DECEMBER 16, 2021

3-LEGGED STEWART GOUGH PLATFORM FOR LONG BONE FRACTURE REALIGNMENT APPLICATION DEMONSTRATION

FINAL PROJECT: INTRODUCTION TO EMBEDDED SYSTEMS, SECTION 3

NICHOLAS ZANON, KRUTIK SHAH
ROWAN UNIVERSITY
Introduction to Embedded Systems, Dr. Jannatun Naher

Table of Contents

Abstract

In this embedded design project, a 3-Legged Stewart-Gough platform was created for biomedical applications, particularly orthopedic surgeries to realign long bone fractures. A normal procedure is dangerous and has a high risk of failure, but an embedded system designed to realign the bone would make it easier and less risky for medical professionals. While this is a proof-of-concept prototype, this is a demonstration to show an embedded system can be used in place of traditional methods of surgery. This system includes several electromechanical parts as well as a custom PCB with a PIC18 microcontroller. We had some success in driving the motors with PWM, and quite a few shortcomings, with lots of future plans to look forward to for improvement on the design.

Introduction

There are many biomedical applications of embedded systems. One example is a smartwatch such as the Fit Bit, which utilizes heart rate sensors and blood-oxygen level sensors and can even compile this information to get accurate data to measure stress levels. For our project, we will design an embedded system to be used for orthopedic procedures.

Project Motivations

This project is motivated by Junior/Senior Engineering Clinic project from the Biomedical Engineering department, led by Dr. Abedin Nassab, called “Robossis”. This is a robot used in a biomedical application, designed to realign a long bone fracture.

In a normal procedure for long bone fracture realignment, there are concerns with mostly human error; there are medical professionals that will hold the upper leg in place while another medical professional will try to manually realign the femur and will apply pressure. This could be very dangerous and there is high risk of failure despite the expertise of medical professionals, because after all they are still human beings, and thus there is always some human error. This could also damage soft tissue surrounding the bone. “Robossis” makes it easier, safer, and less stressful for medical professionals when conducting this surgery because it is designed to eliminate that human error with the accuracy of a robot (Saeedi-Hosseiny, et al., 2021). The figure below is a picture of the actual “Robossis”.

Figure 1: Robossis (Abedin-Nassab, et al., 2020)

The video below is a simulation of how Robossis is used.

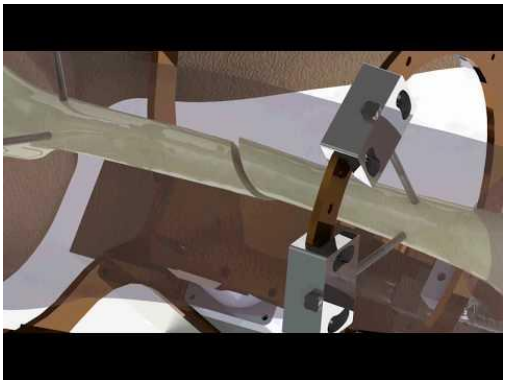

Figure 2: Robossis Simulation (Abedin-Nassab, et al., 2020)

For this project in particular, we are designing a smaller portion of “Robossis” with an embedded system. This is a proof of concept designed to demonstrate a similar product can be achieved through the various benefits of an embedded system. In the figure above, we are designing a simpler version of the moving plate.

The mechanical design and the movement is similar to a 3-legged Stewart-Gough platform (seen in the figure below), but controlled by an embedded system consisting of 3 potentiometers, 3 stepper motors, 3 limit switches, PIC18 microcontroller, and an LCD screen. Instead of the moving platform, there would be 3 screws that would hold the bone in place.

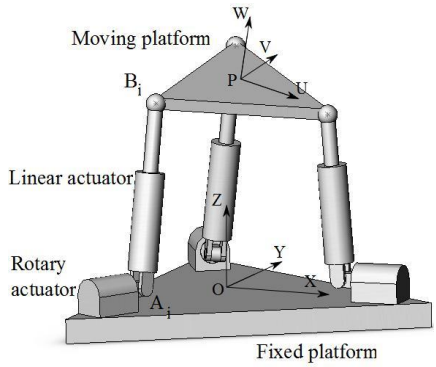

Figure 3: 3-Legged Stewart-Gough Platform (Kucuk, 2012)

Project Goals

1. Design a PCB
2. Wirelessly communicate with Bluetooth
3. Control stepper motors with PWM
4. Design electronics for a mechanical application (Electromechanical design)
5. Gain experience working with industry-standard equipment and hardware
6. Accomplish a more complex and challenging project for the course

Project Design Specifications

As mentioned earlier, we are utilizing 3 potentiometers, 3 limit switches, 3 stepper motors, an LCD screen and a PIC18 microcontroller (PIC18F47Q43). We also wanted to include a gyroscope when we started, but due to the parts shortage we could not find one that was cheap enough for our purposes; the cheapest we could find was $50.

The three potentiometers will be utilized to change direction of the PWM control, which will change the direction of the motors. Since they are analog inputs, they will require the ADC in the microcontroller. Then, the digital signal will be sent to the motor drivers, to indicate which direction to move the motors. The values sent from the potentiometer, after the conversion, range from 0 to 512. From 0 to 170 inclusive the motors would spin counter clockwise, from 171 to 340 inclusive the motors would not spin and the Enable Not (ENN) register was set to 1 disabling the motors, and from 341 to 512 inclusive the motors would spin clockwise.

The three limit switches are digital inputs and are utilized as interrupts. When the screw that goes into the femur reaches the limit switch (which is located near the base of the stepper motor) the interrupt is

activated and stops the motor from moving. This is essentially a safety measure to make sure the screw doesn't go below the mechanical limit.

The three stepper motors are driven by three TMC2226-SA motor drivers. In the microcontroller, they are controlled with PWM, with 4 kHz at 50% duty cycle, and sent to a DAC to drive the motors.

The LCD screen has 16-pins, which is connected to a serial backpack which brings it down to 4-pins connecting to the microcontroller, programming with serial data and clock. The LCD is designed to display the direction the motors are turning (or if they are not moving at all).

Methodology & Results

PCB Design

The first step to designing the PCB was to understand what all of our inputs and outputs were. We then followed the datasheet for PIC18 and TMC motor drivers to make proper connections. For power, we wanted to be able to power off of the wall outlet, but we were having trouble finding a 5V 3A power supply (5V for the microcontroller, and up to 3A for all three of our motors) so we had initially decided to go with a 5V 3A linear voltage regulator, but that never came in time due to the parts shortage, and we never got that onto the board. However, we were lucky enough to eventually find a 5V 2.6A power supply which was sufficient enough, so we ended up shorting the linear voltage regulator connection and removed the passive components that went with it on the PCB. We also have a switch that would turn on or off the power to the board. Additionally, based on the datasheet of our debugger Pickit 4, we included some passive components to the debugger circuit on our PCB. The full schematic and a 3D model of our board can be seen in the figures below.

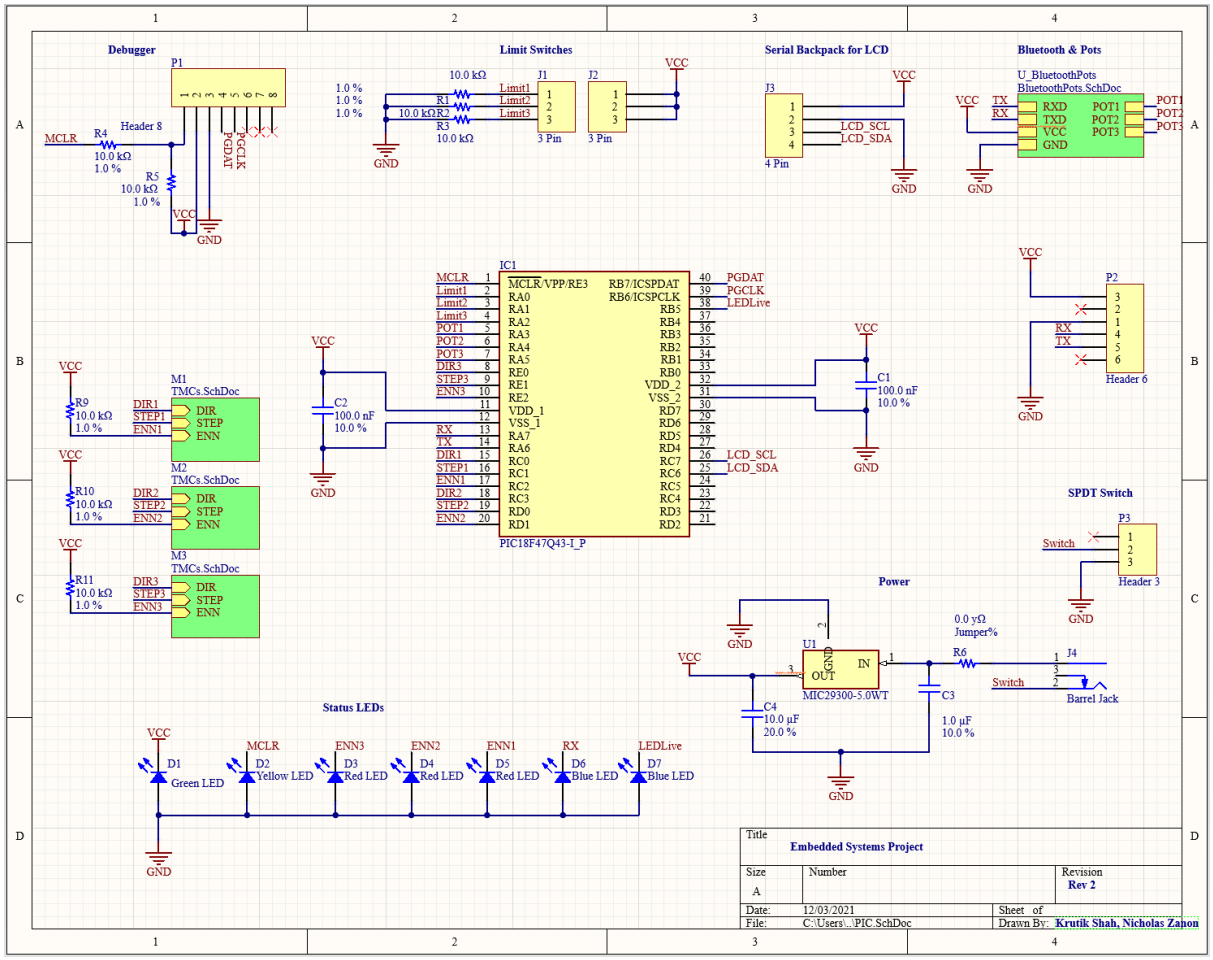

Figure 4: PCB Schematic

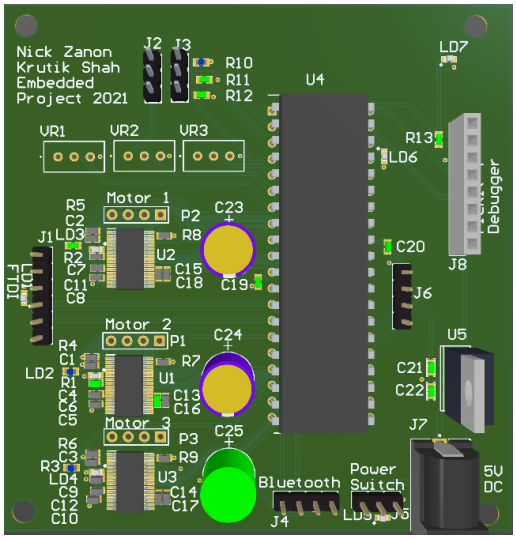

Figure 5: PCB 3D Model

Code

Serial Ports

The serial ports were used for UART communications with the HM10 Bluetooth breakout board. The SPEN, TXEN, and RXEN bits need to be set to 1 to enable Serial port communications, Transmit, and Receiving pins. Then the RCSTA register can be used for receiving data over the RX assigned pin. Unfortunately, the Bluetooth module did not transmit any data.

Interrupt

The INTCON registers were used to take an external interrupt signal from the limit switches. The INTEDG0 bit was set to 1 so the interrupt would occur on a rising edge. When the limit switches were pressed, a 5V signal would go into the pin causing a rising edge and the interrupt would stop the user's control, then set DIR to 1 and ENN to 0 causing the motor to rotate clockwise and raise the nut connector up off the limit switch, then the user had control of the motors once again.

Timer

The timers were primarily used in the PWM and the I2C. The FOSC is 8MHz, and the equation for the period of the timer is $\frac{1}{\frac{FOSC}{4}}$ or $\frac{1}{\frac{8MHz}{4}}$, so the period of the timer is 0.5 microseconds. No pre or post scalers were used.

Analog to Digital (ADC)

The Analog to Digital converter was used to take the user's input through the potentiometers into the PIC by scaling the input to a value from 0 to 512. The ADCON0 was set to reference the PIC's VDD pin as the voltage level for the input. The ADCS (clock select) bits were set to select the FOSC/4 option. The ADON bit was set to 1 to enable the ADC option, and was kept at 1 to ensure that the ADC never stopped taking in data unless an interrupt occurred.

Mechanical Design

The mechanical design consists of several 3D printed parts, which includes the tracks, the ball joints, and the slides that allow the movement of the ball joints up and down the lead screw. The tracks were created to prevent the rotation of the ball joint around the lead screw and made sure it stood in place while the slides went up and down the track. There are two acrylic bases, with our names engraved onto them from a laser cutter. The stepper motors lie in between the two bases. We used three skewers in place of the medical-grade screws that would go into the bone (because those screws are expensive) and we used a piece of foam to represent the bone.

Results

The final product can be seen in the figure below.

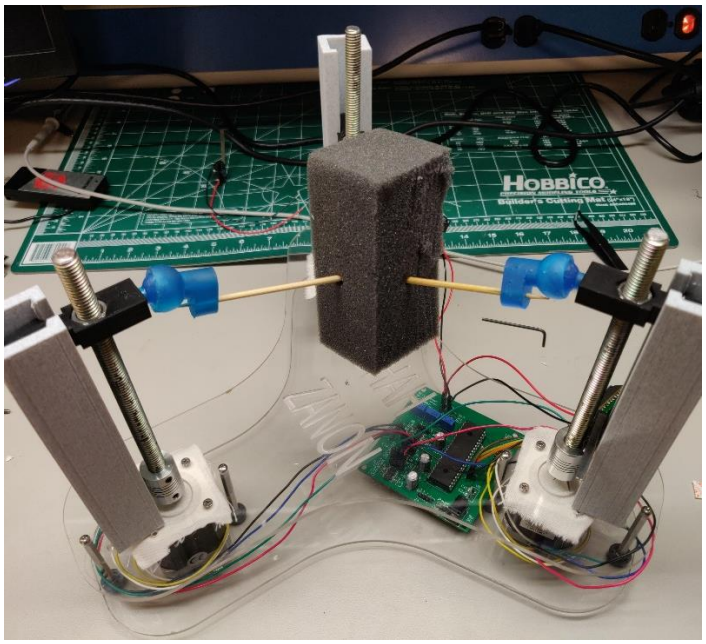

Figure 6: Final Product

We were able to get the motors to spin and change direction with our PWM code. However, there was one issue with the TMC motor drivers: the motor drivers were outputting the 2 PWM signals in phase with each other, causing essentially a “rotor lock” in the motors. After removing the drivers from the breadboard however the problem did not occur again.

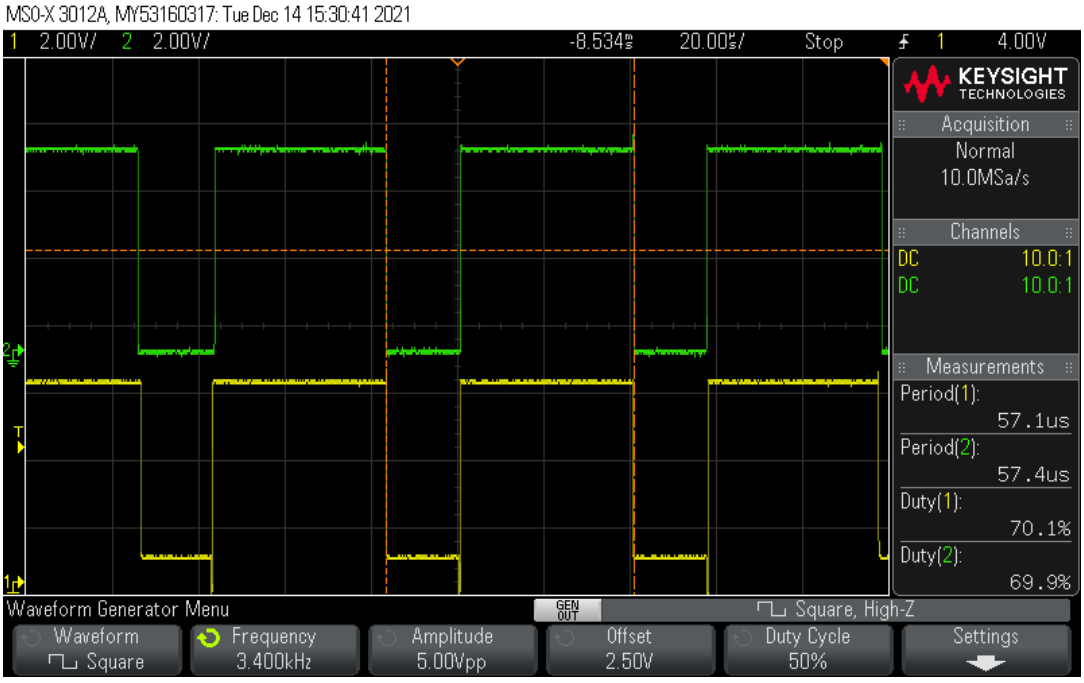

Figure 7: TMC Driver Output

The potentiometers would not work as expected. The potentiometers that were installed on the PCB were finicky and changing the position on the potentiometer would often make the motors grind, or sometimes slow down.

On the layout of our PCB, one of the motor driver connections was not properly connected to the microcontroller (the STEP pin of the motor driver that received the PWM for the motor) so there was no way to get that to work without ordering a new PCB.

The interrupt code for the limit switch also did not work properly, as the motors continued to spin after the switch was pressed. The first problem targeted was a faulty wire connection, however more troubleshooting is needed.

Conclusion

Ultimately, despite the shortcomings from this project, we enjoyed the process of designing this embedded system, and we hope to continue to improve the design and the code in the future. One of our largest setbacks was the parts shortage, which no one had control of anyway. We were originally going to incorporate a gyroscope into our design but seeing the cost and the quantity available steered us away quite early from that. Additionally, we never got our linear voltage regulator which held us back in assembling and testing our PCB, until we created a temporary solution (which was to short the connection for the regulator). Another thing that held us back a bit was work (whether from other classes or from our own student jobs). This led to a lack of time management skills, which was our own fault.

If we were to restart the project, one important thing we would do differently is to take the assembly options in MPLab into greater consideration. The C code, while more efficient, does not give the entire picture into which registers are being used and controlled.

References

Abedin-Nasab, M. H., & Saeedi-Hosseiny, M. S. (2020). Robossis: Orthopedic Surgical Robot. In Handbook of Robotic and Image-Guided Surgery (pp. 515–528). Elsevier. https://doi.org/10.1016/b978-0-12-814245-5.00030-x

Kucuk, S. (Ed.). (2012). Serial and Parallel Robot Manipulators - Kinematics, Dynamics, Control and Optimization. InTech. https://doi.org/10.5772/2301

Saeedi-Hosseiny, Alruwaili, F., McMillan, S., Iordachita, I., & Abedin-Nasab, M. H. (2021). A Surgical Robotic System for Long-Bone Fracture Alignment: Prototyping and Cadaver Study. IEEE Transactions on Medical Robotics and Bionics, 1–1. https://doi.org/10.1109/TMRB.2021.3129277

Appendices

Main.c

```
void main(void)
{
    // Initialize the device
    SYSTEM_Initialize();
    ADCC_Initialize();
    I2C1_Initialize();



    I2C1_WriteNBytes(SDA1, uint8_t *data, size_t len);
    // If using interrupts in PIC18 High/Low Priority Mode you need to enable the Global High and Low
Interrupts
    // If using interrupts in PIC Mid-Range Compatibility Mode you need to enable the Global Interrupts
    // Use the following macros to:

    // Enable the Global Interrupts
    INTERRUPT_GlobalInterruptEnable();

    // Disable the Global Interrupts
    //INTERRUPT_GlobalInterruptDisable();
    ENN2_SetLow();
    ENN3_SetLow();
    DIR2_SetLow();
    DIR3_SetLow();

    while (1)
    {
        if(Limit1_GetValue() == 1)
        {
            ENN1_SetLow();
        }
        if(Limit2_GetValue() == 1)
        {
            ENN2_SetLow();
        }
        if(Limit3_GetValue() == 1)
        {
            ENN3_SetLow();
        }

        if(170 >= ADCC_GetSingleConversion(Pot1))
         {
            ENN1_SetLow();
            DIR1_SetHigh();
            break;
```

```
    }
    else if (340 >= ADCC_GetSingleConversion(Pot1))
    {
       ENN1_SetHigh();
       DIR1_SetLow();
       break;
    }
    else
    {
       ENN1_SetLow();
       DIR1_SetLow();
       break;
    }



   if(170 >= ADCC_GetSingleConversion(Pot2))
    {
       ENN2_SetLow();
       DIR2_SetHigh();
       break;
    }
    else if (340 >= ADCC_GetSingleConversion(Pot2))
    {
       ENN2_SetHigh();
       DIR2_SetLow();
       break;
    }
    else
    {
       ENN2_SetLow();
       DIR2_SetLow();
       break;
    }

    if(170 >= ADCC_GetSingleConversion(Pot3))
    {
       ENN3_SetLow();
       DIR3_SetHigh();
       break;
    }
    else if (340 >= ADCC_GetSingleConversion(Pot3))
    {
       ENN3_SetHigh();
       DIR3_SetLow();
       break;
    }
    else
```

```
        {
            ENN3_SetLow();
            DIR3_SetLow();
            break;
        }

    }
}
```

ADC Settings

```
void ADCC_Initialize(void)
{
    // set the ADCC to the options selected in the User Interface
    // ADLTH 0;
    ADLTHL = 0x00;
    // ADLTH 0;
    ADLTHH = 0x00;
    // ADUTH 0;
    ADUTHL = 0x00;
    // ADUTH 0;
    ADUTHH = 0x00;
    // ADSTPT 0;
    ADSTPTL = 0x00;
    // ADSTPT 0;
    ADSTPTH = 0x00;
    // ADACC 0;
    ADACCU = 0x00;
    // ADRPT 0;
    ADRPT = 0x00;
    // PCH ANA0;
    ADPCH = 0x00;
    // ADACQ 0;
    ADACQL = 0x00;
    // ADACQ 0;
    ADACQH = 0x00;
    // CAP Additional uC disabled;
    ADCAP = 0x00;
    // ADPRE 0;
    ADPREL = 0x00;
    // ADPRE 0;
    ADPREH = 0x00;
    // ADDSEN disabled; ADGPOL digital_low; ADIPEN disabled; ADPPOL Vss;
    ADCON1 = 0x00;
    // ADCRS 1; ADMD Basic_mode; ADACLR disabled; ADPSIS RES;
    ADCON2 = 0x10;
    // ADCALC First derivative of Single measurement; ADTMD disabled; ADSOI ADGO not cleared;
    ADCON3 = 0x00;
```

```
    // ADMATH registers not updated;
    ADSTAT = 0x00;
    // ADNREF VSS; ADPREF VDD;
    ADREF = 0x00;
    // ADACT disabled;
    ADACT = 0x00;
    // ADCS FOSC/2;
    ADCLK = 0x00;
    // ADGO stop; ADFM right; ADON enabled; ADCS FOSC/ADCLK; ADCONT disabled;
    ADCON0 = 0x84;
}

void ADCC_StartConversion(adcc_channel_t channel)
{
    // select the A/D channel
    ADPCH = channel;

    // Turn on the ADC module
    ADCON0bits.ADON = 1;

    // Start the conversion
    ADCON0bits.ADGO = 1;
}

adc_result_t ADCC_GetSingleConversion(adcc_channel_t channel)
{
    // select the A/D channel
    ADPCH = channel;

    // Turn on the ADC module
    ADCON0bits.ADON = 1;

    //Disable the continuous mode.
    ADCON0bits.ADCONT = 0;

    // Start the conversion
    ADCON0bits.ADGO = 1;


    // Wait for the conversion to finish
    while (ADCON0bits.ADGO)
    {
    }

    // Conversion finished, return the result
    return ((adc_result_t)((ADRESH << 8) + ADRESL));
}
```

I2C

```
void I2C1_Initialize()
{
   //EN disabled; RSEN disabled; S Cleared by hardware after Start; CSTR Enable clocking; MODE 7-bit address;
   I2C1CON0 = 0x04;
   //ACKCNT Acknowledge; ACKDT Acknowledge; ACKSTAT ACK received; ACKT 0; RXO 0; TXU 0; CSD Clock Stretching enabled;
   I2C1CON1 = 0x80;
   //ACNT disabled; GCEN disabled; FME disabled; ABD enabled; SDAHT 300 ns hold time; BFRET 8 I2C Clock pulses;
   I2C1CON2 = 0x18;
   //CLK MFINTOSC;
   I2C1CLK = 0x03;
   //CNTIF 0; ACKTIF 0; WRIF 0; ADRIF 0; PCIF 0; RSCIF 0; SCIF 0;
   I2C1PIR = 0x00;
   //CNTIE disabled; ACKTIE disabled; WRIE disabled; ADRIE disabled; PCIE disabled; RSCIE disabled; SCIE disabled;
   I2C1PIE = 0x00;
   //BTOIF No bus timeout; BCLIF No bus collision detected; NACKIF No NACK/Error detected; BTOIE disabled; BCLIE disabled; NACKIE disabled;
   I2C1ERR = 0x00;
   //Count register
   I2C1CNT = 0xFF;
   return;
}
```

PWM1

(PWM2 and PWM3 are identical)

```
void PWM1_16BIT_Initialize(void)
{
   //PWMERS External Reset Disabled;
   PWM1ERS = 0x00;

   //PWMCLK FOSC;
   PWM1CLK = 0x02;

   //PWMLDS Autoload disabled;
   PWM1LDS = 0x00;

   //PWMPRL 231;
   PWM1PRL = 0xE7;

   //PWMPRH 3;
   PWM1PRH = 0x03;
```

```
//PWMCPRE No prescale;
PWM1CPRE = 0x00;

//PWMPIPOS No postscale;
PWM1PIPOS = 0x00;

//PWMS1P2IF PWM2 output match did not occur; PWMS1P1IF PWM1 output match did not occur;
PWM1GIR = 0x00;

//PWMS1P2IE disabled; PWMS1P1IE disabled;
PWM1GIE = 0x00;

//PWMPOL2 disabled; PWMPOL1 disabled; PWMPPEN disabled; PWMMODE Left aligned mode;
PWM1S1CFG = 0x00;

//PWMS1P1L 244;
PWM1S1P1L = 0xF4;

//PWMS1P1H 1;
PWM1S1P1H = 0x01;

//PWMS1P2L 244;
PWM1S1P2L = 0xF4;

//PWMS1P2H 1;
PWM1S1P2H = 0x01;

//Clear PWM1_16BIT period interrupt flag
PIR4bits.PWM1PIF = 0;

//Clear PWM1_16BIT interrupt flag
PIR4bits.PWM1IF = 0;

//Clear PWM1_16BIT slice 1, output 1 interrupt flag
PWM1GIRbits.S1P1IF = 0;

//Clear PWM1_16BIT slice 1, output 2 interrupt flag
PWM1GIRbits.S1P2IF = 0;

//PWM1_16BIT interrupt enable bit
PIE4bits.PWM1IE = 0;

//PWM1_16BIT period interrupt enable bit
PIE4bits.PWM1PIE = 0;

//Set default interrupt handlers
```

```
PWM1_16BIT_Slice1Output1_SetInterruptHandler(PWM1_16BIT_Slice1Output1_DefaultInterruptHandler);

PWM1_16BIT_Slice1Output2_SetInterruptHandler(PWM1_16BIT_Slice1Output2_DefaultInterruptHandler);
   PWM1_16BIT_Period_SetInterruptHandler(PWM1_16BIT_Period_DefaultInterruptHandler);

   //PWMEN enabled; PWMLD disabled; PWMERSPOL disabled; PWMERSNOW disabled;
   PWM1CON = 0x80;
}

void PWM1_16BIT_Enable()
{
   PWM1CON |= _PWM1CON_EN_MASK;
}
```

UART

```
void UART1_Initialize(void)
{
   // Disable interrupts before changing states

   // Set the UART1 module to the options selected in the user interface.

   // P1L 0;
   U1P1L = 0x00;

   // P1H 0;
   U1P1H = 0x00;

   // P2L 0;
   U1P2L = 0x00;

   // P2H 0;
   U1P2H = 0x00;

   // P3L 0;
   U1P3L = 0x00;

   // P3H 0;
   U1P3H = 0x00;

   // BRGS high speed; MODE Asynchronous 8-bit mode; RXEN enabled; TXEN enabled; ABDEN disabled;
   U1CON0 = 0xB0;
```

```
    // RXBIMD Set RXBKIF on rising RX input; BRKOVR disabled; WUE disabled; SENDB disabled; ON
enabled;
    U1CON1 = 0x80;

    // TXPOL not inverted; FLO off; C0EN Checksum Mode 0; RXPOL not inverted; RUNOVF RX input
shifter stops all activity; STP Transmit 1Stop bit, receiver verifies first Stop bit;
    U1CON2 = 0x00;

    // BRGL 25;
    U1BRGL = 0x19;

    // BRGH 0;
    U1BRGH = 0x00;

    // STPMD in middle of first Stop bit; TXWRE No error;
    U1FIFO = 0x00;

    // ABDIF Auto-baud not enabled or not complete; WUIF WUE not enabled by software; ABDIE
disabled;
    U1UIR = 0x00;

    // ABDOVF Not overflowed; TXCIF 0; RXBKIF No Break detected; RXFOIF not overflowed; CERIF No
Checksum error;
    U1ERRIR = 0x00;

    // TXCIE disabled; FERIE disabled; TXMTIE disabled; ABDOVE disabled; CERIE disabled; RXFOIE
disabled; PERIE disabled; RXBKIE disabled;
    U1ERRIE = 0x00;


    UART1_SetFramingErrorHandler(UART1_DefaultFramingErrorHandler);
    UART1_SetOverrunErrorHandler(UART1_DefaultOverrunErrorHandler);
    UART1_SetErrorHandler(UART1_DefaultErrorHandler);

    uart1RxLastError.status = 0;

}

uint8_t UART1_Read(void)
{
    while(!PIR4bits.U1RXIF)
    {
    }

    uart1RxLastError.status = 0;

    if(U1ERRIRbits.FERIF){
        uart1RxLastError.ferr = 1;
```

```
        UART1_FramingErrorHandler();
    }

    if(U1ERRIRbits.RXFOIF){
        uart1RxLastError.oerr = 1;
        UART1_OverrunErrorHandler();
    }

    if(uart1RxLastError.status){
        UART1_ErrorHandler();
    }

    return U1RXB;
}
```