

Text, letter

Description automatically generated

**Introduction:**

The purpose of this lab is to familiarize students with pulse width modulation using the Texas Instruments TM4C123GH6PM microcontroller. Students will continue to gain practice with the temperature module and the digilent 8 led module. The control system follows a structure similar to what might be implemented in a nuclear reactor.

**Body:**

In this control system, the position of the servo motor is controlled by a potentiometer; the position of the potentiometer on its axis should linearly adjust the position of the servo and the lights on the 8 led module. The system needs to display the ambient temperature on the LCD display, and if the temperature gets too high, the system needs to display a message that "scrams" the system.

For example:

| **Potentiometer** | **Servo** | **Leds** |
| --- | --- | --- |
| 100% | +90 deg | 8 lights on |
| 50% | 0 deg | 4 lights on |
| 0% | -90deg | 0 lights on |

**Pseudocode:**

The logic for the program can be implemented as follows:

initializeSystem()

while(True):

currentTemperature = getCurrentTemperature()

if(currentTemperature > threshold):

scramSystem()

displayCurrentTemperatureOnLcdDisplay(currentTemperature)

currentPosition = getCurrentPositionOfPotentiometer()

adjustFuelRodsAkaServoMotor(currentPosition)

displayLightsOnLedModule(currentPosition)

## Source Code:

/\* PWM\_ADC\_servo.c

\*

\* This program demonstrates the use of the PWM module to generate

\* a varying duty cycle PWM output. The program uses a potentiometer

\* in a voltage divider circuit to provide a varying amplitude

\* signal to the ADC. The digital output is then used to modify

\* the compare register, resulting in a changing duty cycle on

\* the PWM output.

\*

\* This program is configured to drive a servo motor where

\* the control pulse ranges from 1 - 2 mSec. The ADC output is

\* scaled so that the position can be more precisely controlled.

\*

\* The output signal is M0PWM3 which is output on GPIO PB5

\* The PWM source is module 0, generator 1, pwmB

\*

\* The program uses direct register access mode.

\*

\* The program uses a sytem clock of 80 Mhz.

\*

\*

\* written by: E.J. Nava, UNM, 11/05/22

\*

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include <stdint.h>

#include <stdbool.h>

#include <stddef.h>

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

#include <tm4c123gh6pm.h>

#define DELAY\_VALUE 0xF9F // 0xF9F = 1 mSec delay at 4 MHz

#define SYSDIV2 4

#define SLAVE\_ADDR 0x4F /\* 0100 1111 \*/

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Prototypes \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

void init\_SSI0(void);

void SSI0Write(unsigned char data);

void putsSPI0(size\_t buflen, char \* buffer);

void SysTick\_init(void);

void SysTick\_mSecDelay(uint32\_t delay);

void PWM0\_0\_init(void);

void init\_ADC0(void);

void I2C1\_init(void);

char I2C1\_byteWrite(int slaveAddr, char data);

char I2C1\_burstWrite(int slaveAddr, int byteCount, char\* data);

char I2C1\_read(int slaveAddr, int byteCount, char\* data);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Global Variables \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

int result = 0;

char i2c\_data[4]; // buffer for date read from or written to I2C

char \*i2c\_char\_p = &i2c\_data[0];

void enablePhaseLockLoopByClearingBypass()

{

SYSCTL\_RCC2\_R &= ~0x00000800; // Enable PLL by clearing BYPASS

}

bool checkIfSystemIsOverheating(const int currentTemperatureInFahrenheit)

{

const int temperatureThreshold = 78;

if(currentTemperatureInFahrenheit > temperatureThreshold)

{

return true;

}

return false;

}

void waitForPhaseLockLoopToLock()

{

while((SYSCTL\_RIS\_R&0x00000040)==0)

{

}; // Wait for PLL to lock - poll PLLRIS

}

void configureI2C()

{

i2c\_data[0] = 0; // select register 0 on TMP3 module

}

void init\_SSI0(void)

{

SYSCTL\_RCGCSSI\_R |= 1; // enable clock to SSI0

SYSCTL\_RCGCGPIO\_R |= 0x1; // Enable clock to PORT A

/\* configure PORTA 2..5 for SSI0 clock, FS, Tx & Rx \*/

GPIO\_PORTA\_AMSEL\_R = 0; // turn off analog function

GPIO\_PORTA\_DEN\_R |= 0x3C; // make PA2..PA5 digital

GPIO\_PORTA\_AFSEL\_R = 0x3C; // make PA2.. PA5 alternate function

GPIO\_PORTA\_PCTL\_R = 0x00222200; // configure PA2..PA5 as SSI0

/\* SPI Master, POL = 0, PHA = 0, SysClk = 80 MHz, 8 bit data \*/

SSI0\_CR1\_R = 0; // disable SSI and make it master

SSI0\_CC\_R = 0; // use system clock

SSI0\_CPSR\_R = 0x64; // prescaler divided by 100

SSI0\_CR0\_R = 0x0707; // 800KHz/8 = SSI clock, SPI mode, 8 bit data

SSI0\_CR1\_R |= 2; // enable SSI0

}

void initializeSystemClockFor80mhz()

{

SYSCTL\_RCC2\_R |= 0x80000000;

/\* Bypass the PLL while initializing it \*/

SYSCTL\_RCC2\_R |= 0x00000800;

/\* Select the crystal value and oscillator source \*/

SYSCTL\_RCC\_R = (SYSCTL\_RCC\_R & ~0x000007C0) // clear bits 10-6

+ 0x00000540; // 10101 configure for 16Mhz XTL

/\* Set for main oscillator source \*/

SYSCTL\_RCC2\_R &= ~0x00000070;

/\* Activate the PLL by clearing PWRDN \*/

SYSCTL\_RCC2\_R &= ~0x00002000;

/\* Set the desired system divider \*/

SYSCTL\_RCC2\_R |= 0x40000000;

SYSCTL\_RCC2\_R = (SYSCTL\_RCC2\_R & ~0x1FC00000) +(SYSDIV2<<22);

/\* Wait for the PLL to lock by polling PLLRIS \*/

while((SYSCTL\_RIS\_R&0x00000040)==0){};

/\* Enable use of the PLL by clearing BYPASS \*/

SYSCTL\_RCC2\_R &= ~0x00000800;

}

int convertDegreesCelsiusToDegreesFahrenheit(int inputTemperature)

{

return ((inputTemperature \* 9)/ 5) + 32;

}

int getCurrentTemperatureFromSensorInFahrenheit(const char slaveAddress)

{

int raw\_temp; // raw temp data in int form

short int a, b, currentTempInDegCelsius; // temperature variables

I2C1\_read(slaveAddress, 2, i2c\_char\_p);

a = (short int) i2c\_data[0]; // cast bytes to 16 bits

b = (short int) i2c\_data[1];

raw\_temp = b | (a << 8); // combine bytes to 9 bit result

currentTempInDegCelsius = raw\_temp >> 8; // shift out ls 7 bits of 0 divide by 256 -> 0 Deg C

return convertDegreesCelsiusToDegreesFahrenheit(currentTempInDegCelsius);

}

void displayDecimalNumberOnGpioBoardInBinary(uint8\_t decimalToDisplayInBinary)

{

const uint32\_t decimalToDisplayInBinaryMod256 = decimalToDisplayInBinary % 256;

GPIO\_PORTE\_DATA\_R = decimalToDisplayInBinary & 0xF; // lower 4 count bits - PE0..PE3

GPIO\_PORTD\_DATA\_R = (decimalToDisplayInBinary & 0xF0) >> 4; // upper 4 count bits - PD0..PD3

}

void initializeGPIOPorts()

{

SYSCTL\_RCGCGPIO\_R |= 0x38; // activate port D,E & F clocks

GPIO\_PORTF\_DIR\_R |= 0x04; // make PF2 out (built-in blue LED)

GPIO\_PORTE\_DIR\_R |= 0x0F; // make PE0..PE3 out

GPIO\_PORTD\_DIR\_R |= 0x0F; // make PD0..PD3 out

GPIO\_PORTF\_AFSEL\_R &= ~0x04;// disable alt funct on PF2

GPIO\_PORTE\_AFSEL\_R &= ~0x0F;// disable alt funct on PE0..PE3

GPIO\_PORTD\_AFSEL\_R &= ~0x0F;// disable alt funct on PD0..PD3

GPIO\_PORTF\_DEN\_R |= 0x04; // enable digital I/O on PF2

GPIO\_PORTE\_DEN\_R |= 0x0F; // enable digital I/O on PE0..PE3

GPIO\_PORTD\_DEN\_R |= 0x0F; // enable digital I/O on PD0..PD3

// configure PF2 as GPIO (Selectively - others left unchanged)

GPIO\_PORTF\_PCTL\_R = (GPIO\_PORTF\_PCTL\_R&0xFFFFF0FF)+0x00000000;

// configure PE0..PE3 as GPIO (Selectively - others left unchanged)

GPIO\_PORTE\_PCTL\_R = (GPIO\_PORTF\_PCTL\_R&0xFFFF0000)+0x00000000;

// configure PD0..PD3 as GPIO (Selectively - others left unchanged)

GPIO\_PORTD\_PCTL\_R = (GPIO\_PORTD\_PCTL\_R&0xFFFF0000)+0x00000000;

GPIO\_PORTF\_AMSEL\_R = 0; // disable analog functionality on PF

GPIO\_PORTE\_AMSEL\_R = 0; // disable analog functionality on PE

GPIO\_PORTD\_AMSEL\_R = 0; // disable analog functionality on PD

}

uint8\_t discreteUniform1400(const uint16\_t decimalToConvert){

const uint16\_t upperBound = 1400;

const uint16\_t stepSize = upperBound / 8;

for(uint8\_t i = 0; i < 9; i++)

{

if(decimalToConvert < (stepSize \* i))

{

return i - 1;

}

}

return 0;

}

uint8\_t convertDecimalToPercentLightBar(const uint16\_t decimalNumberToConvert)

{

uint8\_t interval = discreteUniform1400(decimalNumberToConvert);

if(0 == interval)

{

return 1;

}

if(1 == interval)

{

return 2;

}

if(2 == interval)

{

return 7;

}

if(3 == interval)

{

return 15;

}

if(4 == interval)

{

return 31;

}

if(5 == interval)

{

return 63;

}

if(6 == interval)

{

return 127;

}

if(7 == interval)

{

return 255;

}

return 0;

}

void I2C1\_init(void)

{

SYSCTL\_RCGCI2C\_R |= 0x02; // enable clock to I2C1

SYSCTL\_RCGCGPIO\_R |= 0x01; // enable clock to GPIOA

/\* PORTA 7, 6 for I2C1 \*/

GPIO\_PORTA\_AFSEL\_R |= 0xC0; /\* PORTA 7, 6 for I2C1 \*/

GPIO\_PORTA\_PCTL\_R &= ~0xFF000000; /\* PORTA 7, 6 for I2C1 \*/

GPIO\_PORTA\_PCTL\_R |= 0x33000000;

GPIO\_PORTA\_DEN\_R |= 0xC0; /\* PORTA 7, 6 as digital pins \*/

GPIO\_PORTA\_ODR\_R |= 0x80; /\* PORTA 7 as open drain \*/

I2C1\_MCR\_R = 0x10; /\* master mode \*/

I2C1\_MTPR\_R = 39; /\* 100 kHz @ 80 MHz \*/

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\* Wait until I2C master is not busy and return error code \*/

/\* If there is no error, return 0 \*/

static int I2C\_wait\_till\_done(void)

{

while(I2C1\_MCS\_R & 1); /\* wait until I2C master is not busy \*/

return I2C1\_MCS\_R & 0xE; /\* return I2C error code \*/

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\* Write one byte only \*/

/\* byte write: S-(saddr+w)-ACK-maddr-ACK-data-ACK-P \*/

char I2C1\_byteWrite(int slaveAddr, char data)

{

char error;

/\* send slave address and starting address \*/

I2C1\_MSA\_R = slaveAddr << 1;

I2C1\_MDR\_R = data;

I2C1\_MCS\_R = 7; /\* S-(saddr+w)-ACK-maddr-ACK \*/

error = I2C\_wait\_till\_done(); /\* wait until write is complete \*/

if (error) return error;

return 0; /\* no error \*/

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\* Use burst write to write multiple bytes to consecutive locations \*/

/\* burst write: S-(saddr+w)-ACK-maddr-ACK-data-ACK-data-ACK-...-data-ACK-P \*/

char I2C1\_burstWrite(int slaveAddr, int byteCount, char\* data)

{

char error;

if (byteCount <= 0)

return -1; /\* no write was performed \*/

/\* send slave address and starting address \*/

I2C1\_MSA\_R = slaveAddr << 1;

I2C1\_MDR\_R = \*data++;

I2C1\_MCS\_R = 3; /\* S-(saddr+w)-ACK-maddr-ACK \*/

byteCount--; // send first char with start & ACK

error = I2C\_wait\_till\_done(); /\* wait until write is complete \*/

if (error) return error;

/\* send remaining data one byte at a time \*/

while (byteCount > 1)

{

I2C1\_MDR\_R = \*data++; /\* write the next byte \*/

I2C1\_MCS\_R = 1; /\* -data-ACK by slave- \*/

error = I2C\_wait\_till\_done();

if (error) return error;

byteCount--;

}

/\* send last byte and a STOP \*/

I2C1\_MDR\_R = \*data++; /\* write the last byte \*/

I2C1\_MCS\_R = 5; /\* -data-ACK-P \*/

error = I2C\_wait\_till\_done();

while(I2C1\_MCS\_R & 0x40); /\* wait until bus is not busy \*/

if (error) return error;

return 0; /\* no error \*/

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\* Read memory \*/

/\* read: S-(saddr+w)-ACK-maddr-ACK-R-(saddr+r)-ACK-data-ACK-data-ACK-...-data-NACK-P \*/

char I2C1\_read(int slaveAddr, int byteCount, char\* data)

{

char error;

if (byteCount <= 0)

return -1; /\* no read was performed \*/

/\* configure bus from for read, send start with slave addr \*/

I2C1\_MSA\_R = (slaveAddr << 1) + 1; /\* restart: -R-(saddr+r)-ACK \*/

if (byteCount == 1) /\* if last byte, don't ack \*/

I2C1\_MCS\_R = 7; /\* -data-NACK-P \*/

else /\* else ack \*/

I2C1\_MCS\_R = 0xB; /\* -data-ACK- \*/

error = I2C\_wait\_till\_done();

if (error) return error;

\*data++ = I2C1\_MDR\_R; /\* store the data received \*/

if (--byteCount == 0) /\* if single byte read, done \*/

{

while(I2C1\_MCS\_R & 0x40); /\* wait until bus is not busy \*/

return 0; /\* no error \*/

}

/\* read the rest of the bytes \*/

while (byteCount > 1)

{

I2C1\_MCS\_R = 9; /\* -data-ACK- \*/

error = I2C\_wait\_till\_done();

if (error) return error;

byteCount--;

\*data++ = I2C1\_MDR\_R; /\* store data received \*/

}

I2C1\_MCS\_R = 5; /\* -data-NACK-P \*/

error = I2C\_wait\_till\_done();

\*data = I2C1\_MDR\_R; /\* store data received \*/

while(I2C1\_MCS\_R & 0x40); /\* wait until bus is not busy \*/

return 0; /\* no error \*/

}

int main(void)

{

//initializeSystemClockFor80mhz();

initializeGPIOPorts();

int currentTemperatureInFahrenheit;

char slaveAddress = SLAVE\_ADDR;

uint32\_t pw, RCC;

char \* bufferp;

int len; // string length

// char i;

char buffer[40] = {"hello world - it's a fine day "};

char \* cbuffer = {"this is a command string"}; // buffer for commands

char str[10]; // char string for viewing value

bufferp = &buffer[0];

/\* ------------ configure system clock for 80 Mhz operation -------------\*/

SYSCTL\_RCC2\_R |= 0x80000000; // Use RCC2

SYSCTL\_RCC2\_R |= 0x00000800; // Bypass PLL while initializing it

// Select crystal value and osc source

SYSCTL\_RCC\_R = (SYSCTL\_RCC\_R & ~0x000007C0) // clear bits 10-6

+ 0x00000540; // 10101 configure for 16Mhz XTL

SYSCTL\_RCC2\_R &= ~0x00000070; // Use main oscillator

SYSCTL\_RCC2\_R &= ~0x00002000; // Activate PLL - clear PWRDN

SYSCTL\_RCC2\_R |= 0x40000000; // Set system divider

SYSCTL\_RCC2\_R = (SYSCTL\_RCC2\_R & ~0x1FC00000) +(SYSDIV2<<22);

while((SYSCTL\_RIS\_R&0x00000040)==0){}; // Wait for PLL to lock - poll PLLRIS

SYSCTL\_RCC2\_R &= ~0x00000800; // Enable PLL by clearing BYPASS

SysTick\_init(); // initialize SysTick timer

init\_SSI0(); // Configure and initialize SSI1 interface

I2C1\_init(); // Configure & Initialize I2C1 interface

configureI2C();

/\* command TMP3 to read from register 0 for desired temperature format \*/

I2C1\_byteWrite(slaveAddress, i2c\_data[0]);

init\_ADC0(); // Configure and initialize ADC0

PWM0\_0\_init(); // Configure and initialize SSI1 interface

/\* preliminary display message \*/

// use the LCDS reset command sequence

SSI0Write(0x1b); // Display reset - write an escape character

cbuffer="[\*"; // command sequence for reset

putsSPI0(2, cbuffer);

SysTick\_mSecDelay(100); // 10->approximately .1 s

bufferp ="Hello World ! ";

putsSPI0(strlen(bufferp), bufferp);

SysTick\_mSecDelay(625); // approximately .625 s

SSI0Write(0x1b); // Display reset - write an escape character

cbuffer="[j"; // command sequence for clear screen

// and reset cursor

putsSPI0(2,cbuffer); // write out string

SysTick\_mSecDelay(625); // approximately .625 s

char lcdDisplayMessage[16];

while(1)

{

ADC0\_PSSI\_R |= 8; /\* start a conversion sequence on SS3 \*/

while((ADC0\_RIS\_R & 8) == 0) ; /\* wait for conversion complete \*/

result = ADC0\_SSFIFO3\_R; /\* read conversion result \*/

ADC0\_ISC\_R = 8; /\* clear completion flag \*/

sprintf(str,"%d",result ); // convert int to ASCII string

SSI0Write(0x1b); // Display reset - write an escape character

cbuffer="[j"; // command sequence for clear screen

// and reset cursor

putsSPI0(2,cbuffer); // write out string

SysTick\_mSecDelay(10); // approximately .01 s

currentTemperatureInFahrenheit = getCurrentTemperatureFromSensorInFahrenheit(slaveAddress);

bool systemIsOverheating = checkIfSystemIsOverheating(currentTemperatureInFahrenheit);

sprintf(lcdDisplayMessage,"%df",currentTemperatureInFahrenheit);

if(systemIsOverheating)

{

sprintf(lcdDisplayMessage,"reactor scram");

}

//putsSPI0(strlen(str),str); // write out string

putsSPI0(strlen(lcdDisplayMessage),lcdDisplayMessage); // write out string

//uint8\_t statusLightBar = atoi(str);

displayDecimalNumberOnGpioBoardInBinary(convertDecimalToPercentLightBar(result));

SysTick\_mSecDelay(100); // delay before reading new temperature

PWM0\_1\_CMPA\_R = 937 + (result + 200);

}

}

void PWM0\_0\_init(void)

{

/\* enable clocks \*/

SYSCTL\_RCGCGPIO\_R |= 0x2; // enable clock to GPIOB (M0PWM3 is on PB5)

SYSCTL\_RCGCPWM\_R |= 1; // enable clock to PWM0

while((SYSCTL\_PRPWM\_R & 0x1)==0) {}; // wait for PWM0 ready

// use default divider of 64 for PWM clock

/\* initialize GPIO pin - PB5 \*/

GPIO\_PORTB\_AFSEL\_R |= 0x20; // PORTB bit 5

GPIO\_PORTB\_PCTL\_R &= ~0x00F00000; // PORTB bit 5 for M0PWM1

GPIO\_PORTB\_PCTL\_R |= 0x00400000;

GPIO\_PORTB\_DIR\_R |= 0x20; // PORTB bit 5 as output

GPIO\_PORTB\_DEN\_R |= 0x20; // PORTB bit 5 as digital pins

/\* Configure PWM0 module, generator 1, pwmB \*/

// PWM0\_1B - PB5 - M0PWM3 pin - Module 0 Generator 1 - pwm1B

PWM0\_1\_CTL\_R = 0x0; // disable PWM0 - entire generator 1 block

PWM0\_1\_GENB\_R = 0x00000083; // high on ZERO, low on CMPA down

PWM0\_1\_LOAD\_R = 24999; // load = 25000 -1

PWM0\_1\_CTL\_R = 0x1; // enable PWM0 Gen 1 - count down mode

PWM0\_INVERT\_R = 0x8; // invert PWM0 Gen 1 - pwmB MOPWM3

PWM0\_ENABLE\_R = 0x08; // enable PWM0 Gen 1 - pwmB M0PWM3

}

void SysTick\_init(void){

NVIC\_ST\_CTRL\_R = 0; // 1. disable SysTick before configuring

NVIC\_ST\_RELOAD\_R = DELAY\_VALUE; // 2. set to desired delay value (1 mSec)

NVIC\_ST\_CURRENT\_R = 0; // 3. clear CURRENT by writing any value

NVIC\_ST\_CTRL\_R &= ~0x00000004; // 4. set clock to POSC/4

NVIC\_ST\_CTRL\_R |= 0x00000001; // 5. enable SysTick timer

}

/\* 1 mSec Time delay using busy wait. \*/

void SysTick\_mSecDelay(uint32\_t delay){

uint32\_t i;

for(i=0; i<delay; i++){

NVIC\_ST\_RELOAD\_R = DELAY\_VALUE; // wait one cycle = DELAY\_VALUE

NVIC\_ST\_CURRENT\_R = 0;

while((NVIC\_ST\_CTRL\_R & 0x00010000) == 0){ };

}

}

void init\_ADC0(void)

{

/\* enable clocks \*/

SYSCTL\_RCGCGPIO\_R |= 0x10; /\* enable clock to GPIOE (AIN9 is on PE4) \*/

SYSCTL\_RCGCADC\_R |= 1; /\* enable clock to ADC0 \*/

/\* initialize PE4 for AIN0 input \*/

GPIO\_PORTE\_AFSEL\_R |= 0x10; /\* enable alternate function \*/

GPIO\_PORTE\_DEN\_R &= ~0x10; /\* disable digital function \*/

GPIO\_PORTE\_AMSEL\_R |= 0x10; /\* enable analog function \*/

/\* initialize ADC0 \*/

ADC0\_ACTSS\_R &= ~8; /\* disable SS3 during configuration \*/

ADC0\_EMUX\_R &= ~0xF000; /\* software trigger conversion \*/

ADC0\_SSMUX3\_R = 9; /\* get input from channel 9 \*/

ADC0\_SSCTL3\_R |= 6; /\* take one sample at a time, set flag at 1st sample \*/

ADC0\_ACTSS\_R |= 8; /\* enable ADC0 sequencer 3 \*/

}

/\* This function writes one byte to a slave device via the SSI0 interface \*/

void SSI0Write(unsigned char data)

{

while((SSI0\_SR\_R & 2) == 0); // wait until FIFO not full

SSI0\_DR\_R = data; // transmit high byte

while(SSI0\_SR\_R & 0x10); // wait until transmit complete

}

/\* --------------------------------------------------------------------------- \*/

/\* This function writes the characters in a string to the SPI 0 interface \*/

/\* The input arguments are a character count and the start address of the buffer \*/

/\* As SS01Write() waits for the FIFO buffer to not be full, no waiting is \*/

/\* Needed in this routine. \*/

/\* --------------------------------------------------------------------------- \*/

void putsSPI0(size\_t buflen, char \* buffer) {

char \* i;

for (i = buffer; i < (buffer + buflen); i++)

{

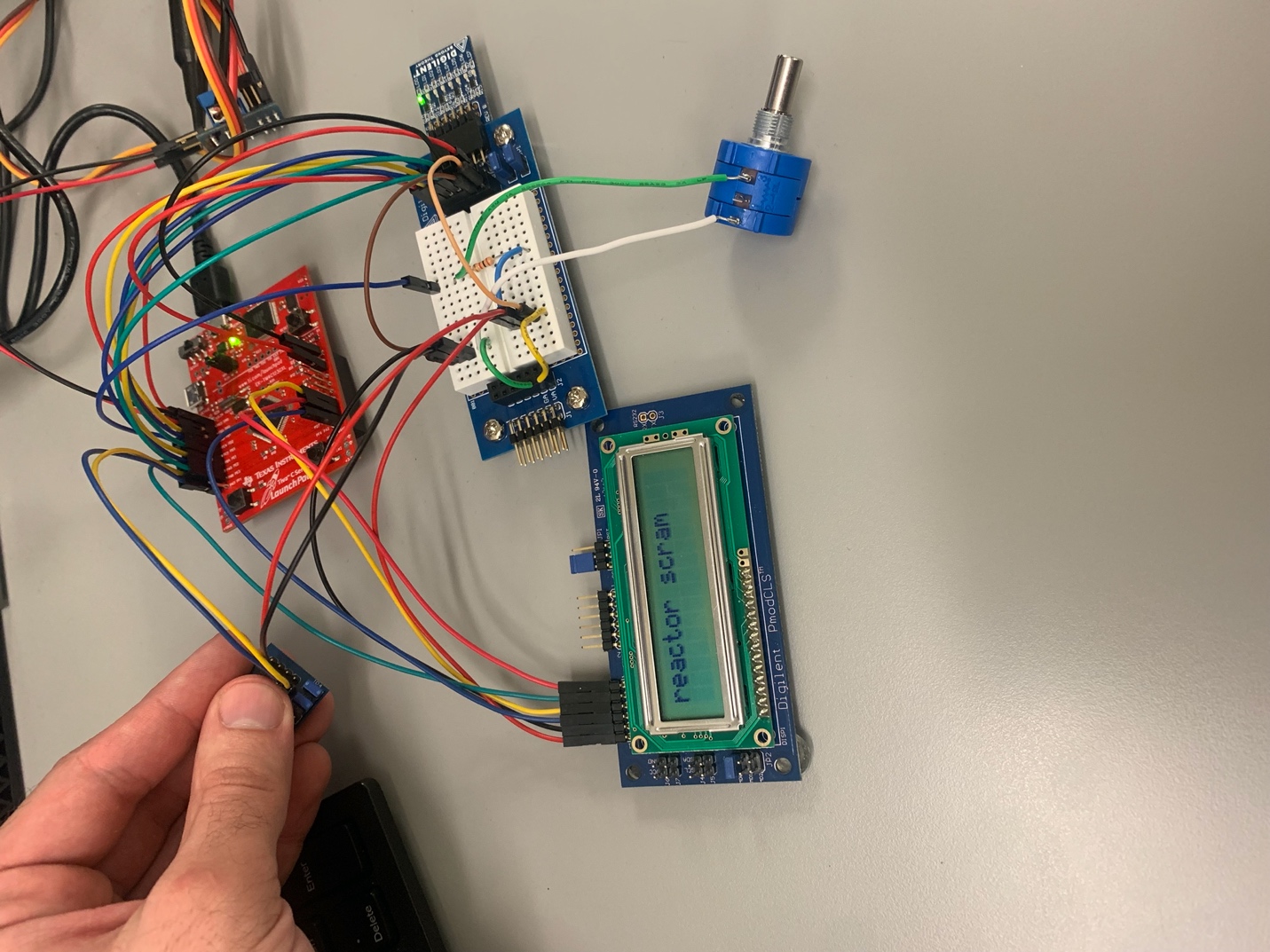
SSI0Write(\*i); /\* write a character \*/

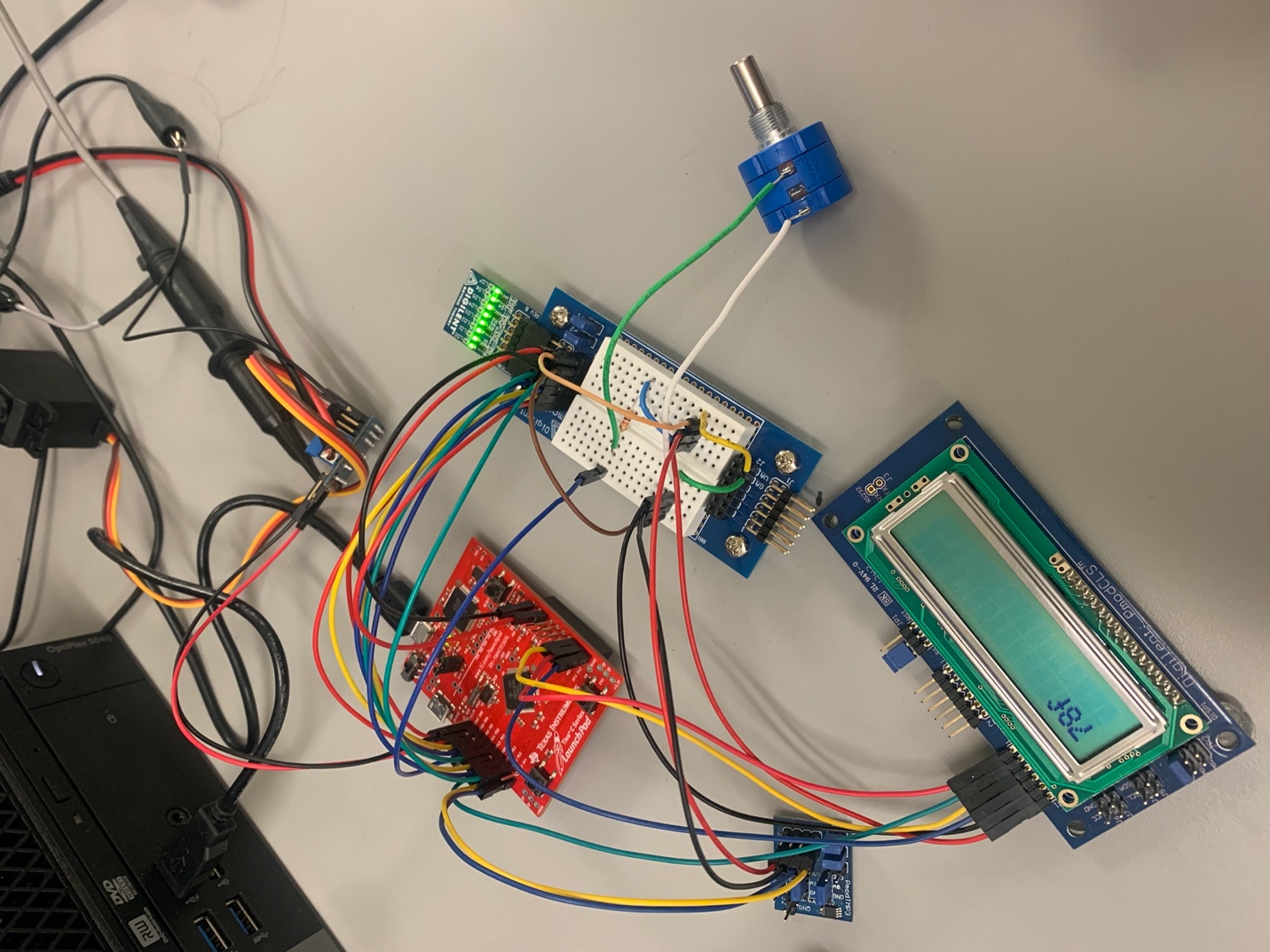
}

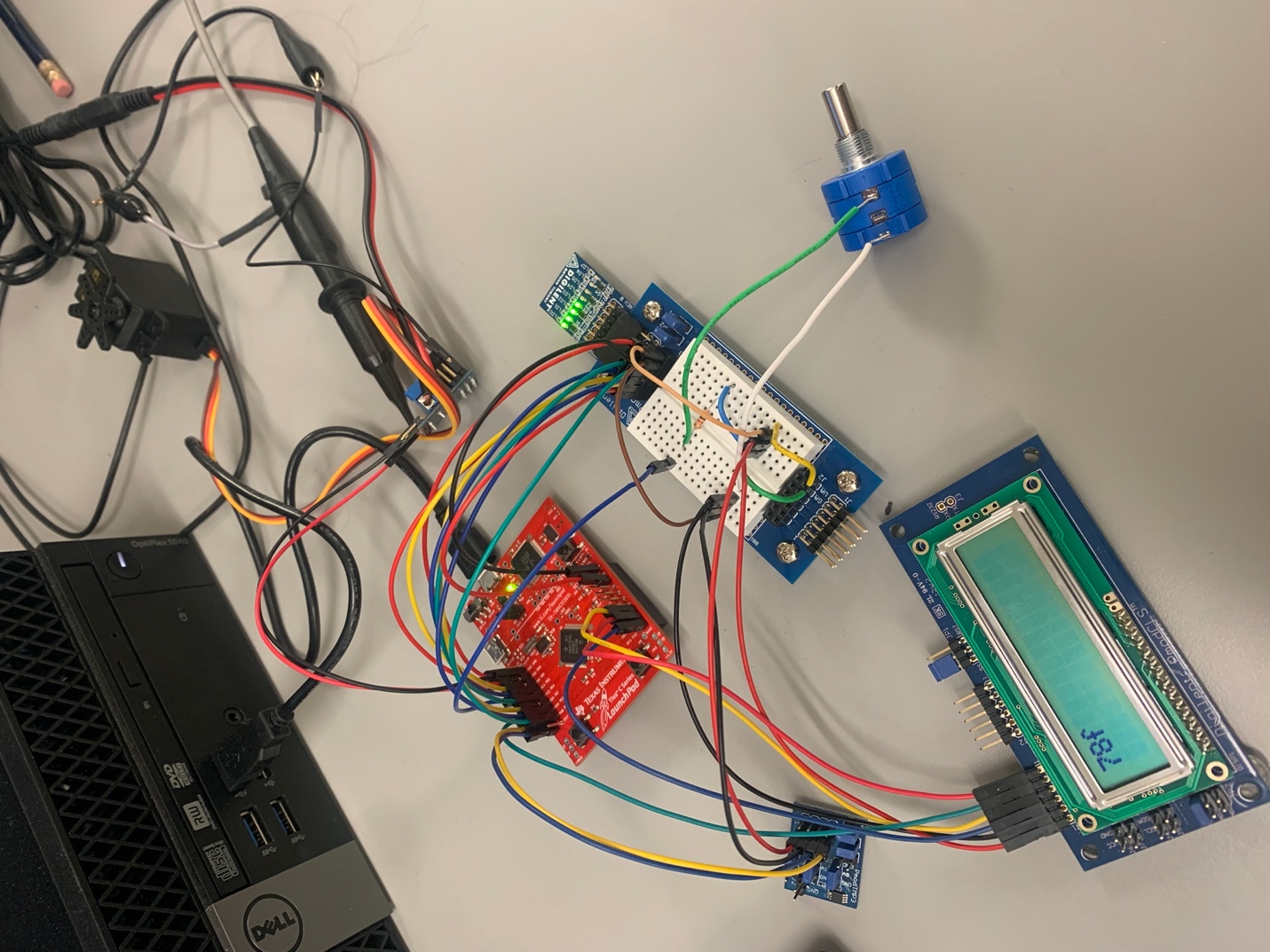
}

**Testing:**

We tested the control system with inspection--checking that the three devices adjusted accordingly with each other, and the program detected a scram when the temperature reached a threshold. The results are shown below:

****

****

****

## Conclusion:

Pulse width modulation is useful for controling servo motors. When using pulse width modulation, you need to be careful and check that the input signal will not force the servo beyond its limits.