690 Project 3: Motor Control

Luke Weaver

February 7, 2018

Abstract

The purpose of this project is to configure a DC Motor controlled by a PWM signal. The motor has a built in encoder; the encoder gets information from the motor such as velocity, direction, and position. Upon running the program, UART will display two options; option 1 will ask the user for a velocity between -100 to 100 inclusive. A value of 100 indicating full speed clockwise, -100 full speed counter clockwise, and 0 to a stop. Choosing option 1 will also report the direction and velocity to the ReportData facility. Option 2 will turn the motor axle to a specified position; after selecting option 2 UART will ask for a degree between -180 and 180 inclusive to rotate the motor.

Contents

List of Figures

List of Tables

none

1 Principles of Operation

The Motor is powered by an external power supply and has an input channel which is configured to port G pin 1; this pin can be found on the Tiva LaunchPad's GPIO pin header. The decoder has two input channels connected to the Tiva board. The motors channels are connected to Port L Pin 1 and 2. These gpio pins can be found on the BoosterPack1 header. Listing the pin configuration is crucial in order to recreate the lab. In order to rotate the motor I used PWM functions that can be found in the PWM header file. By creating a PWM signal I was able to rotate the motor at various speeds. The next step was to gather data from the motor via the encoder. After enabling the aforementioned pins I was able to configure the quadratic encoder interface and gather information such as velocity and position. Once I knew the position of the motor, I was able to write a program to rotate the motor from -180 to 180 degrees. A top-level block diagram of the design is shown below:

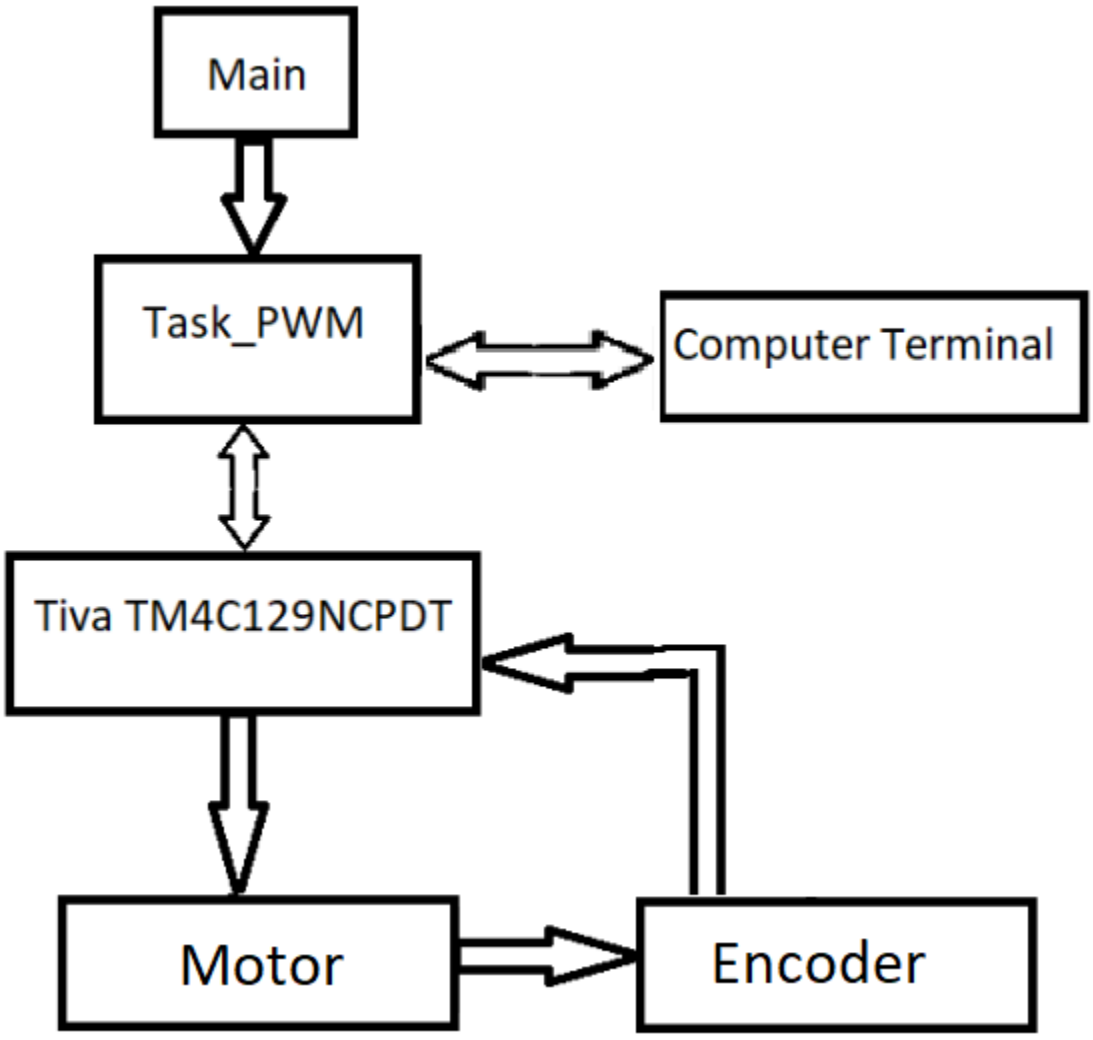

Figure 1: Top-Level Block Diagram

`Task_PWM`, is instantiated within the main using a function `xTaskCreate()`. This is part of the FreeRTOS language and is used to add a task to a list of tasks running; this is used for memory allocation. Upon creation of the PWM task, the PWM is initialized and the task asks the user which mode they want to enter. Mode 1 will run task 1 of the project and Mode 2 implements Task 2 and 3. If the user selects 1, the program will

prompt the user to set a direction and velocity of the motor ranging from -100 to 100 and if the user selects 2 the program will prompt the user to set a position of the motor ranging from -180 to 180.

2 Data Structure Descriptions

```
No major data structures were used.
```

3 Function Descriptions

3.1 Task_PWM(void)

The PWM task controls all order of operation. Task_PWM initializes and implements the motor and decoder.

3.2 extern void PWMClockSet(uint32_t ui32Base, uint32_t ui32Config)

Sets the PWM clock configuration.

3.3 void PWMGenConfigure(uint32_t ui32Base, uint32_t ui32Gen, uint32_t ui32Config)

This function configures a PWM generator to be used.

3.4 void PWMGenPeriodSet(uint32_t ui32Base, uint32_t ui32Gen,uint32_t ui32Period)

Sets the period of a PWM generator.

3.5 void PWMPulseWidthSet(uint32_t ui32Base, uint32_t ui32PWMOut,uint32_t ui32Width)

This function sets the pulse width for the specified PWM output.

3.6 void PWMGenEnable(uint32_t ui32Base, uint32_t ui32Gen)

This function enables PWM generator 2.

3.7 void PWMOutputState(uint32_t ui32Base, uint32_t ui32PWMOutBits,bool bEnable)

This function enables or disables PWM outputs.

3.8 void QEIConfigure(uint32_t ui32Base, uint32_t ui32Config,uint32_t ui32MaxPosition)

Configures the quadrature encoder.

3.9 void QEIEnable(uint32_t ui32Base)

Enables the quadrature encoder.

3.10 void QEIVelocityEnable(uint32_t ui32Base)

Enables the velocity capture.

3.11 void QEIVelocityConfigure(uint32_t ui32Base, uint32_t ui32PreDiv,uint32_t ui32Period)

Configures the velocity capture.

3.12 uint32_t QEIPositionGet(uint32_t ui32Base)

Gets the current encoder position.

3.13 Main_Sharp1_Test.c(void)

Instantiates Task_PWM and starts the FreeRTOS Task Scheduler.

3.14 PsuedoCode

```
Project 3 Psuedo Code

//function call in main
//xtaskcreate(Task_PWM())

//function inside Task_PWM.c
extern void Task_PWM( void )
{
PWM_Initialization();
enable peripherials(Port L and G)
configure pwm and qei
uint userinput
public float pulsewidth;
pulsewidthset( user input);
if(postion of motor == desired position)
{
stop rotating();
}
uart (qeipositionget)
uart (qeivelocityget)

}
```

4 Parameters

4.1 char mode[1]

This parameter captures user input via UART; mode determines which mode the user wishes to enter.

4.2 char UserInput[3]

This parameter captures user input via UART;

userInput is to be used to set the desired velocity or position.

4.3 uint32_t currentMultiplier

This integer converts the userInput character array to an integer.

4.4 uint32_t servoPW

This value is used to set the pulse width for the motor.

4.5 uint32_t servoPeriod

This value is used to set the period for the motor.

4.6 uint32_t servoPosition

This value stores the desired position of the motor.

4.7 uint32_t qeipos

This value gets the current position of the motor.

5 Testing

First, I would test my PWM code on a servo since they were more widely available and required less set-up. I used a servo that could rotate freely without constraints to emulate a motor as closely as possible. Once I was able to enter proper velocity input through UART I felt confident enough to hook up the motor. At first I thought I set up the motor incorrectly but I later learned that some of the motor controllers were not functioning properly. Once I properly set up the motor the next step was to gather information from the encoder. I had to set up channels A and B to port L pin 1 and 2 respectively. I learned that these pins are not interchangeable; as Port L pin 1 is expecting input from channel A and vice versa for Port L pin 2.

I spent a majority of my time testing part 3 of the project (rotating motor a certain degree from origin). There seems to be residual rotation after the motor finishes rotating. The way I got around this was to add leniency, a +- 2 tics of uncertainty so that even if the motor rotated a little too much, it would still register as a degree. This translates to a +-.64 degree uncertainty.

After experimenting with setting various velocities and positions I was able to accurately set and retrieve the information from the QEI. Figure 2 tests task 1's output. The parameters fed into the Que are: time-stamp, task name, velocity, and position. Figure 3 tests task 1's output. The parameters fed into the Que are: time-stamp, task name, and position.

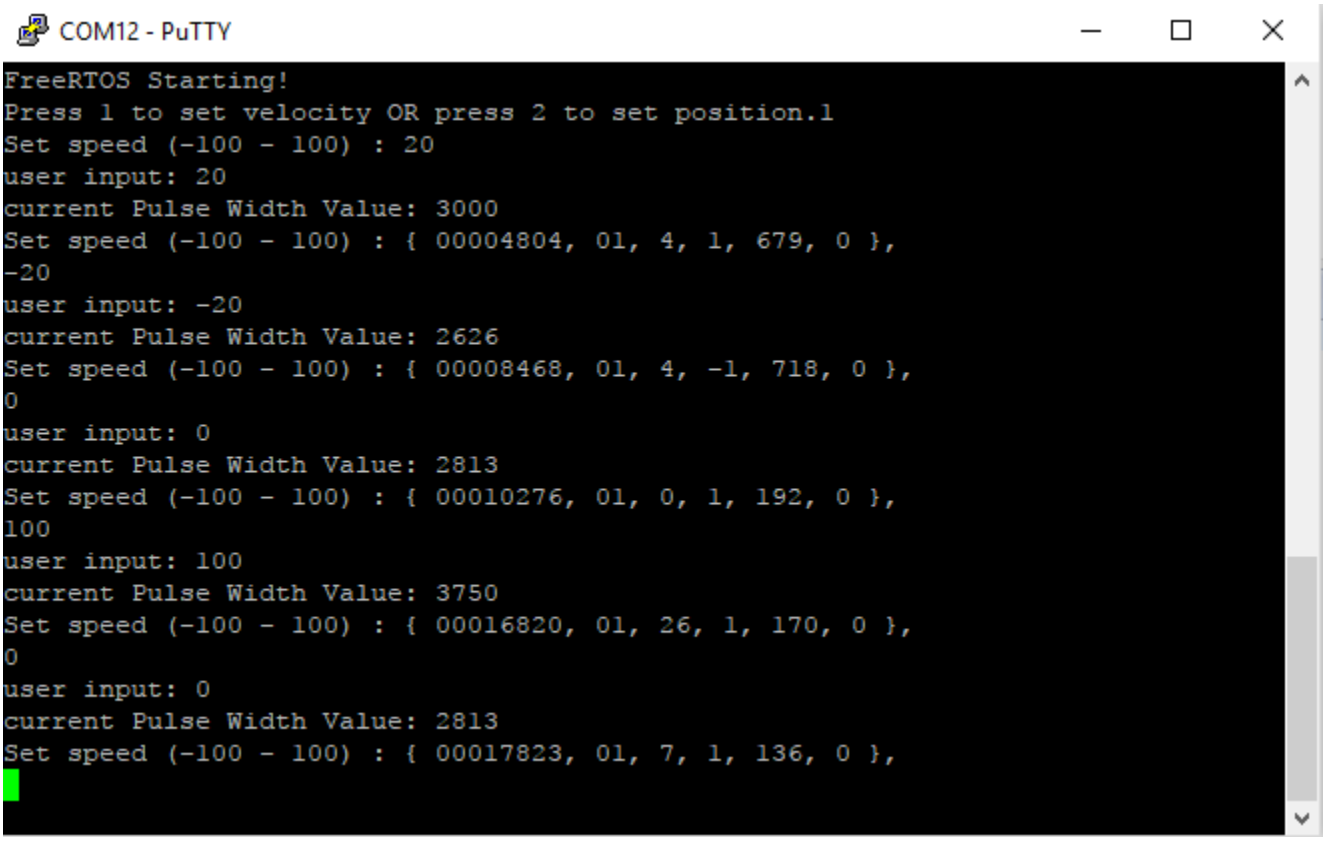

Figure 2: UART output task 1

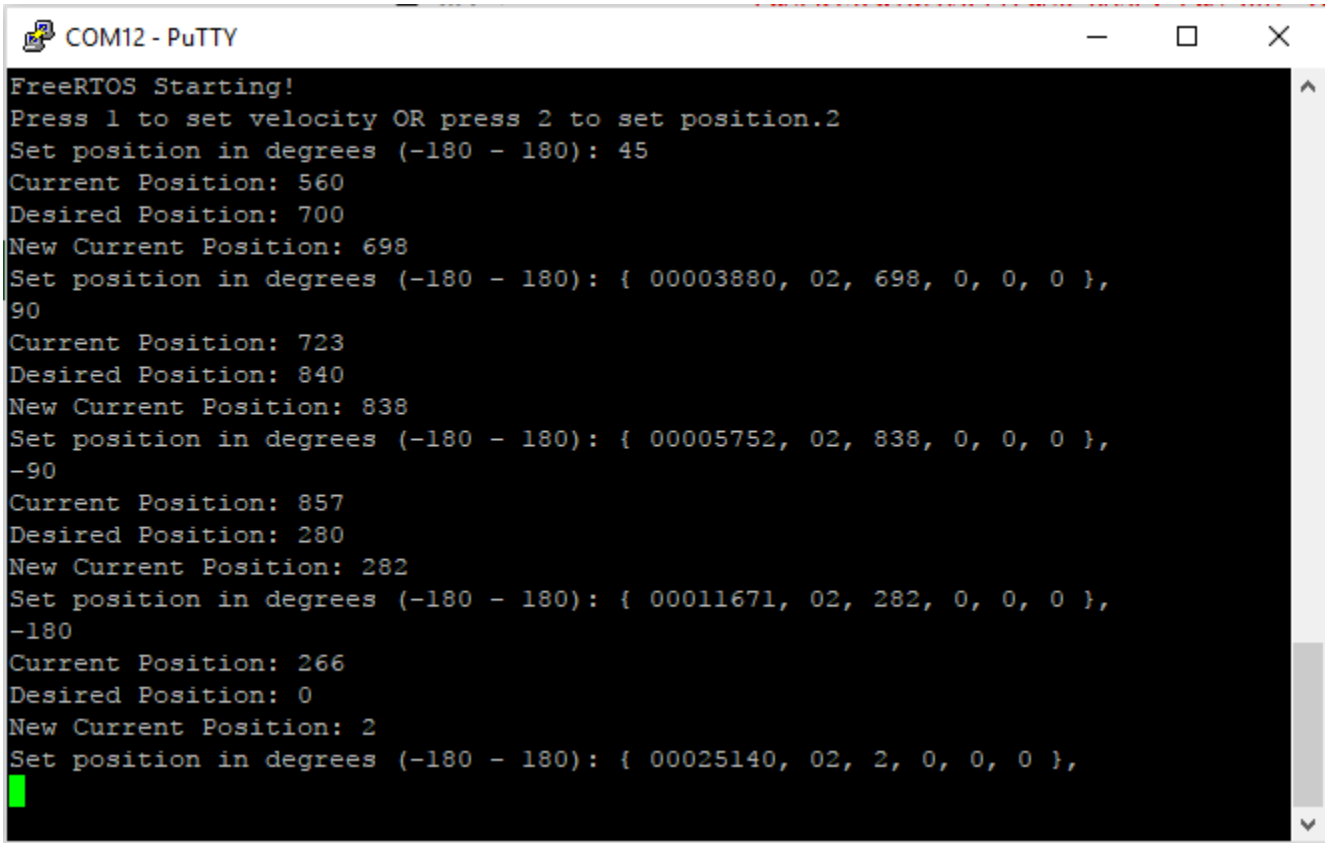

Figure 3: UART output task 2 and 3

6 Lessons Learned and Revision History

Luckily this project uses components from project 2. I utilized the PWM knowledge I learned from project 2 to drive the motor. With Doctor Minden's help I was able to locate the correct pins to use on the Tiva for channel A and B on the QEI. I thoroughly enjoyed this project and wish we had time to do a fourth project. The new concepts used in this project include PWM motor control, and UART input. I have greater knowledge on motor control and plan on using the knowledge gained in my senior design project.

7 Program Listing

7.1 Main_NoSharp1_Test.c

```
/*--Main_Sharp1_Test.c
 *
 *  Author: Luke Weaver
 * Organization: KU/EECS/EECS 690
 *  Date: October 30, 2017 (B60313)
 *
 *  Description: Starts RTOS Scheduler and Creates tasks: Task_PWM
 *
 *
 *
 *
 */


/*
 * main.c
 */

#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_uart.h"

#include <stddef.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdarg.h>

#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"
#include "driverlib/gpio.h"

#include "Drivers/Processor_Initialization.h"
#include "Drivers/UARTStdio_Initialization.h"
#include "Drivers/uartstdio.h"

#include "FreeRTOS.h"
#include "task.h"

#include <stdio.h>

extern void Task_PWM( void *pvParameters );
//extern void Task_MotorPosition( void *pvParameters );
extern void Task_Blink_LED_PortN_1( void *pvParameters );
extern void Task_ReportTime( void *pvParameters );
extern void Task_ReportData( void *pvParameters );

//extern void Task_I2C7_Handler( void *pvParameters );

int main( void ) {
```

```
Processor_Initialization();
UARTStdio_Initialization();
//
// Create a task to blink LED
//
xTaskCreate( Task_Blink_LED_PortN_1, "Task_Blink_LED_PortN_1", 128, NULL, 1, NULL );
UARTprintf( "FreeRTOS Starting!\n" );

//
// Create a task to report data.
//
xTaskCreate( Task_ReportData, "ReportData", 512, NULL, 1, NULL );




xTaskCreate( Task_PWM, "PWM", 512, NULL, 1, NULL );
//xTaskCreate( Task_MotorPosition, "PWM", 512, NULL, 1, NULL );



//
// Create a task to report SysTickCount
//
//xTaskCreate( Task_ReportTime, "ReportTime", 512, NULL, 1, NULL );

//
// Create a task to initialize I2C7_Handler.
//
//xTaskCreate( Task_I2C7_Handler, "I2C7_Handler", 1024, NULL, 2, NULL );


//
// Start FreeRTOS Task Scheduler
//
vTaskStartScheduler();

while ( 1 ) {

}

}
```

7.2 Task_PWM.c

```
/*--Task_PWM.c
 *
 *  Author: Luke Weaver
 * Organization: KU/EECS/EECS 690
 *  Date: November 28, 2017
 *
 *  Description: Controls motor functions
 *
 *
 *
 *
 */

#include    <stddef.h>
#include    <stdbool.h>
#include    <stdint.h>
#include    <stdarg.h>

#include "inc/hw_types.h"
#include "inc/hw_gpio.h"
#include "inc/hw_memmap.h"
#include "inc/hw_uart.h"
#include "inc/hw_timer.h"

#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/pwm.h"
#include "driverlib/timer.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
#include "driverlib/qei.c"
#include "FreeRTOS.h"
#include    "task.h"
#include    "Tasks/Task_ReportData.h"


char userInput[3];
int32_t currentMultiplier;
uint32_t servoPW;
uint32_t servoPeriod;
int32_t servoPosition;
int32_t qeipos;
char mode[1];

extern void Task_PWM( void *pvParameters ) {

    ReportData_Item        QeiInfo;


    servoPeriod = 120000000/3200;
```

```
servoPW = 2813;
SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);
SysCtlPeripheralEnable(SYSCTL_PERIPH_QEI0);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG); //port G
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOL); //port L

GPIOPinConfigure(GPIO_PG1_M0PWM5); //pg p1
GPIOPinConfigure(GPIO_PL1_PHA0); //pg L1
GPIOPinConfigure(GPIO_PL2_PHB0); //pg L2

GPIOPinTypeQEI(GPIO_PORTL_BASE, GPIO_PIN_1);
GPIOPinTypeQEI(GPIO_PORTL_BASE, GPIO_PIN_2);
//pwm
GPIOPinTypePWM(GPIO_PORTG_BASE, GPIO_PIN_1);
PWMClockSet(PWM0_BASE, PWM_SYSCLK_DIV_64);
PWMGenConfigure(PWM0_BASE, PWM_GEN_2, (PWM_GEN_MODE_DOWN | PWM_GEN_MODE_NO_SYNC));
PWMGenPeriodSet(PWM0_BASE, PWM_GEN_2, servoPeriod);
PWMPulseWidthSet(PWM0_BASE, PWM_OUT_5, (servoPW));
PWMGenEnable(PWM0_BASE, PWM_GEN_2);
PWMOutputState(PWM0_BASE, (PWM_OUT_5_BIT), true);

//qei
  QEIConfigure(QEI0_BASE,(QEI_CONFIG_CAPTURE_A_B | QEI_CONFIG_RESET_IDX | QEI_CONFIG_QUADRATURE | QE
  QEIEnable (QEI0_BASE);
  QEIVelocityEnable(QEI0_BASE);
  QEIVelocityConfigure(QEI0_BASE,QEI_VELDIV_1,1120000);
  QEIPositionSet(QEI0_BASE,560); //set baseline
/*
 *     servoPeriod = 40704; //16 least signifcant bits 1001111100000000. 20ms
    servoPW = 43392; //54464 cw, 43392 ccw
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER3);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    GPIOPinConfigure(GPIO_PA7_T3CCP1); //Sets up pin to use alternative function. The General Purpos
    GPIOPinTypeTimer(GPIO_PORTA_BASE, GPIO_PIN_7); //Sets as a timer pin.

    TimerConfigure(TIMER3_BASE, TIMER_CFG_SPLIT_PAIR|TIMER_CFG_B_PWM); //Specifies mode as PWM.
    TimerMatchSet(TIMER3_BASE, TIMER_B, servoPW);                    //Initialized Pulse Width valu
    TimerPrescaleMatchSet(TIMER3_BASE,TIMER_B,3); //counter clock wise, 2 clockwise
    TimerPrescaleSet(TIMER3_BASE,TIMER_B,36); //8 most significant bits 100100, extends of 16-bit ti
    TimerLoadSet(TIMER3_BASE, TIMER_B, servoPeriod);                      //Set period
    TimerEnable(TIMER3_BASE, TIMER_B);                                   //enables timer
 */
    UARTprintf("Press 1 to set velocity OR press 2 to set position.");
    UARTgets(mode,2);
    while (1) {
        if(mode[0] == '1')
        {
        UARTprintf("Set speed (-100 - 100) : ");
        //alternate between high and low pulse widths
        userInput[0] = NULL;
        userInput[1] = NULL;
        userInput[2] = NULL;
        userInput[3] = NULL;
```

```
    //set pulse width
    //TimerMatchSet(TIMER3_BASE, TIMER_B, servoPW);
    vTaskDelay( ( 5000 * configTICK_RATE_HZ ) / 10000 );
    UARTgets(userInput,5);
    currentMultiplier = atoi(userInput);
    if(currentMultiplier>-101 && currentMultiplier<101)
    {
//1875 - 3750
     servoPW = 2813 + (937*currentMultiplier)/100;
    }
    else
    {
        UARTprintf("%d is not a valid input. Please choose a value between -100 and 100 inclusi
    }
    PWMPulseWidthSet(PWM0_BASE, PWM_OUT_5, (servoPW));

    UARTprintf("user input: %d \n",currentMultiplier);
    UARTprintf("current Pulse Width Value: %d \n",servoPW);

    vTaskDelay( ( 5000 * configTICK_RATE_HZ ) / 10000 );

    QeiInfo.TimeStamp = xPortSysTickCount;
    QeiInfo.ReportName = 1;
    QeiInfo.ReportValue_0 = QEIVelocityGet(QEI0_BASE);
    QeiInfo.ReportValue_1 = QEIDirectionGet(QEI0_BASE);
    QeiInfo.ReportValue_2 = QEIPositionGet(QEI0_BASE);
    QeiInfo.ReportValue_3 = 0;
    xQueueSend(  ReportData_Queue, &QeiInfo, 0 );


    //UARTprintf("Position: %d \n",QEIPositionGet(QEI0_BASE));
    //UARTprintf("Velocity: %d \n",QEIVelocityGet(QEI0_BASE));
    //UARTprintf("Direction: %d \n",QEIDirectionGet(QEI0_BASE));
    currentMultiplier = 0;
    }


    else if(mode[0] == '2')
    {
        //get user input
     qeipos = QEIPositionGet(QEI0_BASE);
    UARTprintf("Set position in degrees (-180 - 180): ");
    userInput[0] = NULL;
    userInput[1] = NULL;
    userInput[2] = NULL;
    userInput[3] = NULL;
    vTaskDelay( ( 5000 * configTICK_RATE_HZ ) / 10000 );
    UARTgets(userInput,5);
    currentMultiplier = atoi(userInput);
    if(currentMultiplier>-181 && currentMultiplier<181){

        servoPosition = 560+ (currentMultiplier*560)/180; //destination
    }
```

```
            else{
                UARTprintf("%d is not a valid input. Please choose a value between -180 and 180 inclusi
                servoPosition = 560;
            }

            vTaskDelay( ( 5000 * configTICK_RATE_HZ ) / 10000 );

            //Determine which direction to go

            if(qeipos - servoPosition > 0) //ccw
            {
                PWMPulseWidthSet(PWM0_BASE, PWM_OUT_5, (2700));
            }
            else if(qeipos - servoPosition == 0)
            {
                PWMPulseWidthSet(PWM0_BASE, PWM_OUT_5, (2813)); //zero
            }
            else
            {
                PWMPulseWidthSet(PWM0_BASE, PWM_OUT_5, (2950)); //cw
            }
            UARTprintf("Current Position: %d \n",QEIPositionGet(QEI0_BASE));
            UARTprintf("Desired Position: %d \n",servoPosition);
            while(abs((qeipos - servoPosition)) > 2)
            {

                qeipos = QEIPositionGet(QEI0_BASE);
                vTaskDelay( configTICK_RATE_HZ / 1000 );
            }
            UARTprintf("New Current Position: %d \n",QEIPositionGet(QEI0_BASE));
            PWMPulseWidthSet(PWM0_BASE, PWM_OUT_5, (2813));

            //report data
            QeiInfo.TimeStamp = xPortSysTickCount;
            QeiInfo.ReportName = 2; //1 is velocity, 2 is position
            QeiInfo.ReportValue_0 = QEIPositionGet(QEI0_BASE);
            QeiInfo.ReportValue_1 = 0;
            QeiInfo.ReportValue_2 = 0;
            QeiInfo.ReportValue_3 = 0;
            xQueueSend(  ReportData_Queue, &QeiInfo, 0 );
            currentMultiplier = 0;
            servoPosition = 560;
            }


            else
            {
                UARTprintf("Enter a valid option 1 or 2.");
                UARTgets(mode,2);
            }
        }
}
```