Shyavan Sridhar, 400388748
Lab Partner: Stephan Foster, 400384917

April 14, 2024

# Developing a PID Temperature Controller

Design Project – PID Temperature Controller | Lab Instructor: Mohammadreza Shahzadeh

ENGPHYS 3BB3 | McMaster University

Shyavan Sridhar, 400388748 April 14, 2024
Lab Partner: Stephan Foster, 400384917

# Contents

Shyavan Sridhar, 400388748 April 14, 2024
Lab Partner: Stephan Foster, 400384917

## Abstract

This project took a look at developing a PID temperature controller using the MSP430 microcontroller and a thermoelectric cooler (TEC) module as an exercise in demonstrating knowledge gained over the semester and as a valuable application of PID. This was done as a culminating assignment for the course to showcase the content learned and apply it to a physical system. The system designed in the end was accurate to the measured temperature by a Fluke thermocouple to within 0.0 – 0.3°C at most and the PID system had a typical time constant for moderate temperature changes of about 10 seconds, settling on the setpoint to within 0.1 °C during steady-state. Factoring the uncertainty of the measurement tool, the entire system can be said to be accurate to ≤ 0.6°C. All of these results indicate that the system was extremely well tuned and calibrated to achieve the full marks on the demo in addition to performing to satisfaction in general.

## Introduction

The objective of this project was to use all the concepts learned throughout the course in relation to programming an MSP430 microcontroller and implement it in a PID temperature controller using the MSP430 and view data through a graphical user interface (GUI) in MATLAB. As such, the experiment's purpose was to showcase understanding of the knowledge taught and also explore what is required to construct a PID controller. Moreover, the understanding gained from this project can be applied in several projects outside the course due to the multifaceted nature of the project, combining several tasks into a project that has a real-life use and could be adapted for various situations as the PID control could be used for mechanical systems such as a gyroscopic balancing mechanism and the system as-built here could be used as a thermostatic control device for small applications.

Shyavan Sridhar, 400388748
Lab Partner: Stephan Foster, 400384917
April 14, 2024

# Theory

## Thermistors

The thermistor works by varying its electrical resistance with small changes in temperature [1]. Specifically, there are two main types of thermistors being positive and negative temperature coefficient thermistors (PTC and NTC) which increase resistance and decrease resistance with an increase in temperature respectively [1]. In this project, an NTC thermistor from muRata was used [2]. In addition, as thermistors respond extremely nonlinearly, they tend to be quite sensitive to temperature changes, but on a limited scale, the relation can be simplified to a third-order approximation provided below [1].

$$\frac{1}{T} = a + b\ln R + c(\ln R)^3 \tag{1}$$

Where T is the temperature read by the thermistor and R is the resistance of the thermistor while a, b, and c are device-specific constants determined by calibration [1]. The thermistor can be placed in series with a fixed resistor in order to construct a voltage divider from which the change in resistance would be equated to a change in voltage as was done for the sensor in this project a diagram of which is shown below in *Figure 1* [3].

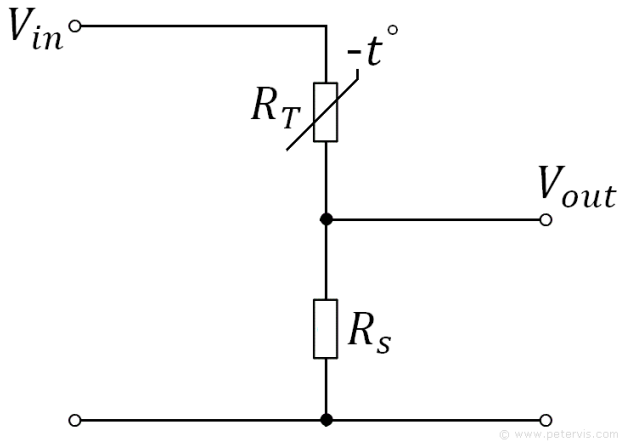


*Figure 1 – Circuit schematic of Thermistor in a Voltage Divider Configuration* [3].

## Peltier Thermoelectric Cooler (TEC)

A Peltier module, thermoelectric cooler, or TEC is a solid-state device that uses the Peltier effect to cool or heat (depending on the side used) various systems [4]. The effect is such that a current passing through two semiconductors with different dopings (n-type and p-type) results in the junction between the two absorbing or releasing heat depending on the direction of the current flowing through them [4]. Thus, in the TEC, there are several of these semiconductor junctions to make the two large sides act as either a cold or hot side [4]. The greater the temperature difference between the two, the greater the power the device can dissipate and as such, one side is often cooled with a heatsink and fan assembly as was done in this experiment [4]. *Figure 2* shows a diagram of these devices [4].

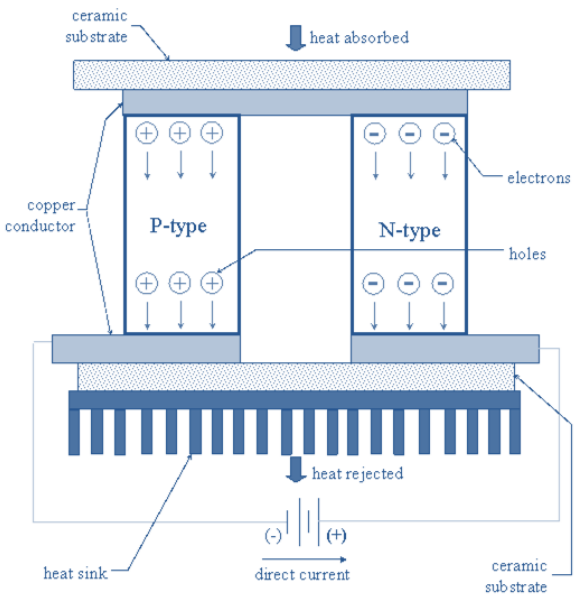


*Figure 2 – Diagram of TEC Construction* [4].

## H-Bridges

To drive the TEC at varying voltages and thus control the amount of heating or cooling applied, an AC, controllable, and comparatively high voltage and high current supply is required. For this purpose, a DC supply can be connected to an H-Bridge and then modulated in magnitude and direction by signals sent by the MSP430 microcontroller. The H-Bridge is a switching device capable of performing this task using 4 field effect transistors (FET) [5]. FETs are outside the scope of this report and are not required reading material and hence, it is left to the reader to read further if interested. The end result of configuring these 4 FETs in a certain manner is a switching power supply that is capable of powering the TEC and is also often used for other inductive AC or DC loads such as motors [5]. Turning on two of the FETs at a time in an H-Bridge configuration results in current flowing in a particular direction and the rate at which they are switched results in variability in the voltage magnitude output in addition to the direction of the current flow [5]. For this experiment the H-Bridge integrated circuit (IC) used is the L298N [2]. A schematic of what this device looks like is presented below in *Figure 3* [5].

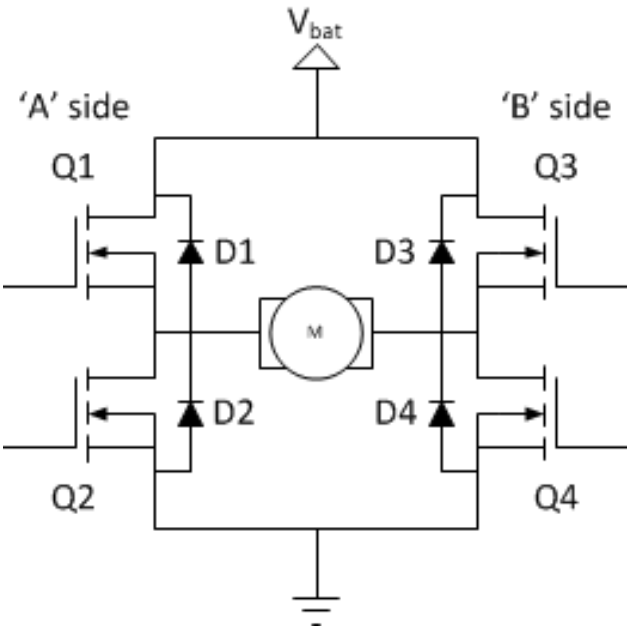


*Figure 3 – Schematic of H-Bridge IC* [5].

## Pulse-Width Modulation (PWM)

As the MSP430 microcontroller cannot directly output an analog signal, an equivalent signal can be achieved using a concept called pulse width modulation (PWM). Using the timer module available on the MSP430, a pin can be triggered high and low at specific time intervals to produce a square wave of varying duty cycle [6]. This is all done by rapidly switching between the high and low voltages at a set frequency to produce a square wave with varying high and low times [7]. The time the signal is high in relation to the time the signal is low is proportional to the average voltage the signal is seen as [7]. By varying this to different values, the average voltage can thus be adjusted, and an analog equivalent signal can be produced [7]. This can be used to drive the L298N H-Bridge by rapidly switching its inputs. By applying a PWM signal on one input, the current flows in one direction through the H-Bridge and so the system outputs voltage proportional to the duty cycle of the input. When the other input is pulsed, the direction of current flow is flipped and as such, this can be used to control which side of the TEC is heating or cooling at any given time to precisely land on a temperature. Examples of PWM signals are shown below in *Figure 4* [7].

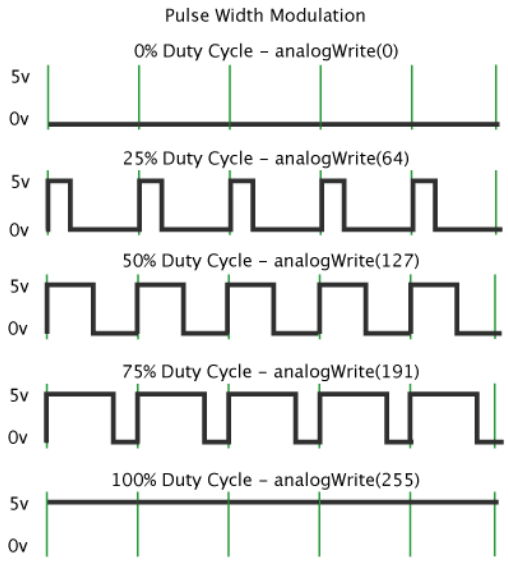


*Figure 4 – Timing Diagram for varying PWM Signals* [7]

## Proportional-Integral-Derivative (PID) Control

In order to accurately, quickly, and stably settle on a setpoint, some method of control must be applied to heat or cool the system. For this project, a PID controller was chosen. This controller is a form of closed-loop feedback control that allows for the system to be adjusted according to an error between the setpoint and the current value [8]. This is done with three coefficients applied to each of the forms of error following [8]. The first is proportional error which is simply a term that makes the system responsive to the magnitude of the error, responding quickly to large swings in temperature, and less so for very small differences [8]. To better respond in the small difference region, the system uses an integral error term which is the sum of the errors [8]. This allows the system to compensate for steady-state error as the sum of the error over time will grow and thus, the controller will apply a change to reduce this sum [8]. Consequently, it also tends to increase oscillation in the system around the setpoint [8]. Finally, there is the derivative error term which is the difference between the last two errors over time [8]. This term as a result, tends to dampen out oscillations in the system by reacting to reduce the variance in the error over time [8]. A flow diagram of how this style of controller can be constructed is shown below in *Figure 5* [8].

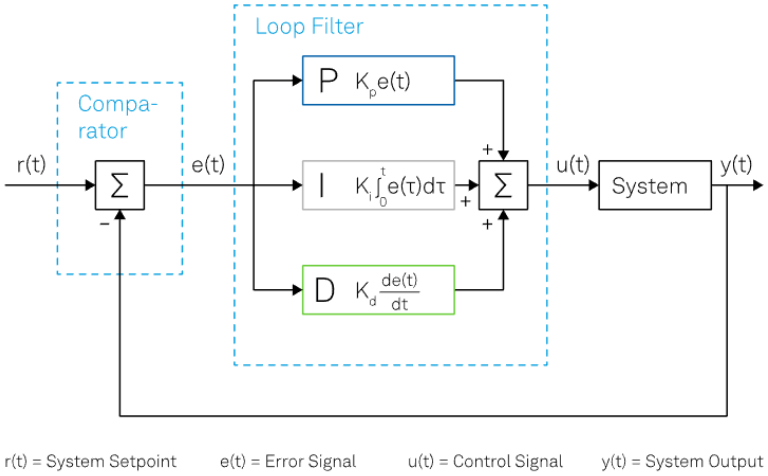


*Figure 5 – Timing Diagram for varying PWM Signals* [7]

## Serial Communications

As this project requires communication between the GUI running on MATLAB on a PC and the MSP430 microcontroller, some level of serial communication had to be employed between the two. This project involved using both the Universal Asynchronous Receiver/Transmitter (UART) serial communication protocol and the programming concept of interrupts. UART is a two-line protocol that allows devices to talk to each other over a dedicated transmit and receive data line [9]. These lines are pulled low first by the device requesting to transmit to the other device's receive port and then 8 data bits are sent in a single packet followed by a stop bit that indicates the end of the transmission [9]. Each device as such, has its transmit port connected to the receive port of the other device and as such, the two lines only allow for one-way communication each [9]. Both devices are also required to be set to the same communication frequency or Baud rate [9]. In addition, interrupts are a method of limiting communication by only performing a certain action upon another action [6]. In this case, the action of sending data from MATLAB to the MSP430 triggers an interrupt which results in the microcontroller adjusting its current heating or cooling output signal and sending back the current temperature data to MATLAB. This reduces issues with latency during the communication period and help keep both devices in-sync to prevent any dropped packets while also being more resource efficient [6].

Shyavan Sridhar, 400388748 April 14, 2024
Lab Partner: Stephan Foster, 400384917

## Analog-to-Digital and Digital-to-Analog Converters (ADC & DAC)

To deal with external variable signals through a microcontroller, there must be some conversion between analog and digital values [10]. Often, motors and sensors will require analog outputs and inputs in terms of a continuously variable voltage signal respectively. However, a microcontroller will typically be dealing in digital quantities, assigning each pin to be high or low. As such, it is necessary to convert between the two signal types by quantizing an analog signal into discrete steps that allow it to be represented by a set of bits digitally [10]. This digital signal can then be converted to an analog one and vice versa by identifying the corresponding value in the opposite signal type [10]. For example, with 8 bits and a range of 5 V, 0 would be 0 V, 128 would be 2.5 V, and 255 would be 5 V [10]. Thus, for receiving input from analog sensors, an ADC would be extremely valuable, while a DAC would help with outputting an analog voltage to an actuator.

It is also important that the analog signal is sampled at a high enough frequency to re-create it correctly in an ADC [10]. This is because if it is sampled at too low a rate, then the signal would appear to have a lower frequency than it has [10]. Thus, it is recommended that the signal is sampled at a rate above twice its frequency, a value known as the Nyquist rate [10]. In this project, the MSP430 microcontroller specifically uses the successive approximation register (SAR) ADC algorithm to generate a 10-bit value via the ADC10 module which performs all the sampling processes behind the scenes apart from some parameters which are set in the code. The SAR algorithm is a highly resource-efficient method of conversion and is suited for low-power applications such as what the MSP430 microcontroller is intended for [10]. This method involves beginning with a guess of the analog voltage at half the total range available and setting a step size to half of that value [10]. This guess is then used to generate an analog signal which is compared to the original [10]. If it is greater, the signal is reduced by the step size and vice versa and the step size is then halved [10]. This repeats until the step size is no longer an integer and the whole cycle is repeated starting at the initial guess again [10]. As a result, it reaches the intended value faster and with limited resources while also being stable at that point until it is reset again [10].

# Methods

## Physical Setup

To create the physical system that would adjust the temperature, 3 components were used. A pre-made module containing the L298N IC, TEC, heatsink, fan, and thermistor affixed via thermal paste and epoxy was provided with jumper wires to be connected, this will be referred to as the apparatus from here on out for ease. The other two components used were the MSP430 itself and a 10 kΩ resistor to be the constant resistance within the voltage divider, the other half of which is the thermistor within the apparatus. These were then connected over a breadboard with the supply to the fan and H-Bridge on the apparatus set to DC 12 V. The 3.3 V rail from the MSP430 was then used as the input to the voltage divider and the TX and RX pins (P1.1 and P1.2) of the microcontroller were connected to a UART-USB bridge which was input to the laptop running the MATLAB code. The MSP430 received power from the laptop over USB as well. P1.4 of the MSP430 was used as the ADC input with a cable connected to the point between the thermistor and resistor as a voltage divider. The fixed resistor was on the high side of the setup and the thermistor connected to ground so that the voltage measured would be the drop over the thermistor directly. P1.6 and P2.6 were used as the PWM outputs of the microcontroller to drive the H-Bridge. Lastly, both the grounds of the microcontroller, laptop power, and 12 V supply were connected together to prevent any floating signals and reduce noise. For writing the MATLAB code, the fixed resistance was also measured using a multimeter to be 9.99 ± 0.02 kΩ with the calculation provided in the appendices and the final setup shown below in *Figure 6*.

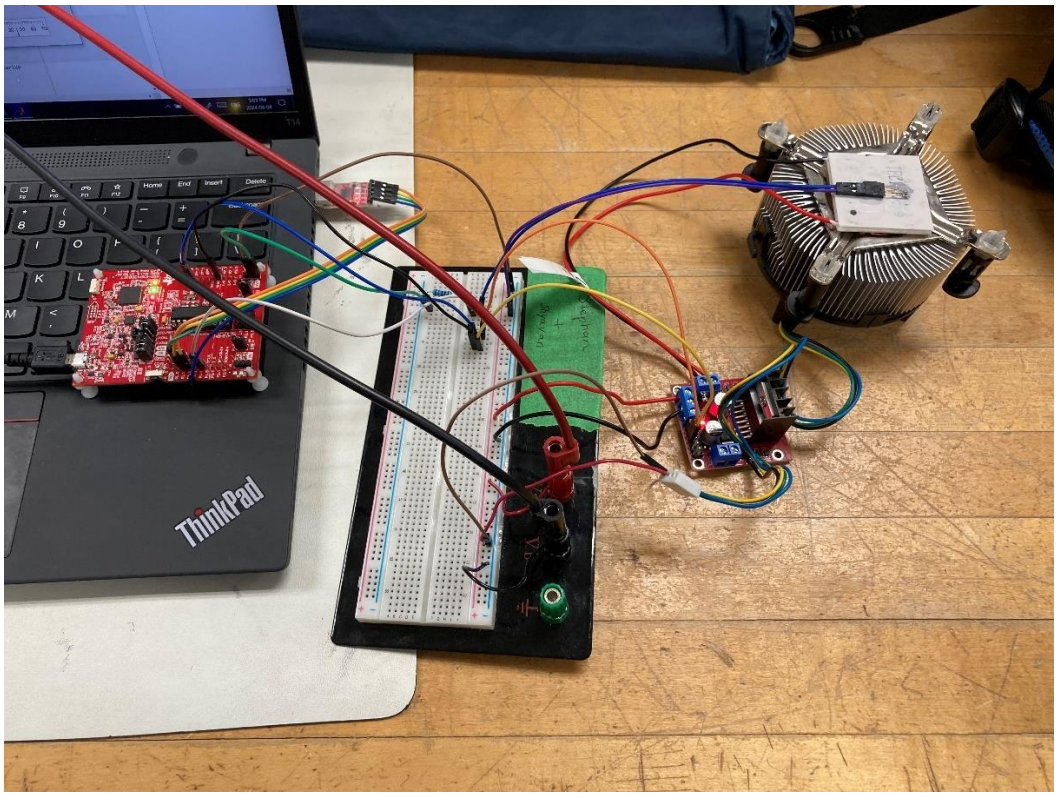


*Figure 6 – Final Setup of System*

It should also be noted that this setup was built progressively, building up in modules. First, the H-Bridge was hooked up directly to a function generator and it was confirmed that varying PWM signals would cause the TEC to heat or cool. Then the ADC and serial communication was set up and confirmed to read temperatures from the thermistor somewhat accurately and output it to MATLAB correctly. Finally, the PWM signals were connected to the microcontroller with the function generator removed and once it was confirmed that this still let the system heat and cool, the PID control was implemented.

## Software

The software was also created progressively, with temperature calibration done using a calibrated thermocouple along the way to attain better accuracy in readings. When writing the code uploaded to the MSP430 in C, the system clock was set to 16 MHz and the Baud rate to 115200 for better performance and response of the system and averaging was performed on several consecutive datapoints which for all intents and purposes were considered taken at the same time to increase resolution of the system. This setup removes the limitation of 8-bit precision due to the size of the UART packets sent while also mitigating any latency issues that might arise from sending a single datapoint split over multiple packets to go beyond an 8-bit precision while also reducing the noise seen by the ADC. Interrupts were also implemented to better synchronize the two devices and ensure that only MATLAB could initiate a transmission and consequently, a command to change the heating or cooling output upon which the temperature data is sent from the MSP430 to MATLAB where it is displayed on a scrolling display that shows the last 100 datapoints alongside the setpoint and used in the PID controller. The MSP430 thusly is constantly sampling, storing 30 datapoints in a buffer value, then copying them all at once to an array that is given to MATLAB as requested over 30 UART packets.

The voltage readings were averaged and converted to resistance and then to temperature in MATLAB using the calibrated data provided for the thermistor fit to a curve in Excel. However, after the PID controller was tuned, it was found that this curve was slightly off due to inconsistencies in each system and so, a linear offset was added which was determined by using a calibrated thermocouple and treating its measurements as constant with no tolerance and identifying the difference between setpoint and the actual temperature at several points. Both of these curves are shown in the Results section along with some further information. Finally, to tune the PID, the standard manual method was employed of starting with all coefficients zeroed, increasing the proportional coefficient until oscillation was observed, increasing the integral coefficient until the steady-state error was removed, and increasing the derivative coefficient until the oscillations were dampened out and a reasonable time constant of the system was achieved. The duty cycle sent back as a result of this process was represented as an 8-bit signed integer with negative values telling the microcontroller to cool and positive values indicating heating, with zero being a “do nothing” point. The PID output was also weighted and clamped such that it would always be in the bounds of an 8-bit signed integer in addition to having all the temperature errors normalized over a delta of 45°C which was deemed as the maximum possible error in the system if it was asked to shoot from the maximum of 45°C down to the minimum of

5°C and building in an overshoot tolerance of 5°C. Moreover, the integral error was also weighted to be a sum of the cube-root of the error as this was found to reduce the effect of large errors, while still allowing small errors to add up, reducing the time required to work off the sum and approach a steady state value. This allowed the integral coefficient to be increased with limited oscillation and improved the stability of the system. The integral error was also reset upon each setpoint change to enable the system to quickly respond to changes. In addition, MATLAB was made to only request temperature data and update the duty cycle from the microcontroller every 0.1 s as this was deemed a reasonable timeframe to balance latency, the amount of data processed, and the responsiveness of the system as the temperature of the TEC has physical limitations as to how fast it can change anyway. A flowchart of the code on both sides is shown in *Figures 7 and 8* while the actual C code and MATLAB App Designer code are provided in the appendices to better understand the final design that was come up with. Note that the dashed lines indicate simultaneous operations.

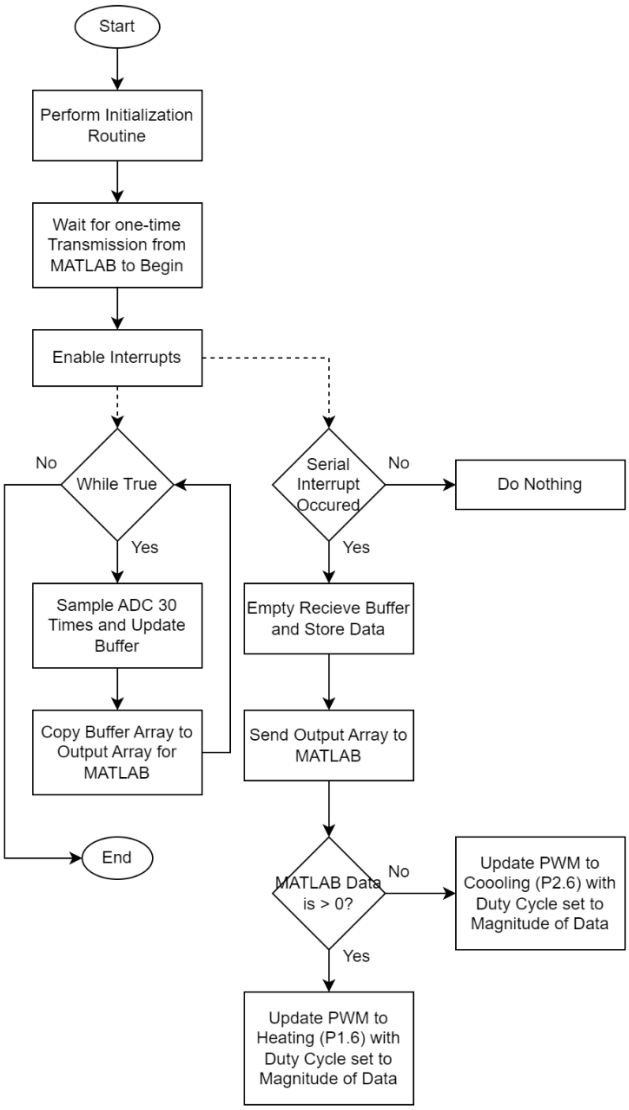


*Figure 7 – Flowchart for MSP430 C Code*

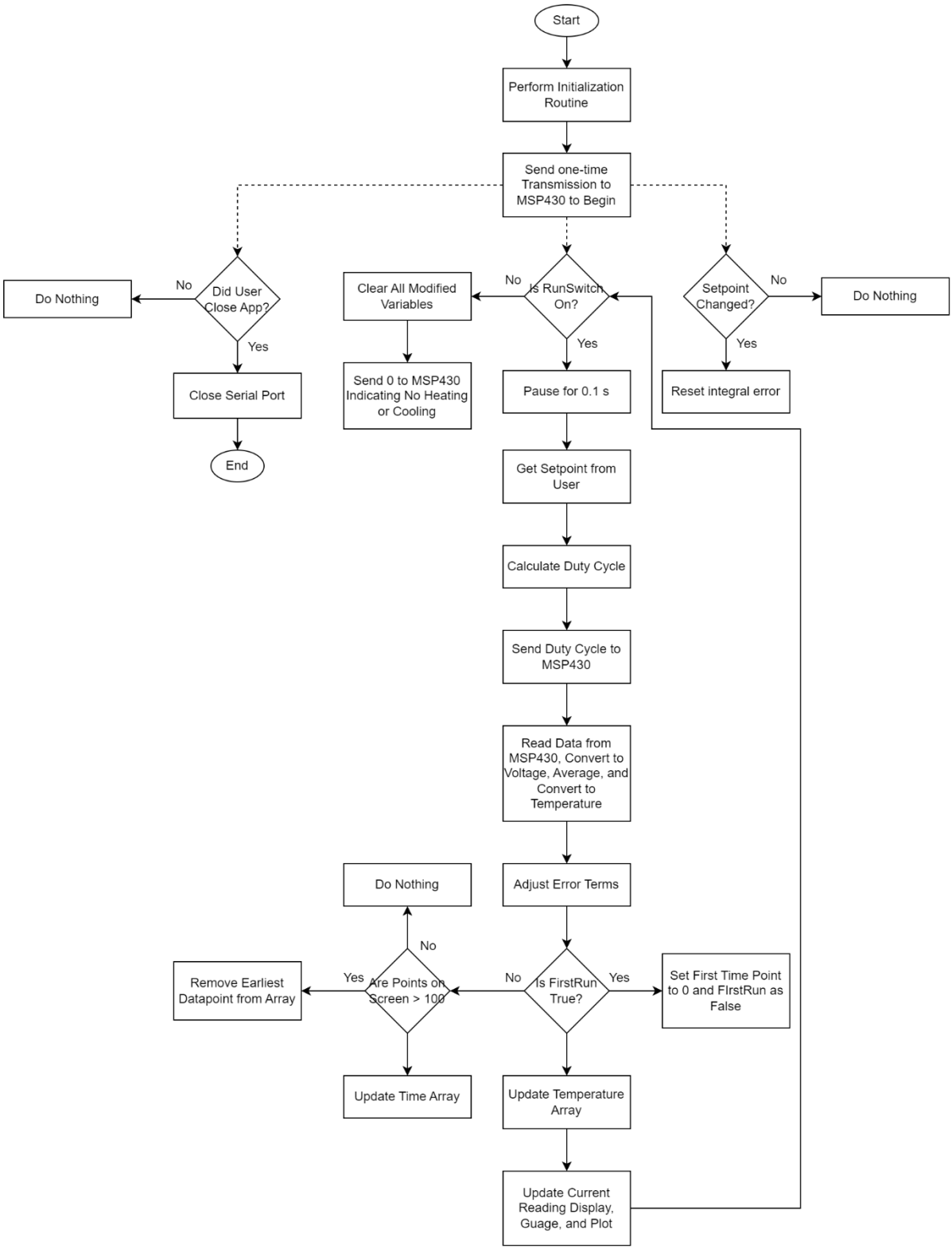


*Figure 8 – Flowchart for MATLAB GUI Code*

Shyavan Sridhar, 400388748
Lab Partner: Stephan Foster, 400384917


## Results

The system untuned, resulted in the following temperature difference with error bars in *Figure 9* taken from the manufacturer of the Fluke thermocouple used in measurement. The calculations and raw data are provided in the appendices. Moreover, the manufacturer provided data for the thermistor is fit to a curve shown in *Figure 10*.

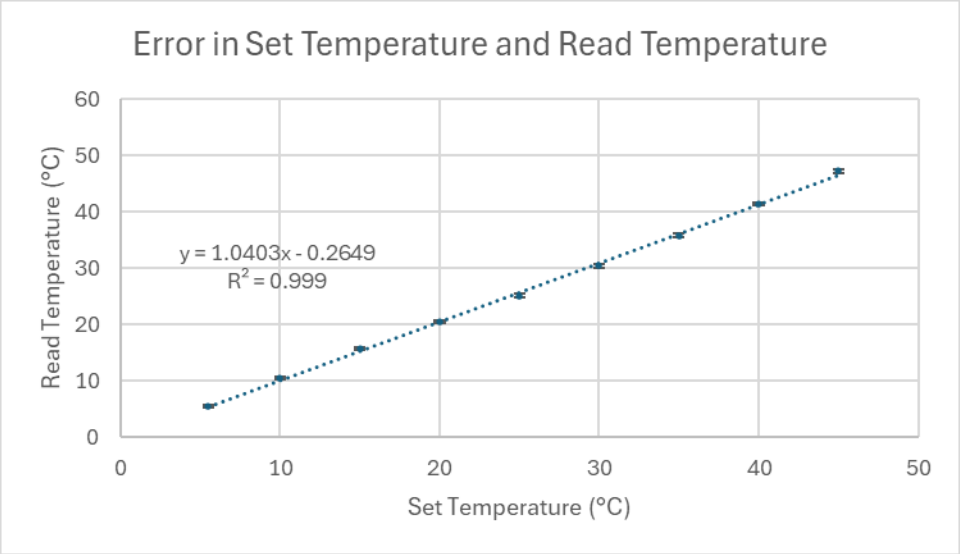


*Figure 9 – Error in Set and Read Temperatures*

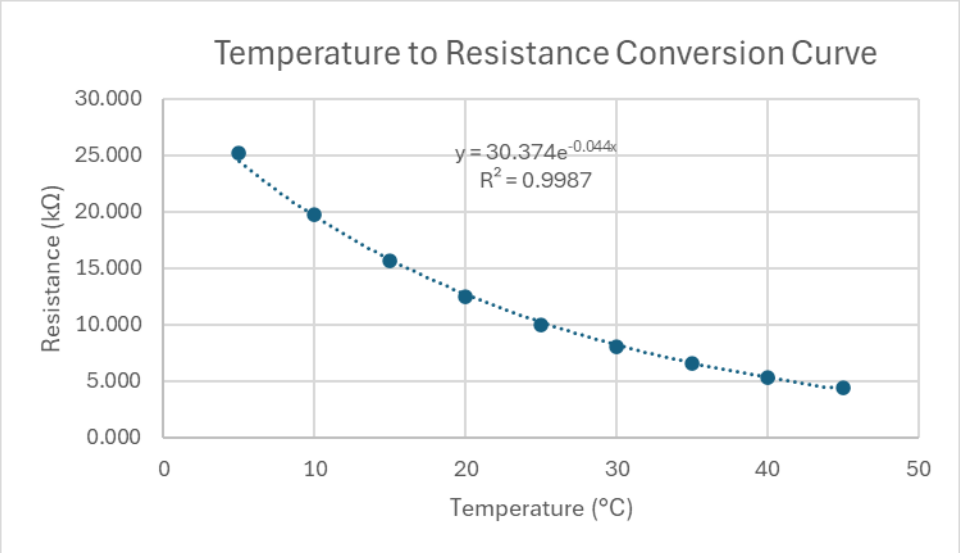


*Figure 10 – Temperature to Resistance Curve from Manufacturer Data*

After tuning the system, the PID coefficients to final values of 15.0 for the proportional error, 0.2 for the integral error, and 5.5 for the derivative error. This system was found to be extremely stable, getting to most temperatures within 10 seconds and the largest temperature change across the board to be within 30 seconds. The temperature was also found to be extremely accurate using a Fluke thermocouple to within 0.0°C – 0.3°C of the reading displayed

by the thermocouple depending on the temperature range. Due to the nature of the system, it would be difficult to accurately determine its exact uncertainty theoretically, and so using the Fluke thermocouple and the manufacturer provided uncertainty, it can be experimentally determined that that at most, the uncertainty in the system is at most ±0.6°C. Calculations for this are listed in the appendices. Pictures of the final system working as intended are shown in *Figures 11-14.*


*Figure 11 – System Performance Image 1*

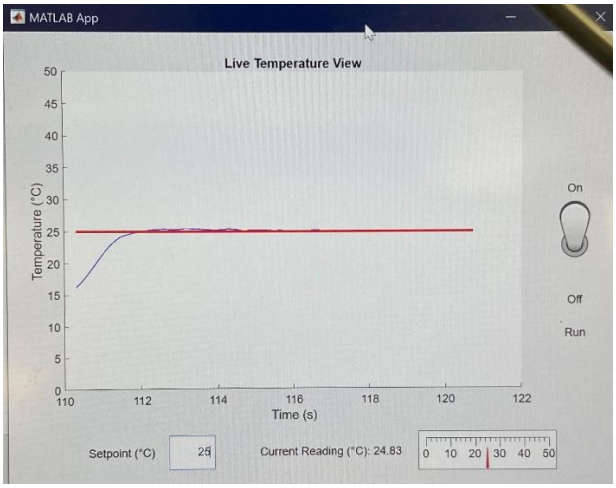


*Figure 12 – System Performance Image 2*

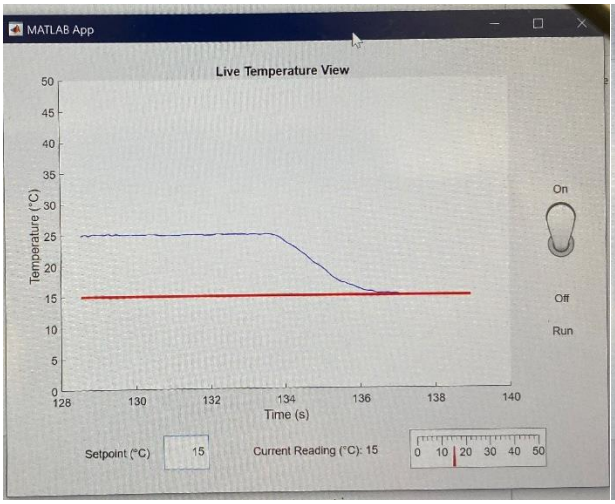


*Figure 13 – System Performance Image 3*


*Figure 14 – System Performance Image 4*

Shyavan Sridhar, 400388748 April 14, 2024
Lab Partner: Stephan Foster, 400384917

## Discussion

Through the results obtained, it is evident that the system worked as intended and had the desired accuracy and consistency with minimal noise. All the processing done had reason behind it and affected the system in a visible way that it was deemed useful to implement. However, there may be a couple areas for improvement. Firstly, this system was quite well made, however, due to the nature of the setup being in a room with ambient temperature and not in a vacuum, there is more effort required from the system to go further in either direction beyond ambient than it is for temperatures closer to ambient resulting in better performance of the PID control around room temperature than otherwise. Thus, a second thermistor should be used to measure the ambient temperature and correct for the magnitude of the temperature by adjusting the PID coefficients proportional to the difference between the setpoint and the ambient temperature measured. Moreover, the measurement instruments used all at best had an accuracy in temperature greater than what was required for the bonus of 0.2°C accuracy. Thus, even if temperatures were observed to be within that value, when considering uncertainty in the measurement device, this value could realistically be even still off from that value by more. Thus, a more accurate measurement device should have been used. This was the main reason why IR thermometers were almost immediately eliminated from the measurement repertoire as they proved to be extremely inaccurate, and variable based on the distance they were held at. Thus, the Fluke thermocouple which was the most accurate instrument available was used. Lastly, the ADC, while a 10-bit module, had to have its precision truncated to 8-bits due to the nature of UART. While this was somewhat compensated for via averaging and rounding the datapoints to ensure they fit within the 8-bit precision available, it would have been more ideal to use the full 10 bits available to the MSP430, and so, it would be more ideal to have had a system to send data over multiple packets in UART given more time. All of these would have further improved the performance of the system; however, it was quite accurate even so and deemed a success despite all of this.

## Conclusion

The results indicate that the system was indeed a success with it operating as expected. While there was room for improvement, this system has been tuned to about the maximum of what is possible given the tools available. Thus, the experiment certainly fulfilled its purpose of showcasing the knowledge obtained and acting as a valuable demonstration of PID control. In addition, the skills and understanding gained through this experiment will be invaluable for future projects such as the capstone for the future.

Shyavan Sridhar, 400388748 April 14, 2024
Lab Partner: Stephan Foster, 400384917

# Appendices

## MSP430 C Code

```
// Design Project - PID Temperature Controller

/*
A temperature controller program that interfaces with a GUI interface built
in
MATLAB over UART to adjust the temperature of a Peltier module to a setpoint.
It uses PID to reach the setpoint quickly and relies on the L298N H-Bridge IC
for driving the module, with the code running on an MSP430G2553
Microcontroller
from the TI Launchpad Development Kit.
*/

#include <msp430.h>
#include <string.h>

// Constant definitions
#define RXD (BIT1)
#define TXD (BIT2)
#define ADC (BIT4)
#define PWM_1 (BIT6)
#define PWM_2 (BIT7)
#define vPOINTS 30

/**
 * main.c
 */

unsigned volatile char tecVoltage[vPOINTS]; // Array of voltage points read
from thermistor as 8-bit value
unsigned volatile char bufferVoltage[vPOINTS]; // Buffer array of voltage
points read from thermistor as 8-bit value
signed volatile char PIDOutput; // 8-bit signed output from PID algorithm
implemented in MATLAB

unsigned int resetTime = 1; // Pulse period in milliseconds

/*
Description: MSP430 Hardware Initialization Subroutine
Inputs:      void
Outputs:     void
Parameters:  void
Returns:     void
*/
```

```
void initMSP(void)
{
	WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer

	// Set clock to 16 MHz
	BCSCTL1 = CALBC1_16MHZ;
	DCOCTL = CALDCO_16MHZ;
}

/*
Description: ADC10 Module Initialization Subroutine
Inputs:	void
Outputs:	void
Parameters:	void
Returns:	void
*/
void initADC(void)
{
	ADC10CTL1 |= INCH_4 | CONSEQ_2; // Use CONSEQ_2 (Mode 2 - Repeat single
channel) and set ADC10 on P1.4
	ADC10AE0 |= ADC; // Enable ADC10 on P1.4
	ADC10CTL0 |= ADC10SHT_0 | MSC | ADC10ON | ADC10SC | ENC; // Select 4
ADC10CLK cycles sample-and-hold time, multiple sample and conversion (MSC),
turn on ADC10, start and enable conversion
}

/*
Description: UART Serial Interface Initialization Subroutine
Inputs:	void
Outputs:	void
Parameters:	void
Returns:	void
*/
void initUART(void)
{
	// initialize the USCI to RXD on P1.1 & TXD on P1.2
	P1SEL |= RXD | TXD;
	P1SEL2 |= RXD | TXD;

	// Set communication data rate to 115200 baud
	UCA0BR1 = 0;
	UCA0BR0 = 138;

	UCA0CTL1 |= UCSSEL_3; // Set clock to SMCLK
	UCA0CTL1 &= ~UCSWRST; // Release UART RESET
```

```
	IE2 |= UCA0RXIE; // Set UCA0RXIE high to enable interrupt request of
UCA0RXIFG
}

/*
Description: PWM Timer Initialization Subroutine
Inputs:	void
Outputs:	void
Parameters:	void
Returns:	void
*/
void initPWM(void)
{
	P1DIR |= PWM_1; // P1.6 is output (Heating)
	P2DIR |= PWM_1; // P2.6 is output (Cooling)
	P1OUT = 0; // Set all outputs to 0
	P2OUT = 0; // Set all outputs to 0

	TACCTL1 |= OUTMOD_3; // Set PWM mode to set/reset

	TACTL |= TASSEL_2 | ID_2 | MC_1; // Set Timer A to SMCLK, scale factor
2, count up to TACCR0
	TACCR0 = resetTime * 4000 - 1; // Configured for 16 MHz & scale factor
2
}

/*
Description: Get 8-bit Data over UART
Inputs:	(unsigned char)UCA0RXBUF
Outputs:	void
Parameters:	void
Returns:	(unsigned char)data
*/
unsigned char getData(void)
{
	signed char data = UCA0RXBUF; // Fetch data as 8-bit char

	return data;
}

/*
Description: Send 30 8-bit Datapoints over UART
Inputs:	(unsigned int[])tecVoltage
Outputs:	(unsigned char)UCA0TXBUF
Parameters:	void
Returns:	void
```

```
*/
void sendData(void)
{
	volatile unsigned int i; // Initialize counter for loop

	for (i = 0; i < vPOINTS; i++)
	{
		while (!(IFG2 & UCA0TXIFG))
			; // Wait if the transmitter has not completed the last
transmission

		UCA0TXBUF = tecVoltage[i]; // Send each of last 30 datapoints as
8-bit char
	}
}

/*
Description: Sample ADC and Return 30 8-bit Voltage Values
Inputs:	(unsigned int)ADC10MEM
Outputs:	void
Parameters:	void
Returns:	(unsigned int[])bufferVoltage
*/
void sample(void)
{
	volatile unsigned int i; // Initialize counter for loop

	for (i = 0; i < vPOINTS; i++)
	{
		bufferVoltage[i] = (float)ADC10MEM / 1023.0 * 255.0 + 0.5; //
Sample ADC 30 times, convert from 10-bit to 8-bit, round float to int, and
add to buffer array
	}
}

/*
Description: Update timers to adjust PWM duty cycle and pin to maintain set
temperature.
Inputs:	void
Outputs:	(float)TACCR1
Parameters:	(signed char)inputPWM
Returns:	void
*/
void updatePWM(signed char inputPWM)
{
```

```
	float dutyCycle = abs(inputPWM) / 127.0; // Duty cycle to write to PWM
pins
	float setTime = resetTime * (1 - dutyCycle); // Time low in
milliseconds

	if (inputPWM > 0) // Heating mode
	{
		// Set registers to enable PWM for P1.6
		P1SEL |= PWM_1;
		P1SEL2 &= ~PWM_1;

		// Clear registers to disable PWM for P2.6
		P2SEL &= ~(PWM_1 | PWM_2);
		P2SEL2 &= ~(PWM_1 | PWM_2);
	}
	else // Cooling mode
	{
		// Set registers to enable PWM for P2.6
		P2SEL |= PWM_1;
		P2SEL &= ~PWM_2;
		P2SEL2 &= ~(PWM_1 | PWM_2);

		// Clear registers to disable PWM for P1.6
		P1SEL &= ~PWM_1;
		P1SEL2 &= ~PWM_1;
	}

	TACCR1 = setTime * 4000 - 1; // Configured for 16 MHz & scale factor 2
}

#pragma vector = USCIAB0RX_VECTOR // Set pragma vector to serial data
recieved vector

/*
Description: Serial Data Recieved Interrupt Subroutine
Inputs:      (unsigned char)dutyCycle
Outputs:     (unsigned char)tecVoltage & (unsigned char)dutyCycle
Parameters:  void
Returns:     void
*/
__interrupt void serialInterrupt(void)
{
	PIDOutput = getData(); // Fetch PWM duty cycle data from recieve buffer
	sendData(); // Send last voltage reading datset over UART
	updatePWM(PIDOutput); // Update PWM duty cycle on timers & pins to
heat/cool
```

```
}

/*
Description: Main Subroutine
Inputs:      void
Outputs:     void
Parameters:  void
Returns:     void
*/
void main(void)
{
      initMSP(); // Calling the initMSP() function
      initUART(); // Calling the initUART() function
      initADC(); // Calling the initADC() function
      initPWM(); // Calling the initPWM() function

      while (!(IFG2 & UCA0RXIFG))
            ;
      getData(); // Wait to receive a one-time transmission from MATLAB to
begin

      __bis_SR_register(GIE); // Enable interrupts

      while (1)
      {
            sample(); // Sample data continuously
            memcpy(&tecVoltage, &bufferVoltage, vPOINTS); // When done
collecting, empty buffer into tecVoltage array to be sent to MATLAB
      }
}
```

## MATLAB App Designer Code

```
classdef DesignProjectSerialInterface < matlab.apps.AppBase

    % Properties that correspond to app components
    properties (Access = public)
        UIFigure                matlab.ui.Figure
        Gauge                   matlab.ui.control.LinearGauge
        Label                   matlab.ui.control.Label
        CurrentReadingLabel     matlab.ui.control.Label
        SetpointCEditField      matlab.ui.control.NumericEditField
        SetpointCEditFieldLabel  matlab.ui.control.Label
        RunSwitch               matlab.ui.control.ToggleSwitch
        RunSwitchLabel          matlab.ui.control.Label
        UIAxes                  matlab.ui.control.UIAxes
    end
```

```
    properties (Access = public)
        vPOINTS % Number of voltage datapoints recieved from MSP430 at a
single time to be averaged
        NPOINTS % Maximum number of datapoints to simultaneously display on
screen
        tSample % Sampling delay
        tArray % Array of times that samples were taken
        TempArray % Array of temperature samples
        FirstRun % Boolean value to conditionally perform tasks on the first
run
        SampleTime % Total sampling interval
        BaudRate % Baud rate
        V_ref % Reference voltage
        bits % Resolution of unsigned 8-bit value
        R_ref % Reference resistance of fixed resistor
        T_ref % Reference maximum possible temperature error
        bit_ref % Reference bit resolution of signed 8-bit value
        error % Temperature error
        last_error % Previous temperature error
        sum_error % Total sum of temperature errors
        K_p % Proportional gain
        K_i % Integral gain
        K_d % Derivative gain
        readingText % Placeholder text for readout label
        comport % Serial port
    end

    % Callbacks that handle component events
    methods (Access = private)

        % Code that executes after component creation
        function startupFcn(app)
            % Set all properties to their initial values on startup
            app.vPOINTS = 30;
            app.NPOINTS = 100;
            app.tSample = 0.1;
            app.tArray = [];
            app.TempArray = [];
            app.FirstRun = true;
            app.BaudRate = 115200;
            app.SampleTime = app.tSample + 480/app.BaudRate; % 480 comes from
8 bits * 2 ways * 30 datapoints
            app.V_ref = 3.3;
            app.bits = 255;
            app.R_ref = 9.994e3;
```

```
            app.T_ref = 45;
            app.bit_ref = 127;
            app.error = 0.0;
            app.last_error = 0.0;
            app.sum_error = 0.0;
            app.K_p = 15.0;
            app.K_i = 0.2;
            app.K_d = 5.5;
            app.readingText = "Current Reading (°C): ";

            % Set app UI properties for clean presentation
            app.UIAxes.YLim = [0, 50];
            app.RunSwitch.Value = "Off";

            app.comport = serialport('COM12', app.BaudRate); % Initialize
serial connection

            write(app.comport, 0, 'int8') % Send character to initialize
communication
        end

        % Close request function: UIFigure
        function UIFigureCloseRequest(app, event)
            app.comport = []; % Close serial connection

            delete(app)
        end

        % Value changed function: RunSwitch
        function RunSwitchValueChanged(app, event)
            UpdateView = app.RunSwitch.Value; % Get switch position

            switch UpdateView % Run program or remain dormant depending on
switch position
                case "On"
                    pause(app.tSample) % Wait for sampling delay

                    set_temp = app.SetpointCEditField.Value; % Get setpoint
from user

                    duty_cycle = app.error*app.K_p/app.T_ref*app.bit_ref +
app.sum_error*app.K_i/app.T_ref*app.bit_ref + (app.error -
app.last_error)/app.SampleTime*app.K_d/app.T_ref*app.bit_ref; % Obtain
desired duty cycle as output of PID controller
```

```
                    adj_duty_cycle = max(min(round(duty_cycle), app.bit_ref -
1), -app.bit_ref + 1); % Clamping the duty cycle to fit within the limits of
an 8-bit signed integer

                    write(app.comport, adj_duty_cycle, 'int8') % Write duty
cycle as signed 8-bit value to serial connection (negative = cooling,
positive = heating)

                    V_read = sum(read(app.comport, app.vPOINTS,
'uint8'))/(app.bits*app.vPOINTS)*app.V_ref; % Read ADC output over serial
connection, average as float, and convert to voltage
                    R_tec = V_read*app.R_ref/(app.V_ref - V_read); % Convert
voltage reading to resistance of thermistor
                    Temp = -1.0403*log(R_tec/30.374e3)/0.044 - 0.2649; %
Convert resistance reading to temperature of Peltier module

                    app.last_error = app.error; % Save previous temperature
error
                    app.error = set_temp - Temp; % Update current temperature
error
                    app.sum_error = app.sum_error + nthroot(app.error, 3); %
Update total sum of temperature error

                    if app.FirstRun % Run conditionally on first run unless
reset
                        app.tArray(end + 1) = 0; % Start time at reference 0
                        app.FirstRun = false;
                    else
                        if length(app.tArray) > app.NPOINTS % Limit live
temperature feed to window of size NPOINTS
                            app.tArray(1) = [];
                            app.TempArray(1) = [];
                        end

                        app.tArray(end + 1) = app.tArray(end) +
app.SampleTime; % Update time array
                    end

                    app.TempArray(end + 1) = Temp; % Update temperature array

                    app.CurrentReadingLabel.Text = app.readingText +
num2str(Temp, 4); % Update current reading output
                    app.Gauge.Value = Temp; % Update guage reading

                    plot(app.UIAxes, app.TempArray, 'XData', app.tArray,
Color='Blue') % Plot data on live window
```

```
                    hold(app.UIAxes, 'on')
                    plot(app.UIAxes, [app.tArray(1), app.tArray(end)],
[set_temp, set_temp], LineWidth=2, Color='Red') % Plot setpoint on live
window
                    hold(app.UIAxes, 'off')

                    app.RunSwitchValueChanged(); % Recursively re-run
function to continuously update
                case "Off"
                    % Reset all variables to initial values
                    app.tArray = [];
                    app.TempArray = [];
                    app.error = 0.0;
                    app.last_error = 0.0;
                    app.sum_error = 0.0;
                    app.CurrentReadingLabel.Text = app.readingText + "N/A";
                    app.FirstRun = true;

                    write(app.comport, 0, 'int8') % Turn off PWM output to
bring Peltier module back to room temperature
            end
        end

        % Value changed function: SetpointCEditField
        function SetpointCEditFieldValueChanged(app, event)
            app.sum_error = 0; % Reset integral error if the setpoint changes
for faster convergence
        end
    end

    % Component initialization
    methods (Access = private)

        % Create UIFigure and components
        function createComponents(app)

            % Create UIFigure and hide until all components are created
            app.UIFigure = uifigure('Visible', 'off');
            app.UIFigure.Position = [100.333333333333 100.333333333333 640
480];
            app.UIFigure.Name = 'MATLAB App';
            app.UIFigure.CloseRequestFcn = createCallbackFcn(app,
@UIFigureCloseRequest, true);

            % Create UIAxes
            app.UIAxes = uiaxes(app.UIFigure);
```

```
            title(app.UIAxes, 'Live Temperature View')
            xlabel(app.UIAxes, 'Time (s)')
            ylabel(app.UIAxes, 'Temperature (°C)')
            zlabel(app.UIAxes, 'Z')
            app.UIAxes.Position = [23 75 525 385];

            % Create RunSwitchLabel
            app.RunSwitchLabel = uilabel(app.UIFigure);
            app.RunSwitchLabel.HorizontalAlignment = 'center';
            app.RunSwitchLabel.Position = [580 157 27 22];
            app.RunSwitchLabel.Text = 'Run';

            % Create RunSwitch
            app.RunSwitch = uiswitch(app.UIFigure, 'toggle');
            app.RunSwitch.ValueChangedFcn = createCallbackFcn(app,
@RunSwitchValueChanged, true);
            app.RunSwitch.Position = [573 215 40 90];

            % Create SetpointCEditFieldLabel
            app.SetpointCEditFieldLabel = uilabel(app.UIFigure);
            app.SetpointCEditFieldLabel.HorizontalAlignment = 'right';
            app.SetpointCEditFieldLabel.Position = [78 33 74 22];
            app.SetpointCEditFieldLabel.Text = 'Setpoint (°C)';

            % Create SetpointCEditField
            app.SetpointCEditField = uieditfield(app.UIFigure, 'numeric');
            app.SetpointCEditField.Limits = [5 45];
            app.SetpointCEditField.ValueChangedFcn = createCallbackFcn(app,
@SetpointCEditFieldValueChanged, true);
            app.SetpointCEditField.Position = [167 26 48 36];
            app.SetpointCEditField.Value = 25;

            % Create CurrentReadingLabel
            app.CurrentReadingLabel = uilabel(app.UIFigure);
            app.CurrentReadingLabel.Position = [262 27 158 34];
            app.CurrentReadingLabel.Text = 'Current Reading (°C): N/A';

            % Create Label
            app.Label = uilabel(app.UIFigure);
            app.Label.HorizontalAlignment = 'center';
            app.Label.Position = [489 -13 25 22];
            app.Label.Text = '';

            % Create Gauge
            app.Gauge = uigauge(app.UIFigure, 'linear');
            app.Gauge.Limits = [0 50];
```

```
            app.Gauge.Position = [429 24 145 41];

            % Show the figure after all components are created
            app.UIFigure.Visible = 'on';
        end
    end

    % App creation and deletion
    methods (Access = public)

        % Construct app
        function app = DesignProjectSerialInterface

            % Create UIFigure and components
            createComponents(app)

            % Register the app with App Designer
            registerApp(app, app.UIFigure)

            % Execute the startup function
            runStartupFcn(app, @startupFcn)

            if nargout == 0
                clear app
            end
        end

        % Code that executes before app deletion
        function delete(app)

            % Delete UIFigure when app is deleted
            delete(app.UIFigure)
        end
    end
end
```

Shyavan Sridhar, 400388748
Lab Partner: Stephan Foster, 400384917
April 14, 2024

## Uncertainty Calculations

Uncertainty for the resistor used in the voltage divider was taken from [11] to be ±0.15% + 3 digits producing the following value:

$R = 9.994\ k\Omega$
$\sigma_R = 9.994(0.0015) + 0.003 = 0.02\ k\Omega$
$\therefore R = 9.99 \pm 0.02\ k\Omega$

Uncertainty in the measured values taken from the Fluke thermocouple were taken to be ±0.05% + 0.3°C as per [12]. These can then be added to the fact that the temperature difference observed was between 0.0 and 0.3°C to produce a maximum error of 0.6°C ignoring the percent term due to the relatively low temperature values worked with in-lab as that term becomes almost insignificant by comparison. The uncertainty from the thermocouple was also applied to measurements taken for the linear offset, a sample calculation for which is shown below:

$T = 47.3°C$
$\sigma_T = 47.3(0.0005) + 0.3 = 0.3°C$
$\therefore T = 47.3 \pm 0.3\ °C$

## Raw Data for Curves

Manufacturer Provided Data:

| Temperature (°C) | Resistance (kΩ) |
|---|---|
| 5 | 25.194 |
| 10 | 19.785 |
| 15 | 15.651 |
| 20 | 12.468 |
| 25 | 10.000 |
| 30 | 8.072 |
| 35 | 6.556 |
| 40 | 5.356 |
| 45 | 4.401 |

Measured Temperature Error Data:

| Set Temperature (°C) | Measured Temperature (°C) | Error in Measured Temperature (°C) |
|---|---|---|
| 5.5 | 5.5 | 0.3 |
| 10 | 10.5 | 0.3 |
| 15 | 15.7 | 0.3 |
| 20 | 20.5 | 0.3 |
| 25 | 25.1 | 0.3 |
| 30 | 30.4 | 0.3 |
| 35 | 35.8 | 0.3 |
| 40 | 41.4 | 0.3 |
| 45 | 47.3 | 0.3 |

# References

[1] "Lab 4 Temperature sensor - ENGPHYS 3L04:Engineering Metrology." Accessed: Mar. 24, 2024. [Online]. Available: https://avenue.cllmcmaster.ca/d2l/le/content/583463/viewContent/4372983/View

[2] "EP3BB3-2023-24-FinalProject-V2.0 - ENGPHYS 3BB3:Embedding and Programming a Micro-Controller." Accessed: Apr. 13, 2024. [Online]. Available: https://avenue.cllmcmaster.ca/d2l/le/content/600995/viewContent/4580102/View

[3] "Voltage Divider Circuit Calculator - For NTC Thermistor." Accessed: Mar. 24, 2024. [Online]. Available: https://www.petervis.com/electronics%20guides/calculators/thermistor/thermistor.html

[4] "How does a Thermoelectric Cooler Module work? – Thermal Book." Accessed: Apr. 13, 2024. [Online]. Available: https://thermalbook.wordpress.com/how-does-a-thermoelectric-cooler-tec-work/

[5] "H-Bridges – the Basics | Modular Circuits." Accessed: Apr. 13, 2024. [Online]. Available: https://www.modularcircuits.com/blog/articles/h-bridge-secrets/h-bridges-the-basics/

[6] "EP3BB3-2023-24-Lab5-V2.7 - ENGPHYS 3BB3:Embedding and Programming a Micro-Controller." Accessed: Apr. 13, 2024. [Online]. Available: https://avenue.cllmcmaster.ca/d2l/le/content/600995/viewContent/4558084/View

[7] "Basics of PWM (Pulse Width Modulation) | Arduino Documentation." Accessed: Apr. 13, 2024. [Online]. Available: https://docs.arduino.cc/learn/microcontrollers/analog-output/

[8] "What are the Principles of PID Controllers?" Accessed: Apr. 13, 2024. [Online]. Available: https://www.azom.com/article.aspx?ArticleID=22851

[9] "EP3BB3-2023-24-Lab4-V2.3 - ENGPHYS 3BB3:Embedding and Programming a Micro-Controller." Accessed: Apr. 13, 2024. [Online]. Available: https://avenue.cllmcmaster.ca/d2l/le/content/600995/viewContent/4542775/View

[10] "EP3BB3-2023-24-Lab6-V1.7 - ENGPHYS 3BB3:Embedding and Programming a Micro-Controller." Accessed: Feb. 25, 2024. [Online]. Available: https://avenue.cllmcmaster.ca/d2l/le/content/600995/viewContent/4561634/View

[11] "Handheld Digital Multimeters Test Bench ® Series Data Sheet", Accessed: Apr. 01, 2024. [Online]. Available: www.bkprecision.com

[12] "Handheld Digital Thermometer | Fluke 51 II | Fluke." Accessed: Apr. 14, 2024. [Online]. Available: https://www.fluke.com/en-ca/product/temperature-measurement/ir-thermometers/fluke-51-ii