Report 2: Motor Labs

Student 1, Student 2, Student 3

Syracuse University

Introduction

In the following experiments, we learn about the PIC16F1769 microcontroller and how to program it through projects with several types of motors. For the experiments we are tasked with interfacing a stepper motor with parallel control, a DC motor with analog control, and a servo motor with potentiometer control. The code is written in c using MPLAB and a snap programmer is used for the microcontroller to understand the code and produce the desired results. Analog Discovery 2 is used to send various signals to our circuit, examine input and output, and power the circuit. A “power brick” is also sometimes used to power the circuit.

Experiment 5 Stepper Motor

Introduction

The goal of this experiment is to introduce the PIC microcontroller, Stepper motors, and transistor drivers with several control input and position control. The PIC's timer uses a separate hardware system called the Capture, Compare, PWM (CCP) unit for timer comparison. Setup of the timer, ccp unit, and interrupt are shown below in the Experimental Setup. A stepper motor consists of a free moving shaft with a permanent magnet surrounded by electromagnet coils attached to the case.. The drawing in Figure 1.7 makes this setup easier to understand. When the coils are charged, they make north-south magnetic poles, depending on the direction of the current. Our motor is unipolar. The motor moves when we charge the coil in order and the permanent magnet aligns to the poles of each newly charged magnet. Stepper motors usually need more voltage than a GPIO pin can provide so we will be using a transistor driver circuit.

Materials

- PIC16F1769
- Snap Programmer
- Analog Discovery 2
- Servo Motor
- Wires
- "Power Brick"
- ZVN2106A transistor, 2N7000 transistor, or ULN transistor array

Experimental Setup

Wire the Snap programmer to the PIC microcontroller by connecting Snap's pin 1 (Vpp) to the PIC's pin 4 (Vpp), pin 3 (GND) to pin 17 (Vss), pin 4 (PGD) to pin 19 (ICSPDAT), and

pin 5 (PGC) to pin 18 (ICSPCLK). Connect V+ from Analog Discovery to PIC pin 1 and ground to PIC pin 17.

When interfacing the stepper motor, ULN2803A transistor array and "power brick" , refer to Figure 1.6.

All of the software setup, such as configuration bits, is included in our appendix's code examples.

Experimental Procedure

The first task in this experiment is to write a simple program to flip an IO pin on our new microcontroller. _. Figure 1.3 shows the code, including the configuration bits, to flip a bit in z while loop. The configuration bits will stay the same moving forward.

Now is the time to set up the Timer1, CCP1, and the interrupts with the lines that follow the OSCCON assignment in Figure 1.4. Refer to the PIC16F1769 Data Sheet for more information on these register assignments.

After the setup, the next task is to create a software shift register. Choose 4 GPIO pins on PIC. In the code for the interrupt handler, our code in Figure 1.4 makes a single GPIO a "1" each time the interrupt is called. Each signal should be 25% of the duty cycle wave and shifted 90° from the last.

Add inputs for direction, speed, and on/off. The code in Figure 1.5 shows how to accomplish this. Input ports A2, B4, B6, B5 are added to control direction, pause, button1, and button2 variables respectively. The button1 and button2 pins are used to make up a 2 bit number to represent 4 different speeds. The speed is controlled by changing the max timer count value with the CCP1 register.

It is at this point that we will interface the stepper motor in our circuit. In our circuit we use a ULN2803A transistor array and "power brick" interfaced as shown in Figure 1.6.

Finally, program the stepper motor for position control. We used variables tracker, howFar, ninetyDeg, and distance to move our motor at multiples of 90°. We no longer needed the speed control, so we repurposed the hardware to control how far our stepper motor moves (i.e. 90°, 1800°, etc.). The final code is shown in Figure 1.8.

Results

The result of Experiment 5 should be a working stepper motor, with position control and several control inputs. This experiment teaches the basics of using a PIC16F1769 microcontroller, configuring the CCP unit and interrupts on the PIC, servo motor hardware/wiring and programming, and transistor driver circuits.

Experiment 6 DC Motor

Introduction

In experiment 6, the goal is to introduce the PIC microcontroller, DC motors and transistor drivers with analog control. We need to change the ADC unit into 4 bit StaticIO to cause it to work as an analog driven binary counter first, then use the same ramp signal to the ADC and change the ADC to PWM to get the square wave. Then, we need to observe the output of the PWM filter. Next, we need to change ADC to DAC to observe the frequency of the sine wave for DAC, and use input capture to get the square waves based on different DAC output. Finally, we need to control the speed of the DC motor and measure the RPM.

Materials

- PIC16F1769
- Snap Programmer

- Analog Discovery 2
- DC Motor
- Wires
- 5.1k Ohm Resistor
- 2N7000 transistor

Experimental Setup

Wire the Snap programmer to the PIC microcontroller by connecting Snap's pin 1 (Vpp) to the PIC's pin 4 (Vpp), pin 3 (GND) to pin 17 (Vss), pin 4 (PGD) to pin 19 (ICSPDAT), and pin 5 (PGC) to pin 18 (ICSPCLK). Connect V+ from Analog Discovery to PIC pin 1 and ground to PIC pin 17.

Also, wire the ADC input to pin3, the PWM input to pin2, and the PWM output to pin1, then use the 5.1k Ohm resistor to connect the ground and RC4, refer to Figure 3.1.

Then, we need to remove the resistor, and add the 2N7000 transistor for step 6 and 7. And we need to connect the DC motor to the ground, power, pin1 and the 2N7000 transistor, refer to Figure 3.2.

Experimental Procedure

First, we change the ADC to 4-bit StaticIO. We apply the ramp voltage that produced by the Analog Discovery Wavegen to the ADC pin to create the binary counter. The code of part 1 refers to Figure 3.3, and the graph result refers to Figure 3.4. Second, we change the ADC to PWM, and control the registers of the microcontroller. We control the PWM frequency between 450 HZ to 500HZ, use the same ramp signal to the ADC and change the ADC to PWM to get the square wave. The code of part 2 refers to Figure 3.5, and the square wave result refers to Figure 3.6. Third, we need to observe the output of the PWM filter. We design a RC filter and set the

cutoff frequency to 10HZ, capacitance to 3.3e-6, and use 5.1k Ohm resistance. After wiring the machine, we observe different output of the PWM filter by changing the frequency and the ramp to a sine wave. The wave result refers to Figure 3.7 for part 3.

Next, we need to change the ADC to DAC and repeat step 3 again. We remove the filter from the last step and configure the Op Amp. When increasing the frequency to around 500 Hz (period of 2ms) in this step, we started to see some boxyness in our output wave. We saw about 14 samples /period. 0.007 seconds for every sample. The code of part 4 refers to Figure 3.8, and the wave result refers to Figure 3.9. For step 5, we need to use input capture to get the square waves based on different DAC output. For 4MHz, Fosc/4, 1:1 prescaler, 5kHz square wave and scale the output voltage to the frequency. So we set DAC1REF = frequency and DACLDbits.DAC1LD = 1 to enable the DAC to convert a digital number to an analog number. The code of part 5 refers to Figure 3.10, and the wave result refers to Figure 3.11.

For step 6, we set up the 2N7000 transistor and DC motor and use the same code from step2 to control the speed of the DC motor. The wave result refers to Figure 3.12. Finally, we need to control the motor speed on the DAC. We connect the rotary encoder to the pin that represents frequency, and use the DAC voltage to drive the tachometer. Therefore, we can control the motor speed and measure the RPM. The code of part 7 refers to Figure 3.13, and the wave result refers to Figure 3.14.

Results

The result of experiment 6 is to control the DC motors and transistor drivers with analog control based on the PIC16F1769 microcontroller. The experiment helps us to understand how to change ADC to different analogs, for example, 4-bit StaticIO, PWM and DAC. And we can

measure different square waves based on different analogs. Also, the experiment helps us to know how to control the speed of a DC motor.

Experiment 7 Servo Motor

Introduction

In experiment 7 we are looking to use a new type of motor, the servo motor. The servo motor is controlled by PWM signals with different pulse width to control the movement of the machine. Our goal is to implement an ADC signal and transfer that signal to a PWM and then run it with the PWM signal. After we test that the PWM signal can control the machine, we will switch the machine that makes the signal. We will switch from the Analog Discovery to the PIC microcontroller. Then with this switch, we will need to control the motor and display a full range of motion to finish out the experiment.

Materials

- PIC16F1769
- Snap Programmer
- Analog Discovery 2
- Servo Motor
- Wires
- Potentiometer

Experimental Setup

Wire the Snap programmer to the PIC microcontroller by connecting Snap's pin 1 (Vpp) to the PIC's pin 4 (Vpp), pin 3 (GND) to pin 17 (Vss), pin 4 (PGD) to pin 19 (ICSPDAT), and pin 5 (PGC) to pin 18 (ICSPCLK). Connect V+ from Analog Discovery to PIC pin 1 and ground to PIC pin 17.

Now that the snap programmer and the PIC microcontroller is connected you need to select an ADC input pin (RC3) and connect that pin to the PWM filter pin (RC2) you select to create a duty cycle.

You also need to connect the servo motor into your board. There are 3 connections to install the servo motor. These are color coded so red is connected to 5V power, black is connected to ground and , white is connected to your PWM duty cycle output. The output is just another pin in which you decide to send the output of the PWM wave (RC1).

Lastly we set up the potentiometer, for this we need to connect this to our board. To connect it we need to connect the leftmost and rightmost pins to power a ground. It does not matter which side goes into power or ground as long as one does. Then the middle pin will go to the ADC input to control the wave cycle sent to the PWM filter so we can control the servo motor. Figure 1.8 will help you visualize the connections.

Experimental Procedure

This is a three part experiment, to accomplish our first goal we need to code our ADC input and transfer that input signal to the PWM input, to create a PWM duty cycle that we can use to control our servo motor. To activate our ADC input and our PWM duty cycle we need to code it then wire it. To code it figure 1.9, and to wire it look at the experimental setup. After everything is wired and coded, show the scope of your analog discovery to the instructor for a signature. It should look like figure 2.0. The next step is to control the servo motor with patterns in analog discovery. For this we keep everything the same and just open patterns. We need to set up patterns, figure 2.1 for reference. so we just open it and set the duty cycle to 5% to 15%. When that is set we can control the servo motor with the slider, moving left to right will control the motor in real life.

The last step is now switching the controls to the PIC microcontroller. We keep everything the same except we now add a potentiometer to our board, to add a potentiometer refer to experiment setup. Now with everything setup connect the potentiometer to the ADC input so we can control the duty cycle produced. We should get something that looks like figure 2.2.

Results

The end result of finishing the lab is a fully controllable servo motor, which is moved by the potentiometer. The experiment helps reinforce how to use a PIC16F1769 microcontroller, which allows you to control and program vast amounts of different hardware. It also explains the setup and the programming of a servo motor. These labs help us deepen our understanding of motors and the code behind how to control and use them.

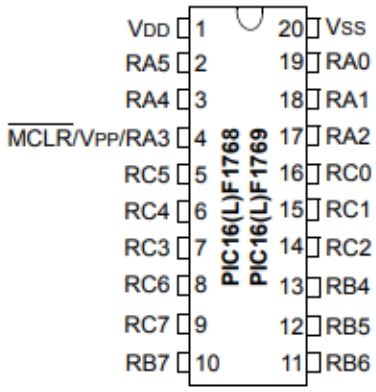
Conclusion

Finishing these experiments has taught us about many different motors and the programs within the microcontroller to control these motors. We learned about the stepper motor, which used interrupts, timer, and the Oscon to control and spin the stepper motor. The DC motor is another motor we learned about in which we used an ADC signal into a PWM signal, where then we connected that signal to the DAC so we could control the DC motor. Finally we have the servo motor, where we used the same signals in experiment 6 to control and the servo motor. These Experiments have helped us understand more about the different kinds of hardware and programs necessary to use them.

Appendix

Figure 1.1:

FIGURE 3: 20-PIN PDIP, SOIC, SSOP



Note: See Table 4 for location of all peripheral functions.

Figure 1.2:

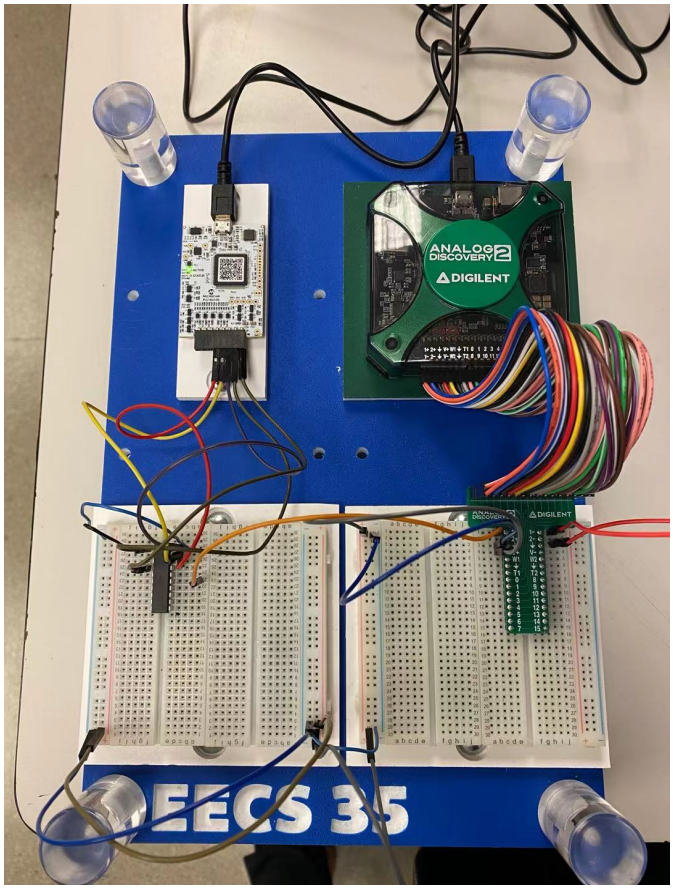


Figure 1.3:

```
/*
* File:   newmain.c
* Author: phu103
*
* Created on October 6, 2021, 5:18 PM
*/

// CONFIG1
#pragma config FOSC = INTOSC        // Oscillator Selection Bits (INTOSC oscillator: I/O function on CLKIN pin)
#pragma config WDTE = OFF           // Watchdog Timer Enable (WDT disabled)
#pragma config PWRTE = OFF          // Power-up Timer Enable (PWRT disabled)
#pragma config MCLRE = ON           // MCLR Pin Function Select (MCLR/VPP pin function is MCLR)
#pragma config CP = OFF             // Flash Program Memory Code Protection (Program memory code protection is disabled)
#pragma config BOREN = ON           // Brown-out Reset Enable (Brown-out Reset enabled)
#pragma config CLKOUTEN = OFF   // Clock Out Enable (CLKOUT function is disabled. I/O or oscillator function on the CLKOUT pin)
#pragma config IESO = ON            // Internal/External Switchover Mode (Internal/External Switchover Mode is enabled)
#pragma config FCMEN = ON           // Fail-Safe Clock Monitor Enable (Fail-Safe Clock Monitor is enabled)

// CONFIG2
#pragma config WRT = OFF            // Flash Memory Self-Write Protection (Write protection off)
```

```
#pragma config PPS1WAY = ON        // Peripheral Pin Select one-way control (The PPSLOCK bit cannot be cleared once it is set by software)
#pragma config ZCD = OFF           // Zero-cross detect disable (Zero-cross detect circuit is disabled at POR)
#pragma config PLLEN = ON          // Phase Lock Loop enable (4x PLL is always enabled)
#pragma config STVREN = ON         // Stack Overflow/Underflow Reset Enable (Stack Overflow or Underflow will cause a Reset)
#pragma config BORV = LO           // Brown-out Reset Voltage Selection (Brown-out Reset Voltage (Vbor), low trip point selected.)
#pragma config LPBOR = OFF         // Low-Power Brown Out Reset (Low-Power BOR is disabled)
#pragma config LVP = ON            // Low-Voltage Programming Enable (Low-voltage programming enabled)

// #pragma config statements should precede project file includes.
// Use project enums instead of #define for ON and OFF.

#include <xc.h>
/*
*
*/
void main (void){

        //Make C1 an output port
        TRISCbits.TRISC1 = 0;

        //KEEP THIS LINE!!!!!
        LATC = 0b00000010; //Starting output for our Cx output ports
        //KEEP THIS LINE!!!!!

        //Changing the FOSC clock frequency
        OSCCON = 0b00111000; //500khz
        T1CON = 0b00100001;

        while(1)
        {
        PORTCbits.RB1 = 0;
        PORTCbits.RB1 = 1;
        }

        return;
}
```

Figure 1.4:

```
void __interrupt() ISR(void)
{
        if (PIR1bits.CCP1IF == 1)
        {
        //Going 1 -> 2 -> 3 -> 4
        if((LATC & 0b00011110) == 0b00000010) //If RC1
        {
        LATC = 0b00000100; //Make RC2
```

```
		}
		else if((LATC & 0b00011110) == 0b00000100) //If RC2
		{
		LATC = 0b00001000; //Make RC3
		}
		else if((LATC & 0b00011110) == 0b00001000) //If RC3
		{
		LATC = 0b00010000; //Make RC4
		}
		else if((LATC & 0b00011110) == 0b00010000) //If RC4
		{
		LATC = 0b00000010; //Make RC1
		}

		PIR1bits.CCP1IF = 0;
		}
}


void main (void){

		//Make these 4 ports output ports
		TRISCbits.TRISC1 = 0;
		TRISCbits.TRISC2 = 0;
		TRISCbits.TRISC3 = 0;
		TRISCbits.TRISC4 = 0;

		//KEEP THIS LINE!!!!!
		LATC = 0b00000010; //Starting output for our Cx output ports
		//KEEP THIS LINE!!!!!

		//Changing the FOSC clock frequency
		OSCCON = 0b00111000; //500khz
		T1CON = 0b00100001;
		CCP1CON = 0b10100001;
		CCPR1 = 77.5; //77.5 Lengthen or shorten our period [Default speed 00, or 2.5ms pulse.)
		PIE1bits.CCP1IE = 1; //Enabling the CCP1 interrupt
		PIR1bits.CCP1IF = 0; //Clearing the CCP1 interrupt flag
		//The two bits  required to make the CCP1 interrupt work
		INTCONbits.PEIE = 1; //enables the peripheral interrupt
		INTCONbits.GIE = 1; //Globally enable interrupts

		while(1) {}

		return;
}
```

Figure 1.5:

```
int direction = 0; //0 is forwards, 1 is backwards
int pause = 0; //0 is on, 1 is off
int button1 = 0;
int button2 = 0;

void __interrupt() ISR(void)
{
if (PIR1bits.CCP1IF == 1)
                {
                 if(pause == 0)
                 {
          if(direction == 0)
                 {
            //Going 1 -> 2 -> 3 -> 4
            if((LATC & 0b00011110) == 0b00000010) //If RC1
            {
               LATC = 0b00000100; //Make RC2
            }
            else if((LATC & 0b00011110) == 0b00000100) //If RC2
            {
               LATC = 0b00001000; //Make RC3
            }
            else if((LATC & 0b00011110) == 0b00001000) //If RC3
            {
               LATC = 0b00010000; //Make RC4
            }
            else if((LATC & 0b00011110) == 0b00010000) //If RC4
            {
               LATC = 0b00000010; //Make RC1
            }
                 }

                  else if(direction == 1)
                 {
            //Going 4 -> 3 -> 2 -> 1
            if((LATC & 0b00011110) == 0b00010000) //If RC4
            {
               LATC = 0b00001000; //Make RC3
            }
            else if((LATC & 0b00011110) == 0b00001000) //If RC3
            {
               LATC = 0b00000100; //Make RC2
            }
            else if((LATC & 0b00011110) == 0b00000100) //If RC2
            {
               LATC = 0b00000010; //Make RC1
            }
            else if((LATC & 0b00011110) == 0b00000010) //If RC1
```

```
            {
               LATC = 0b00010000; //Make RC4
            }
                }
                }
                else if(pause == 1)
                {
        //Keep the high GPIO high and keep other three 3 low
        if((LATC & 0b00011110) == 0b00000010) //If RC1
                {
            LATC = 0b00000010; //Keep RC1
                }
                else if((LATC & 0b00011110) == 0b00000100) //If RC2
                {
            LATC = 0b00000100; //Keep RC2
                }
                else if((LATC & 0b00011110) == 0b00001000) //If RC3
                {
            LATC = 0b00001000; //Keep RC3
                }
                else if((LATC & 0b00011110) == 0b00010000) //If RC4
                {
            LATC = 0b00010000; //Keep RC4
                }
                }

      PIR1bits.CCP1IF = 0;
                }
}

void main (void){

                //Make these 4 ports output ports
                TRISCbits.TRISC1 = 0;
                TRISCbits.TRISC2 = 0;
                TRISCbits.TRISC3 = 0;
                TRISCbits.TRISC4 = 0;
                //Add switch as an input port
                TRISAbits.TRISA2 = 1;
                ANSELAbits.ANSA2 = 0; //Sets as a digital pin
                //Add a stop switch and 2 switches for cardinal position
                TRISBbits.TRISB4 = 1;
                TRISBbits.TRISB5 = 1;
                TRISBbits.TRISB6 = 1;
                ANSELBbits.ANSB4 = 0;
                ANSELBbits.ANSB5 = 0;
                ANSELBbits.ANSB6 = 0;

                //KEEP THIS LINE!!!!!
                LATC = 0b00000010; //Starting output for our Cx output ports
```

```
//KEEP THIS LINE!!!!!

//Changing the FOSC clock frequency
OSCCON = 0b00111000; //500khz
T1CON = 0b00100001;
CCP1CON = 0b10100001;
CCPR1 = 77.5; //77.5 Lengthen or shorten our period [Default speed 00, or 2.5ms pulse.)
PIE1bits.CCP1IE = 1; //Enabling the CCP1 interrupt
PIR1bits.CCP1IF = 0; //Clearing the CCP1 interrupt flag
//The two bits  required to make the CCP1 interrupt work
INTCONbits.PEIE = 1; //enables the peripheral interrupt
INTCONbits.GIE = 1; //Globally enable interrupts

while(1)
{
button1 = PORTBbits.RB6;
button2 = PORTBbits.RB5;
pause = PORTBbits.RB4;
//Will change the speed of our code
if ((button1 == 0) && (button2 == 0)){
CCPR1 = 77.5;
}
else if ((button1 == 0) && (button2 == 1)){
CCPR1 = 155;
}
else if ((button1 == 1) && (button2 == 0)){
 CCPR1 = 310;
}
else if ((button1 == 1) && (button2 == 1)){
CCPR1 = 620;
}
//This will change the direction when the switch changes
if(PORTAbits.RA2 == 1)
{
direction = 1;
}
else if(PORTAbits.RA2 == 0)
{
direction = 0;
}
}

return;
}
```

Figure 1.6:

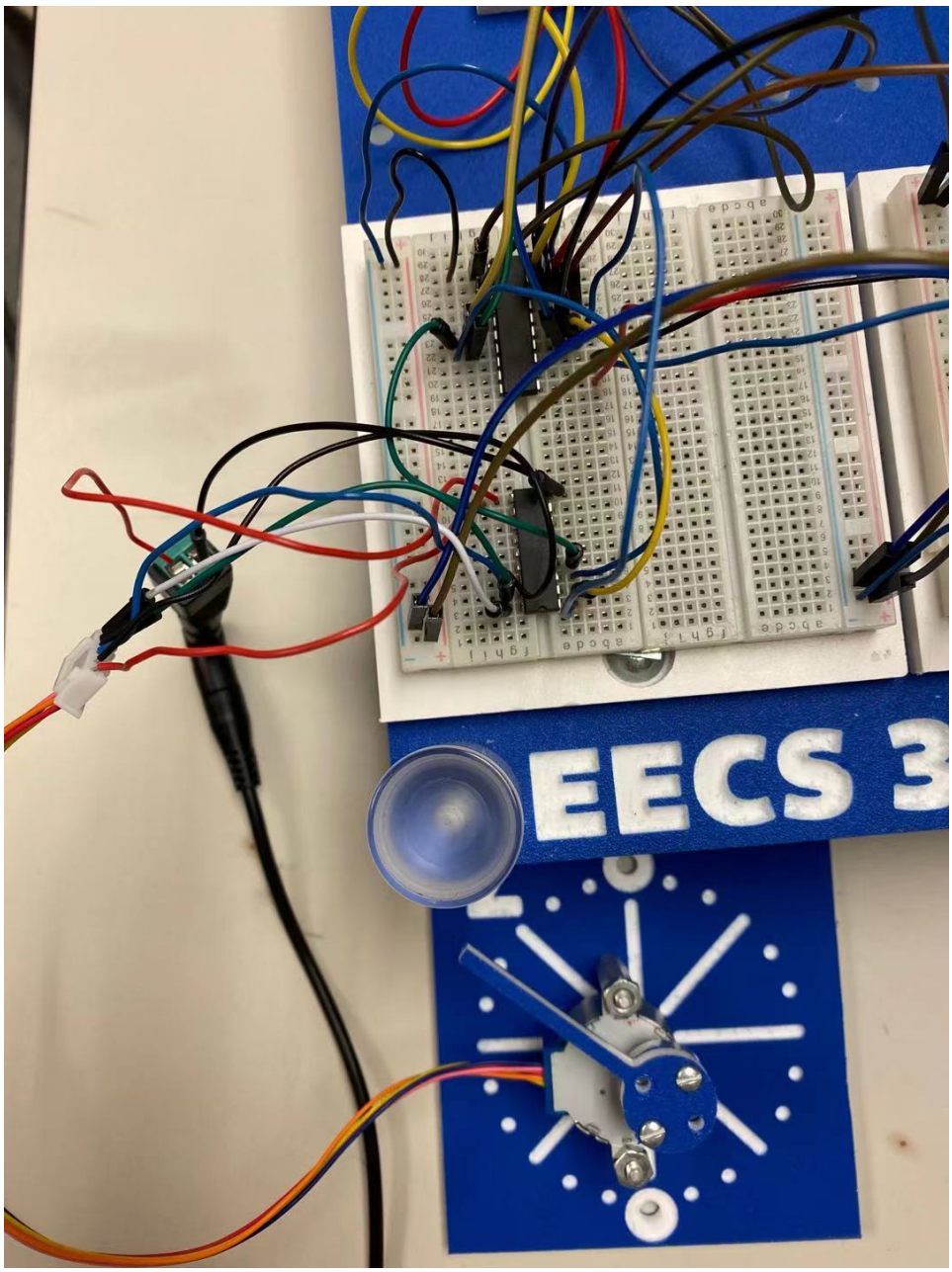


Figure 1.7:

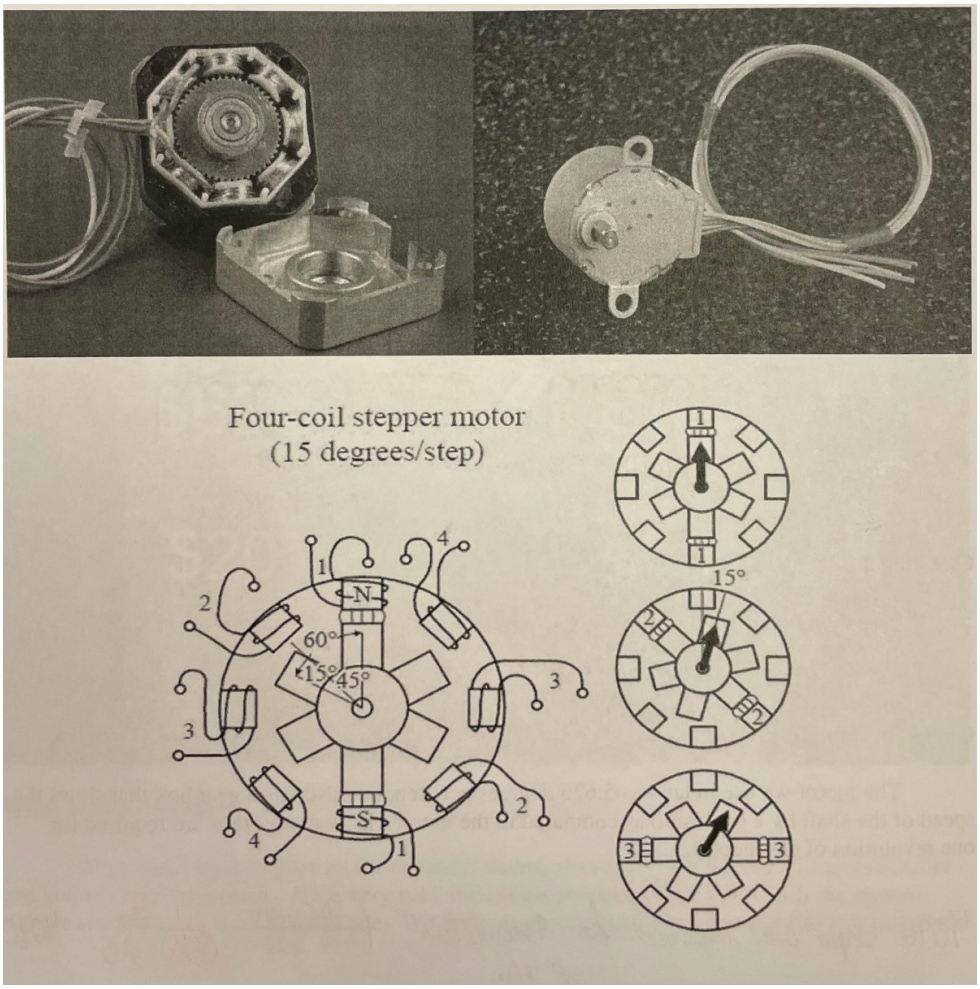


Figure 1.8

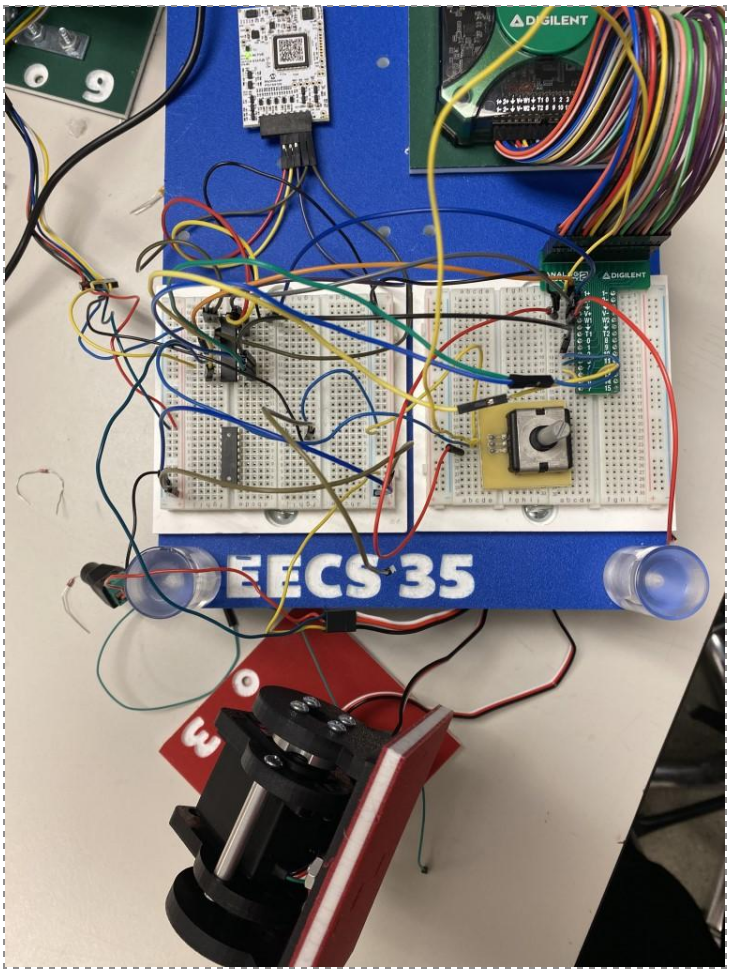


Figure 1.9:

```
void main (void){
	OSCCON = 0b00111000; //500khz
	T1CON = 0b00100001;
	TRISB = 0b00000000;
	ADCON0 = 0b00011101; //Makes pin RC3 an ADC unit
	ADCON1 = 0b11110000; //Right Justified

	T2CLKCON = 0b00000001;
	T2CON = 0b10000000;
	RC4PPS = 0b00001110;
	PWM3CON = 0b10000000;

	PWM3DC = 200;
	PWM3DCL = 128;

	TRISCbits.TRISC4 = 0;//PWM pin, making it an output
	TRISCbits.TRISC6 = 1;
	ANSELCbits.ANSC6 = 1;
	ADCON0bits.ADON = 1;

	while (1){
		ADCON0bits.GO = 1;
		while(ADCON0bits.GO == 1){ } //Nothing goes inside
here
		int variable = 0.1*ADRES + 51;
		PWM3DC = variable<<6;
	}
}
```

Figure 2.0:

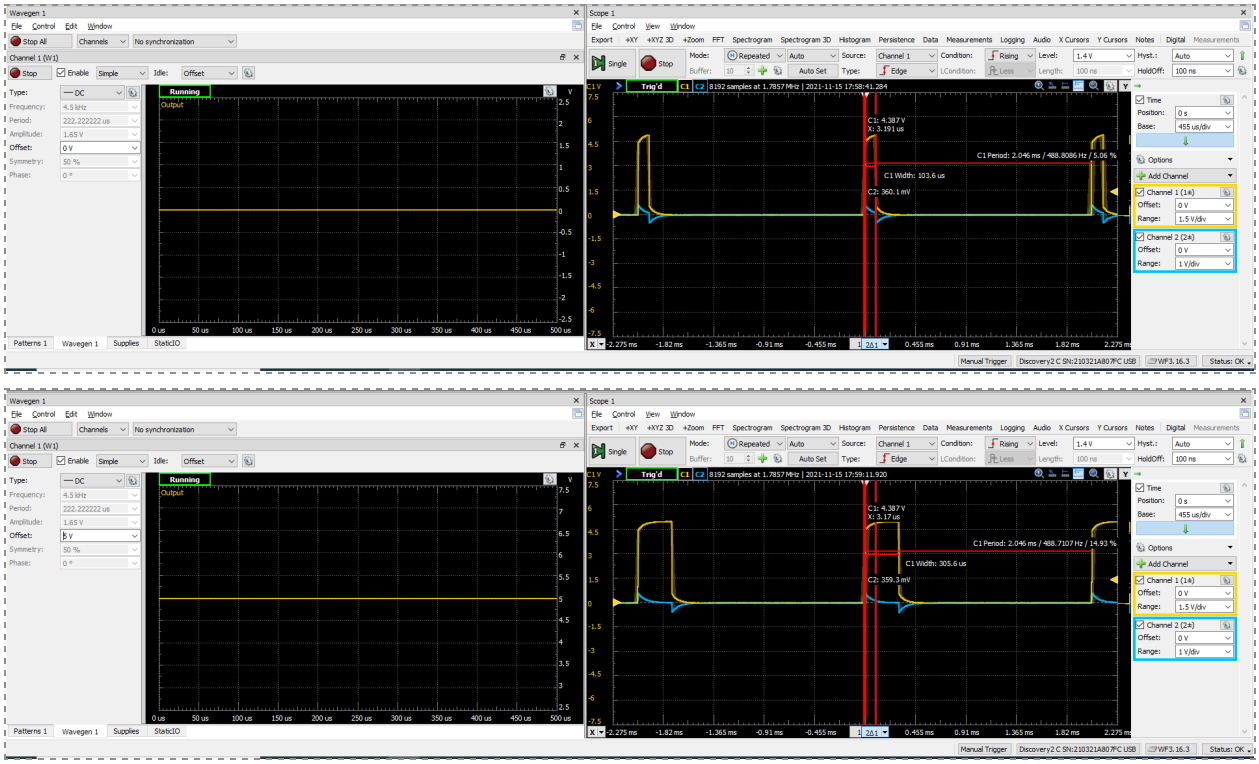


Figure 2.1:

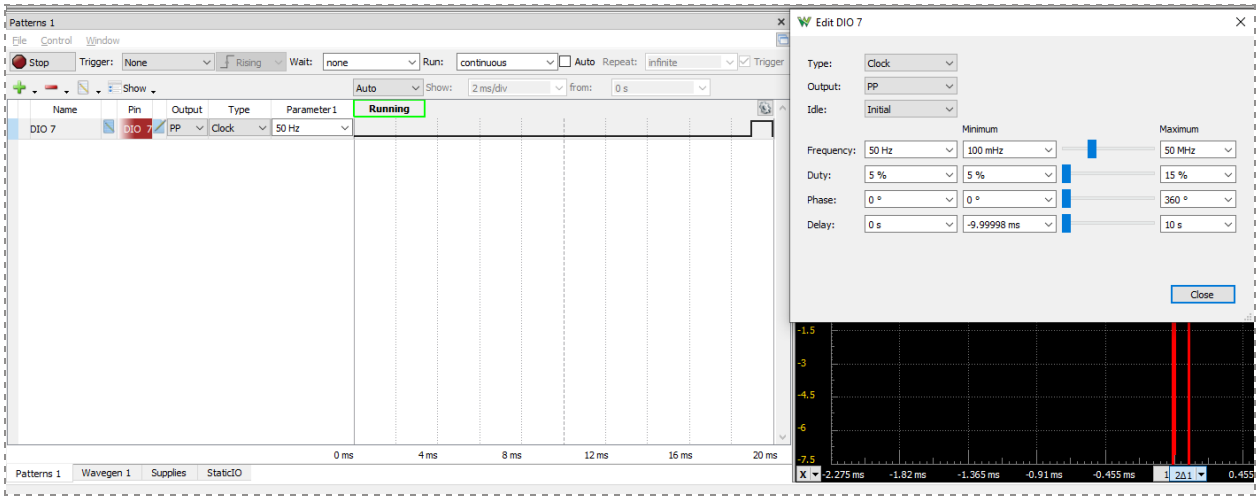


Figure 2.2:

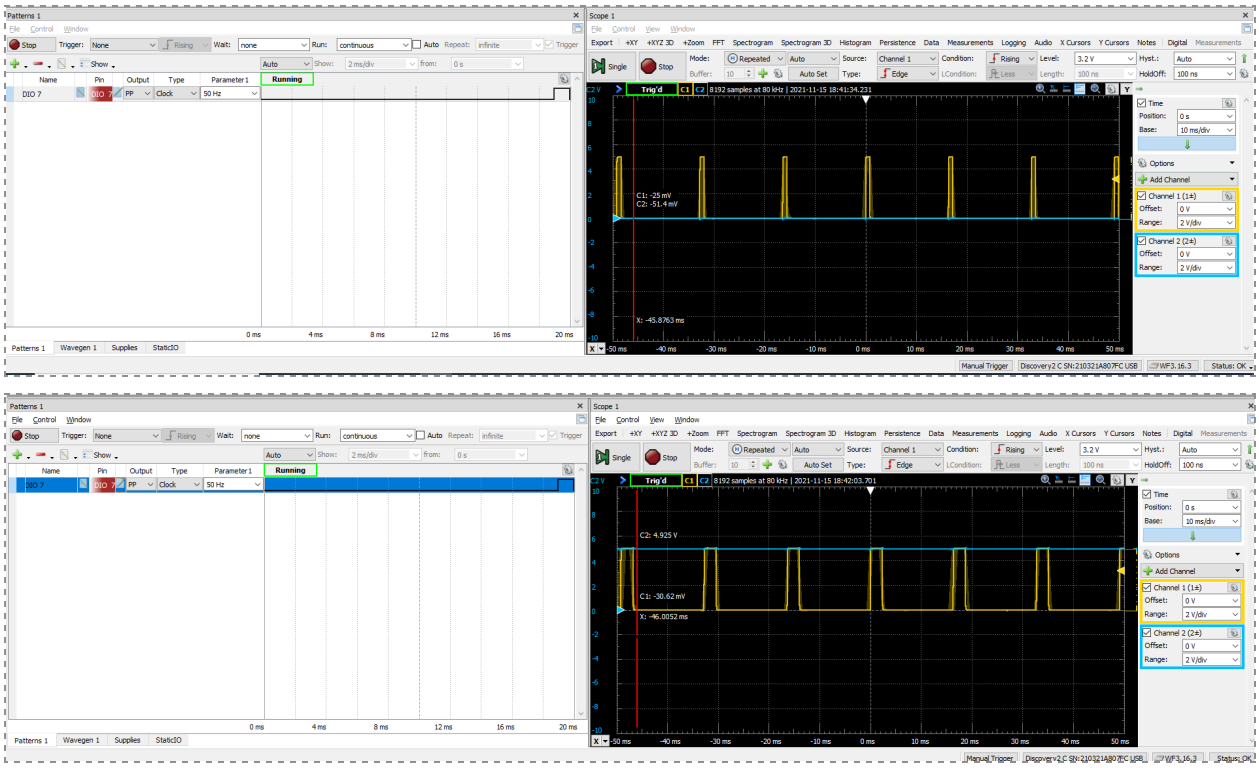

Patterns 1
Scope 1
Running
Trig'd
C1: -25 mV
C2: -51.4 mV
X: -45.8763 ms
C2: 4.925 V
C1: -30.62 mV
X: -46.0052 ms

Figure 3.1:

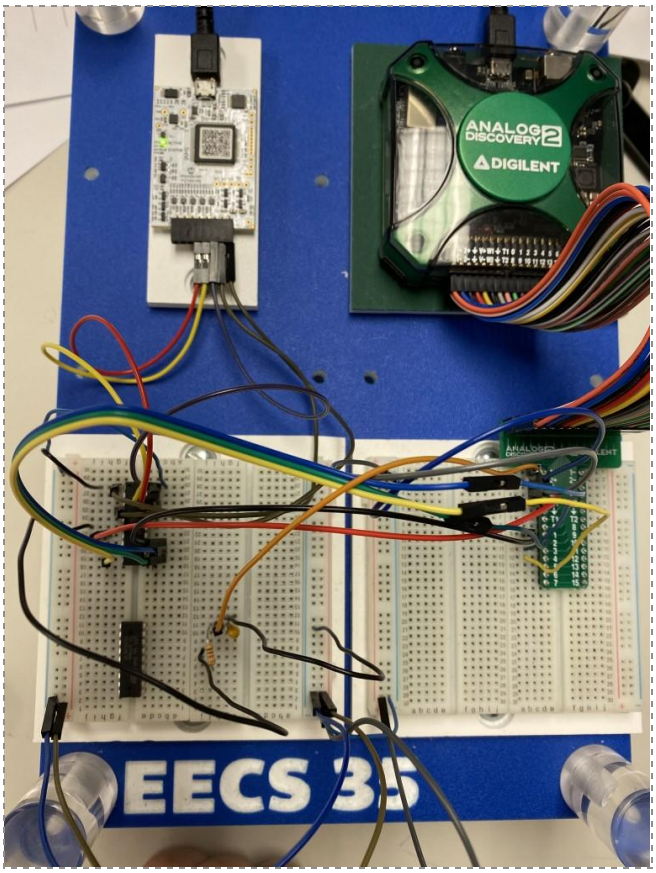


Figure 3.2:

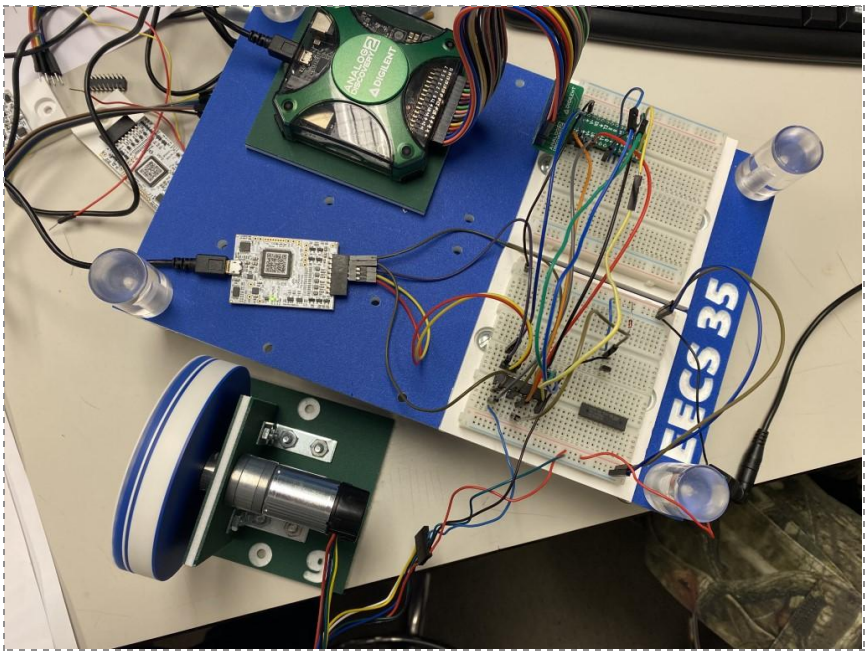
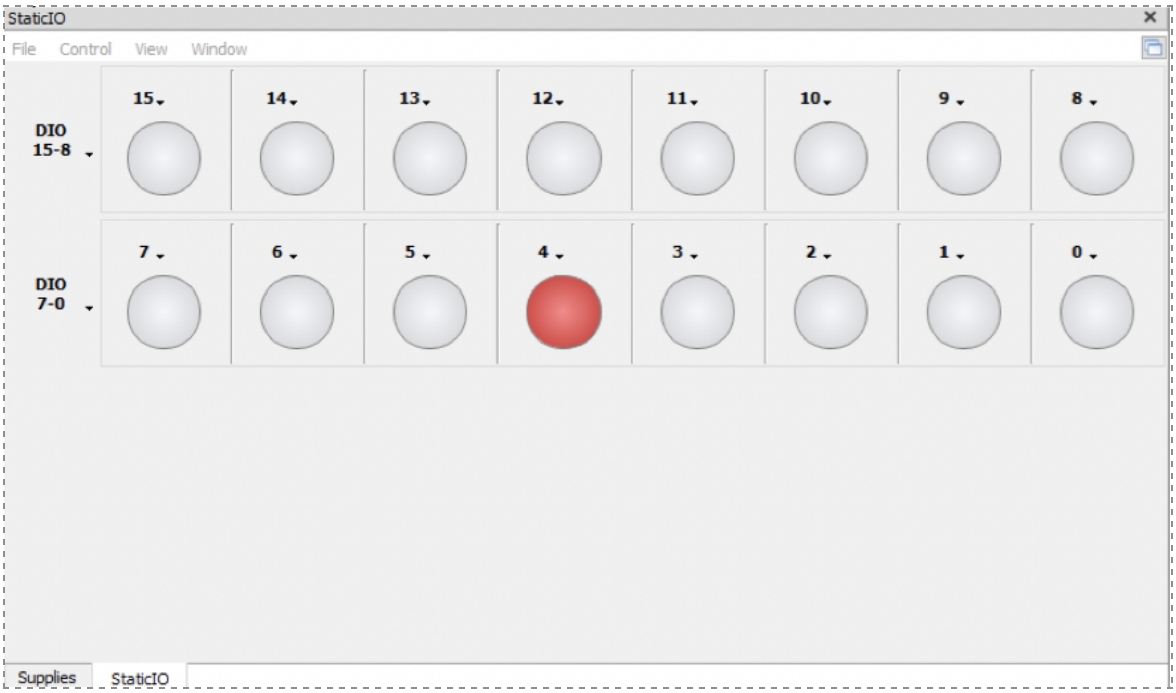


Figure 3.3:

```
//This is the code minus the configuration bits and the library statement
void main (void){

        //Changing the FOSC clock frequency
        OSCCON = 0b00111000; //500khz
        T1CON = 0b00100001;
        CCP1CON = 0b10100001;
        ADCON0 = 0b00011101; //Makes pin RC3 an ADC unit, Left Justified
        ADCON1 = 0b01110000;
        TRISBbits.TRISB4 = 0;
        TRISBbits.TRISB5 = 0;
        TRISBbits.TRISB6 = 0;
        TRISBbits.TRISB7 = 0;

        while (1){

        //To make conversion, set GO/DONE bit and wait for it to become cleared
        //The ADC hardware automatically clears the bit when the ADC conversion finishes
        ADCON0bits.GO = 1;
        while(ADCON0bits.GO == 1){
        //Nothing goes inside here
        }

        //The four most significant bits of ADRESH will be sent to PORTB
        PORTB = ADRESH;

        }
        return;
}
```

Figure 3.4:



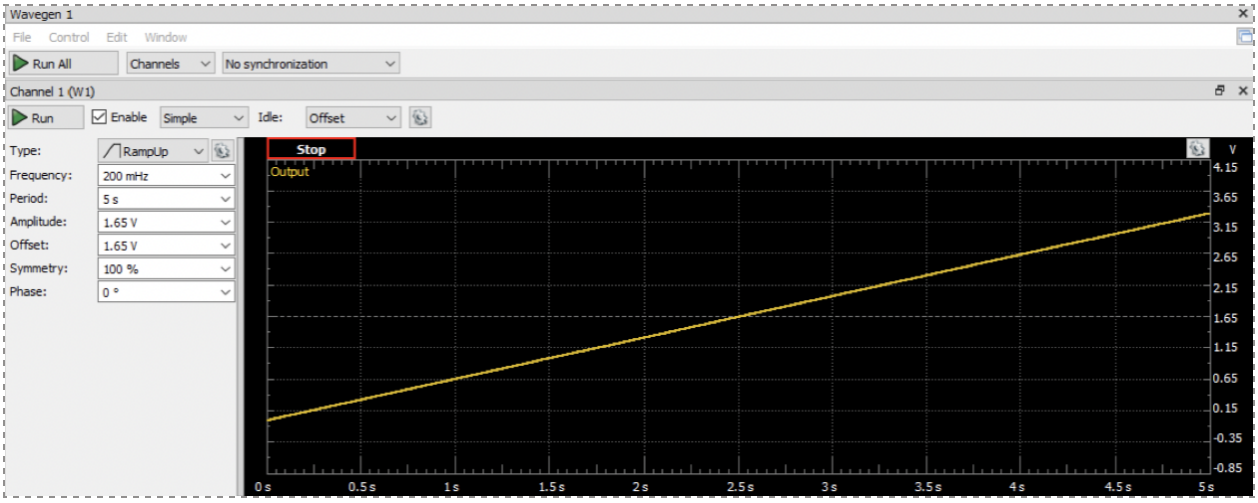


Figure 3.5:

```
void main (void){
//Changing the FOSC clock frequency
OSCCON = 0b00111000; //500khz
T1CON = 0b00100001;
CCP1CON = 0b10100001;
ADCON0 = 0b00011101; //Makes pin RC3 an ADC unit, Left Justified
ADCON1 = 0b01110000;

//Configuring PWM
PWM3CON = 0b10000000;
T2CLKCON = 0b00000001;
T2CON = 0b10000000;
RC4PPS = 0b00001110; //Use RC4 for PWM3

//Setting ports to outputs for the shift register
TRISBbits.TRISB4 = 0;
TRISBbits.TRISB5 = 0;
TRISBbits.TRISB6 = 0;
TRISBbits.TRISB7 = 0;

TRISCbits.TRISC4 = 0; //PWM pin, making it an output
//ANSELCbits.ANSC0 = 1;


while (1){

//To make conversion, set GO/DONE bit and wait for it to become cleared
//The ADC hardware automatically clears the bit when the ADC conversion finishes
ADCON0bits.GO = 1;
while(ADCON0bits.GO == 1){
//Nothing goes inside here
}

//The four most significant bits of ADRESH will be sent to PORTB
PORTB = ADRESH;

//Put ADRESH into the PWM
PWM3DCH = ADRESH;

}
```

```
        return;
}
```

Figure 3.6:

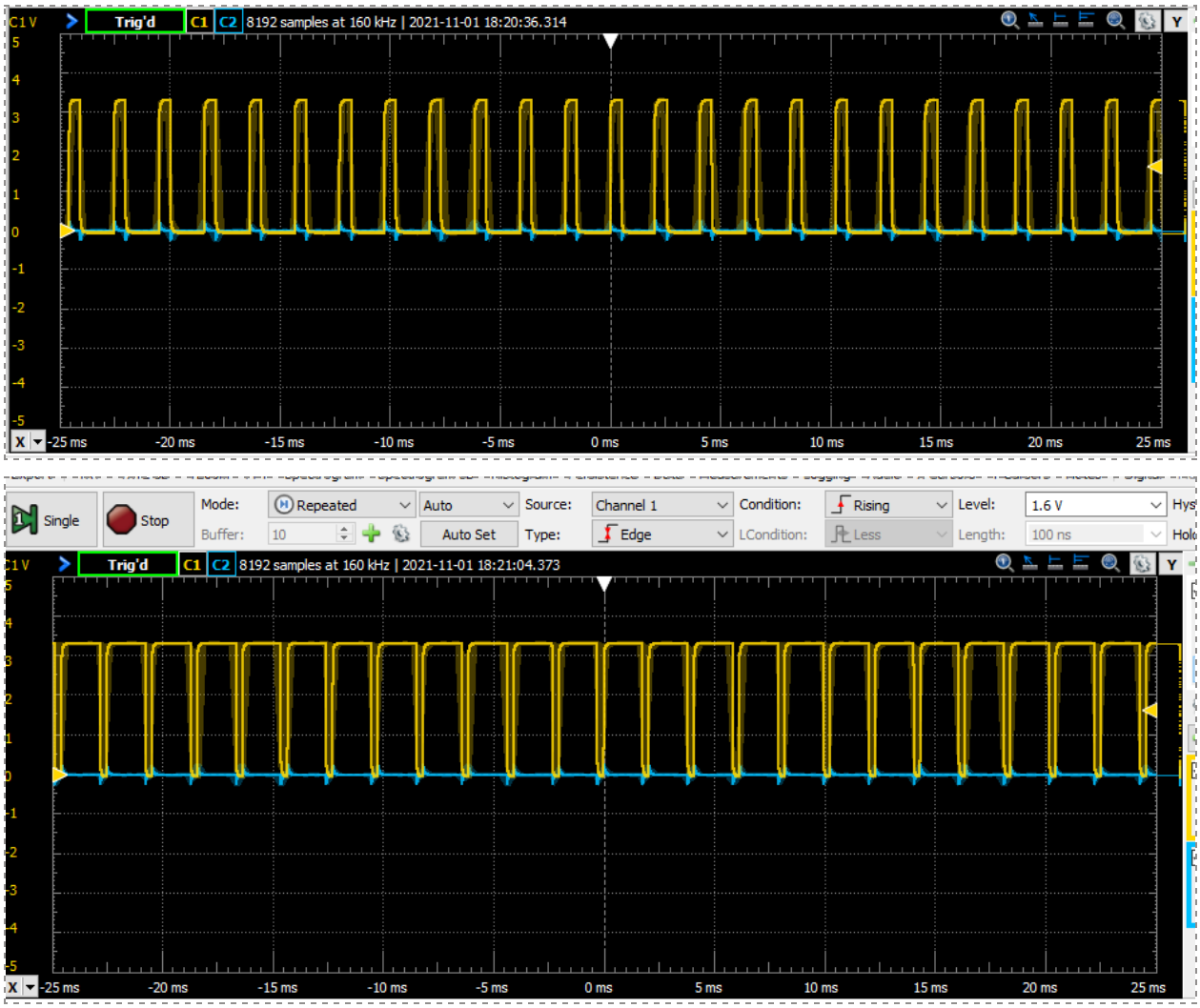


Figure 3.7:

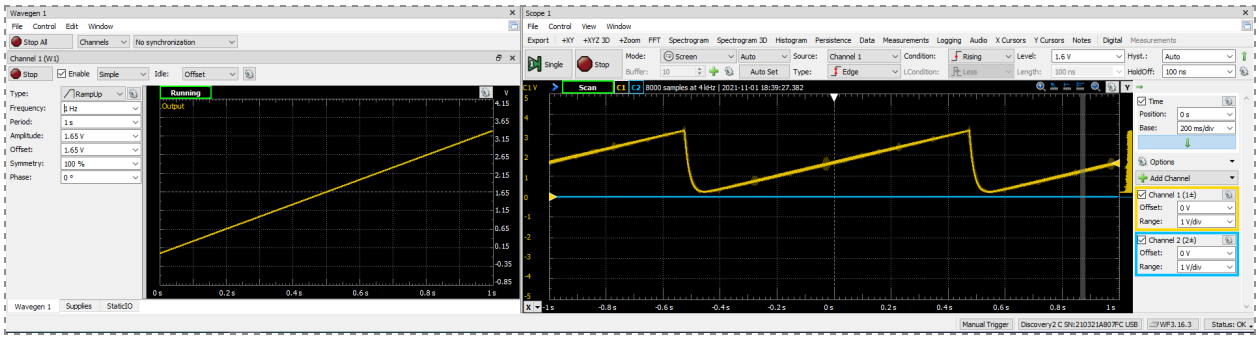


Figure 3.8:

```
//ADDED INTO OUR VOID MAIN TO CONFIGURE DAC

DAC1CON0 = 0b11000000; //This enables DAC, left justified.
        OPA1CON = 0b10010000;//configures unity gain and disables the override function
        OPA1PCHS = 0b00000010;//connect DAC1 output to the AMP non inverting input.

//ADDED INTO OUR WHILE LOOP

DAC1REF = ADRES;
        DACLD = 0b00000001;//enables the DAC to convert a digital number to analog number
```

Figure 3.9:

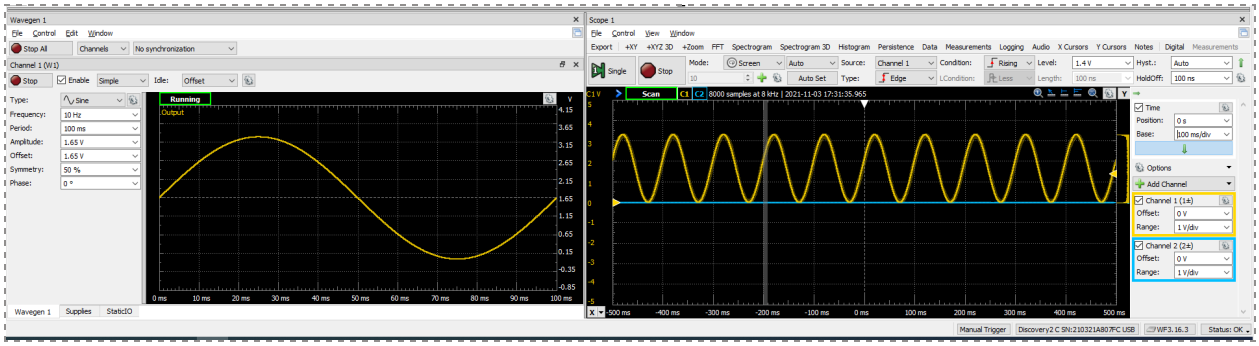


Figure 3.10

```
volatile unsigned int current_capture;
volatile unsigned int period;

void __interrupt() ISR(void) {
        if(PIR1bits.CCP1IF) {
        period = CCPR1 - current_capture;
      current_capture = CCPR1;
        PIR1bits.CCP1IF = 0;
        }
        else
        {
        PIR1bits.CCP1IF = 0;
        }

        //Test to see if interrupt is called
        //PORTBbits.RB5 = 0;
        //PORTBbits.RB5 = 1;

}

void main (void){

        //Changing the FOSC clock frequency
        OSCCON = 0b01101000; //4 MHz
        T1CON = 0b00000001;
        CCP1CON = 0b10000101; //Originally 0b10100001
        CCP1CAP = 0b00000000;
        ADCON0 = 0b00011100; // bit 6-2 is set for AN7              Makes pin RC3 an ADC unit
        ADCON0 = 0b01111010; // Set for DAC1_output    ADCON1 = 0b01000000;
        //ADCON1bits.ADFM = 1; //Right Justified

        //Configuring the DAC
        DAC1CON0 = 0b11000000; //This enables DAC, left justified.
        DACLD = 0b00000001;//enables the DAC to convert a digital number to analog number
        //Configuring Op Amp
        OPA1CON = 0b10010000;//configures unity gain and disables the override function
        OPA1PCHS = 0b00000010;//connect DAC1 output to the AMP non inverting input.

        TRISCbits.TRISC2 = 0; //PWM pin, making it an output
```

```
        //ANSELCbits.ANSC0 = 1;

        //Configuring a GPIO for Input Capture
        TRISCbits.TRISC3 = 1;
        ANSELCbits.ANSC3 = 0;
        CCP1PPS = 0b00010011; //Our wave going into RC3

        //Configuring the interrupt
        PIE1bits.CCP1IE = 1;
        PIR1bits.CCP1IF = 0; //Line of code to clear the flag
    INTCONbits.PEIE = 1;
    INTCONbits.GIE = 1;



        while (1){

        float frequency = 13.2e6 / period;

        //To make conversion, set GO/DONE bit and wait for it to become cleared
        //The ADC hardware automatically clears the bit when the ADC conversion finishes
        ADCON0bits.GO = 1;
        while(ADCON0bits.GO == 1){ } //Nothing goes inside here


        //PART 5
        //For 4MHz, Fosc/4, 1:1 prescaler, 5kHz square wave
        //Scale  the output voltage to the frequency. 5000Hz = 5V, 4000Hz = 4V, etc.
        DAC1REF = frequency;
        DACLDbits.DAC1LD = 1; //enables the DAC to convert a digital number to analog
number

        }

        return;
}
```

Figure 3.11

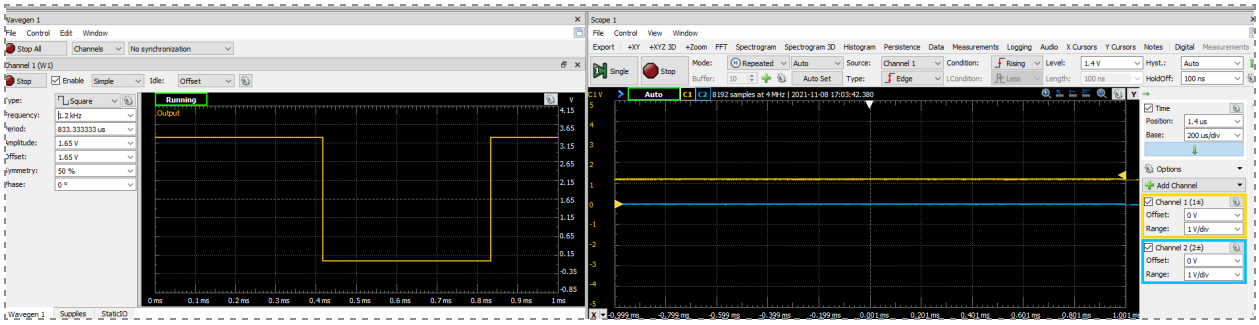


Figure 3.12

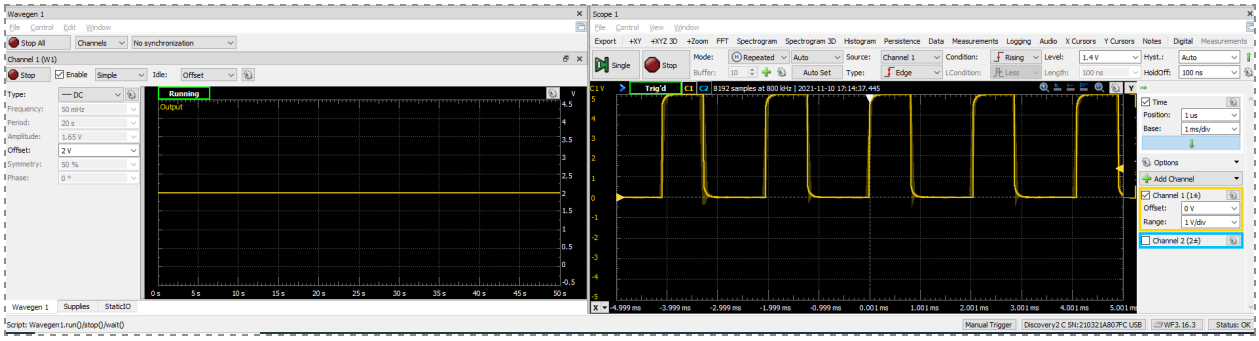


Figure 3.13

```
volatile unsigned int current_capture;
volatile unsigned int period;

void __interrupt() ISR(void) {
        if(PIR1bits.CCP1IF) {
```

```
        period = CCPR1 - current_capture;
      current_capture = CCPR1;
        // Test to see if the interrupt is called
        PORTBbits.RB5 = 0;
        PORTBbits.RB5 = 1;
        PIR1bits.CCP1IF = 0;
        }



}


void main (void){

        //Changing the FOSC clock frequency
        OSCCON = 0b00111000; //500khz
        T1CON = 0b00100001;
        CCP1CON = 0b10000101;
        ADCON0 = 0b00011101; //Makes pin RC3 an ADC unit, Left Justified
        ADCON1 = 0b01110000;

        //Configuring the DAC
        DAC1CON0 = 0b10000000; //This enables DAC, right justified.
        DACLD = 0b00000001;//enables the DAC to convert a digital number to analog number
        //Configuring Op Amp
        OPA1CON = 0b10010000;//configures unity gain and disables the override function
        OPA1PCHS = 0b00000010;//connect DAC1 output to the AMP non inverting input.

        //Configuring PWM
        PWM3CON = 0b10000000;
        T2CLKCON = 0b00000001;
        T2CON = 0b10000000;
        RC4PPS = 0b00001110; //Use RC4 for PWM3
        //Our wave from waveforms going into RC3

        //TRISCbits.TRISC2 = 0;
        TRISCbits.TRISC4 = 0;//PWM pin, making it an output
        //ANSELCbits.ANSC0 = 1;
```

```
        //Configuring a GPIO for Input Capture
        TRISCbits.TRISC6 = 1;
        ANSELCbits.ANSC6 = 0;
        CCP1PPS = 0b00010110; //Our square wave from the motor's encoder going into RC6

        //Configuring the interrupt
        PIE1bits.CCP1IE = 1;
        PIR1bits.CCP1IF = 0;//Line of code to clear the flag
    INTCONbits.PEIE = 1;
    INTCONbits.GIE = 1;

        //Setting ports to outputs for the shift register
        TRISBbits.TRISB4 = 0;
        TRISBbits.TRISB5 = 0;
        TRISBbits.TRISB6 = 0;
        TRISBbits.TRISB7 = 0;



        while (1){

        float frequency = 24000. / period;

        //To make conversion, set GO/DONE bit and wait for it to become cleared
        //The ADC hardware automatically clears the bit when the ADC conversion finishes
        ADCON0bits.GO = 1;
        while(ADCON0bits.GO == 1){ } //Nothing goes inside here

        PORTB = ADRESH;

        //PART 3
        //Put ADRESH into the PWM
        PWM3DCH = ADRESH;
        //PART 4
        //Put ADRESH into the DAC
        //DAC1REF = ADRES;
        //DACLD = 0b00000001;//enables the DAC to convert a digital number to analog
number

        //PART 5
```

```
        //For 4MHz, Fosc/4, 1:1 prescaler, 5kHz square wave
        //Scale  the output voltage to the frequency. 5000Hz = 5V, 4000Hz = 4V, etc.
        DAC1REF = frequency;

        DACLDbits.DAC1LD = 1; //enables the DAC to convert a digital number to analog
number

        }

        return;
}
```

Figure 3.14