_ODR_9)&(~(GPIO_ODR_8)));
351   //trace_printf("GPIO -> ODR LCD Entry Mode Set: %x \n", (uint16_t)GPIOB->ODR);
352   myLCD_Handshake();
353   //Clear display - PB8=DB0=1
354   GPIOB->ODR = GPIO_ODR_8;
355   //trace_printf("GPIO -> ODR LCD Clear Display: %x \n", (uint16_t)GPIOB->ODR);
356   myLCD_Handshake();
357
358 }
359
360
361 //this will be used to measure the frequency of the timer output
362 //will also need to initialize PA4 for the DAC
363 void myGPIOA_Init()
364 {
365     /* Enable clock for GPIOA peripheral */
366     // Relevant register: RCC->AHBENR
367     RCC->AHBENR |= RCC_AHBENR_GPIOAEN;
368
369     /* Configure PA1 as input */
370     // Relevant register: GPIOA->MODER
371     GPIOA->MODER &= ~(GPIO_MODER_MODER1);
372     /* Ensure no pull-up/pull-down for PA1 */
373     // Relevant register: GPIOA->PUPDR
374     GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR1);
375
376     /*Configure PA4 as analog */
377     GPIOA->MODER |= GPIO_MODER_MODER4;
378     /*Ensure no pull-up/pull-down for PA4*/
379     //GPIOA->PUPDR |= GPIO_PUPDR_PUPDR4;
380     GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR4);
381     //Ensure push-pull
382     //GPIOA->OTYPER &= ~(GPIO_OTYPER_OT_4);
383     //Here, you need to initialize pin 4 for the DAC
384     //GPIOA->OSPEEDR |= GPIO_OSPEEDER_OSPEEDR4;
385 }
386
387 void myDAC_Init()
388 {
389     /* Enable clock for the DAC*/
390     RCC->APB1ENR |= RCC_APB1ENR_DACEN;
391     //DACoutput=VDDA*DOR/4096
392     //DAC register in use DAC->DHR12R1 (12 bit register for right-aligned) DAC_DHR12R1_DACC1DHR
393     //Read from ADC1_DR and then send to DAC_DHR12R1 (i.e. DAC->DH12R1 = ADC1->DR)
394     //DAC control register (DAC->CR) - Bit[0]:EN1 (Channell enable) - must configure PA4 as analog before you enable (RCC too)
395     DAC->CR &= 0x00000000;
396     DAC->CR |= DAC_CR_EN1;
397     //Bit[1]:BB0FF1=0/1 (channell tri-state buffer is enabled/disabled) - need this set to 0 (should already be 0 by default)
398     DAC->CR &= ~(DAC_CR_BOFF1);
399     //Bit[2]:TEN1 (channell trigger enable)=0/1: - need this set to 0 (should already be 0 by default)
400     DAC-> CR &= ~(DAC_CR_TEN1);
401 }
402
```

```
403
404 void myTIM2_Init()
405 {
406     /* Enable clock for TIM2 peripheral */
407     // Relevant register: RCC->APB1ENR
408     RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;
409
410     /* Configure TIM2: buffer auto-reload, count up, stop on overflow,
411      * enable update events, interrupt on overflow only */
412     // Relevant register: TIM2->CR1
413     TIM2->CR1 = ((uint16_t)0x008C);
414
415     /* Set clock prescaler value */
416     TIM2->PSC = myTIM2_PRESCALER;
417     /* Set auto-reloaded delay */
418     TIM2->ARR = myTIM2_PERIOD;
419
420     /* Update timer registers */
421     // Relevant register: TIM2->EGR
422     TIM2->EGR = ((uint16_t)0x0001);
423
424     /* Assign TIM2 interrupt priority = 0 in NVIC */
425     // Relevant register: NVIC->IP[3], or use NVIC_SetPriority
426     NVIC_SetPriority(TIM2_IRQn, 0);
427
428     /* Enable TIM2 interrupts in NVIC */
429     // Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
430     NVIC_EnableIRQ(TIM2_IRQn);
431
432     /* Enable update interrupt generation */
433     // Relevant register: TIM2->DIER
434     TIM2->DIER |= TIM_DIER_UIE;
435     /* Start counting timer pulses */
436     TIM2->CR1 |= TIM_CR1_CEN;
437
438 }
439
440
441 void myEXTI_Init()
442 {
443     /* Map EXTI1 line to PA1 */
444     // Relevant register: SYSCFG->EXTICR[0]
445     //** I dont know if this is how we fill EXTI1 with zeros
446     //SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI1_PA;
447     SYSCFG->EXTICR[0]|= SYSCFG_EXTICR1_EXTI1_PA;
448
449     /* EXTI1 line interrupts: set rising-edge trigger */
450     // Relevant register: EXTI->RTSR
451     EXTI->RTSR |= EXTI_RTSR_TR1;
452     /* Unmask interrupts from EXTI1 line */
453     // Relevant register: EXTI->IMR
454     EXTI->IMR |= EXTI_IMR_MR1;
455     /* Assign EXTI1 interrupt priority = 0 in NVIC */
456     // Relevant register: NVIC->IP[2], or use NVIC_SetPriority
457     NVIC_SetPriority(EXTI0_1_IRQn, 0);
458     /* Enable EXTI1 interrupts in NVIC */
459     // Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
460     NVIC_EnableIRQ(EXTI0_1_IRQn);
461 }
462
```

```
463
464 /* This handler is declared in system/src/cmsis/vectors_stm32f0xx.c */
465 void TIM2_IRQHandler()
466 {
467     /* Check if update interrupt flag is indeed set */
468     if ((TIM2->SR & TIM_SR_UIF) != 0)
469     {
470         trace_printf("\n*** Overflow! ***\n");
471
472         /* Clear update interrupt flag */
473         // Relevant register: TIM2->SR
474         TIM2->SR &= ~(TIM_SR_UIF);
475         /* Restart stopped timer */
476         // Relevant register: TIM2->CR1
477         TIM2->CR1 |= TIM_CR1_CEN;
478     }
479 }
480
481
482 /* This handler is declared in system/src/cmsis/vectors_stm32f0xx.c */
483 void EXTI0_1_IRQHandler()
484 {
485     // Declare/initialize your local variables here...
486     //timerTriggered += (int) EXTI_PR_PR1;
487     uint16_t T = 0;
488     uint16_t f = 0;
489     /* Check if EXTI1 interrupt pending flag is indeed set */
490     if ((EXTI->PR & EXTI_PR_PR1) != 0)
491     {
492         //
493         // 1. If this is the first edge:
494         if (timerTriggered == 0){
495             //  - Clear count register (TIM2->CNT).
496             TIM2->CNT &= ~(myTIM2_PERIOD);
497             //  - Start timer (TIM2->CR1).
498             TIM2->CR1 |= TIM_CR1_CEN;
499             //    Else (this is the second edge):
500             timerTriggered++;
501         } else{
502             //  - Stop timer (TIM2->CR1).
503             TIM2->CR1 &= ~(TIM_CR1_CEN);
504             //  - Read out count register (TIM2->CNT).
505             T = (TIM2->CNT-1)/48;
506             //  - Calculate signal period and frequency.
507             f = 1000000/T -1;
508             //  - Print calculated values to the console.
509             //    NOTE: Function trace_printf does not work
510             //    with floating-point numbers: you must use
511             //    "unsigned int" type to print your signal
512             //    period and frequency.
513             //TIM2_Frequency = 0x0000;
514             TIM2_Frequency = f;
515             trace_printf("Period: %u microseconds, Frequency:%u Hz\n", (uint16_t)T, (uint16_t)f);
516             //
517             // 2. Clear EXTI1 interrupt pending flag (EXTI->PR).
518             // NOTE: A pending register (PR) bit is cleared
519             // by writing 1 to it.
520             timerTriggered--;
521         }
522
523         EXTI->PR |= (EXTI_PR_PR1);
524
525
526     }
527 }
528
529
530 #pragma GCC diagnostic pop
531
532 // ----------------------------------------------------------------------------
```

6. References

[1] B. Sirna, K. Kelany, D.N. Rakhmatov. University of Victoria. ECE 355: Microprocessor-Based Systems Laboratory Manual [Online]. (2020). Available: https://www.ece.uvic.ca/~ece355/lab/ECE355-LabManual-2020.pdf

[2] D.N. Rakhmatov. (2020). I/O Examples [Online]. Available: https://www.ece.uvic.ca/~daler/courses/ece355/iox.pdf

[3] D.N. Rakhmatov. (2020). Interface Examples [Online]. Available: https://www.ece.uvic.ca/~daler/courses/ece355/interfacex.pdf

[4] STMicroelectronics, "STM32F051xx", 022265 Rev 4 datasheet, Jan. 2014, https://www.ece.uvic.ca/~ece355/lab/supplement/stm32f051r8t6_datatsheet_DM00039193.pdf (accessed Dec. 3, 20)

[5] MCD Application Team. (2014). "stm32f0xx.h" (Version 1.3.1) [Source Code]. Available: https://www.ece.uvic.ca/~ece355/lab/code/stm32f0xx.h

[6] Hitachi, "HD44780U (LCD-II)", ADE-207-272(Z) Rev 0 datasheet, 1998, https://www.ece.uvic.ca/~ece355/lab/supplement/HD44780.pdf (accessed Dec. 3, 20)

[7] D.N. Rakhmatov. (2020). Interfacing [Online]. Available: https://www.ece.uvic.ca/~daler/courses/ece355/interface.pdf

[8] STMicroelectronics, "RM0091 Reference Manual", 018940 Rev 5 datasheet, Jan. 2014, https://www.ece.uvic.ca/~ece355/lab/supplement/stm32f0RefManual.pdf (accessed Dec. 3, 20)

[9] 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 Dec. 3, 20)

[10] STMicroelectronics, "General-purpose single bipolar timers", 2182 Rev 6 datasheet, Jan. 2012, https://www.ece.uvic.ca/~ece355/lab/supplement/555timer.pdf (accessed Dec. 3, 20)