2023

Keeping it cool: TEC temperature control with a PID controller

ENGPHYS 3BB4 FINAL DESIGN PROJECT REPORT

ABDULJAWAD (AJ) KOURABI AND EMILY NOBES

ENGPHYS 3BB4 Final Report – PID Control

Contents

Abstract

This project involved the development of a PID controller that used an MSP 430 microcontroller and a custom oscilloscope built in MATLAB to heat or cool a thermoelectric cooler unit (TEC). The controller helped the TEC reach a desired temperature set by the user within the graphical user interface of the MATLAB oscilloscope, and had a viable temperature range of about 7°C – 50 °C. The project involved two main sets of code, one in C to communicate with the MSP and send signals to an H-Bridge, and another in MATLAB to take in the current temperature and return a heating or cooling command, in addition to how intense the heating or cooling should be. The H-Bridge connected the MSP to the TEC, and allowed for the signal to be manipulated such that the TEC could heat or cool at designated intensities. Pulse width modulation was used to ensure the duty cycles were sent to the TEC in an appropriate fashion, and an analog to digital converter (ADC) was used to convert the analog data from the TEC into serial data that MATLAB could read and then display to the user.

Future iterations of the project could include improvements on the speed of temperature convergence, achieving lower temperatures through a bigger heat sink and a bigger fan, and the development of an automated script to better determine the PID coefficients.

Introduction

When systems require a specific temperature to carry out an operation, it is difficult to ensure this temperature will be maintained without the intervention of an appropriate control system. A temperature-determining control system must be capable of continuously monitoring and altering the temperature of a certain component to carry out a temperature maintenance process. System monitoring is typically carried out through the

implementation of a temperature sensor, which undergoes a physical change that can be correlated to changes in temperature. Using the data collected by this sensor, a robust control system should be capable of receiving the current state of the temperature-dependent device and initiating a heating or cooling process to accurately alter the temperature to achieve or maintain a target. A PID control system, for example, mathematically interprets the past, present, and predicted future behaviour of the system to predict an optimal next step.

The purpose of this project is to explore how PID control systems can be implemented to control the temperature of a thermoelectric cooler unit. This will be done through the strategic interfacing of hardware and software components. This report should serve as an overview of thermal control system implementation, troubleshooting, and optimization.

Theory

Thermoelectric Cooler Unit

Thermoelectric cooler (TEC) units are devices that can be heated or cooled within a specified range by the introduction of a current [1]. TEC units take advantage of the Peltier effect to dictate its heating or cooling behaviour and intensity. The Peltier effect occurs when a current is passed through several semiconductor junctions fabricated from a mix of p-type and n-type semiconductors. As the current passes through in a specific direction, some junctions will jump to a higher energy level, while the others will diminish to a lower energy level [1]. This effect is opposite to the Seebeck effect, which is used in thermocouples, and is the creation of a voltage in dissimilar materials due to a temperature gradient [1].

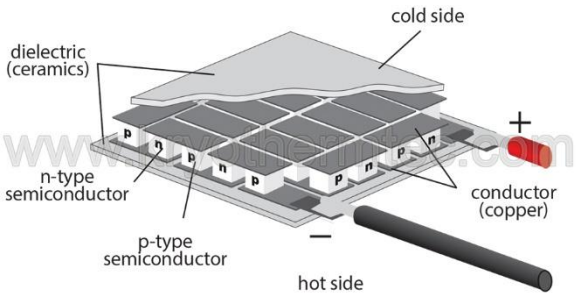

Figure 1 TEC unit [1]

Placing many of these junctions together between two ceramic tiles leads to the creation of a thermoelectric cooler, as shown in Figure 1. As current passes through the TEC, one side will experience heating and the other side will experience cooling. The heated/cooled side depends on the direction of the current flowing through the device [1].

The amount of heat transferred into the TEC is described below:

$$Q \propto I * t$$

Where Q is the amount of heat transferred, I is the current, and t is the amount of time that has passed. This means that the larger the current passed into the TEC the larger the amount of heat exchanged is going to be [1]. The addition of a heat sink enables effective cooling, as the heat dissipated needs to go somewhere in the device.

Analog to Digital Conversion

An Analog to Digital converter (ADC) is a system that converts an analog signal into a digital signal. The overall process is comprised of two subprocesses: sampling and quantization [2].

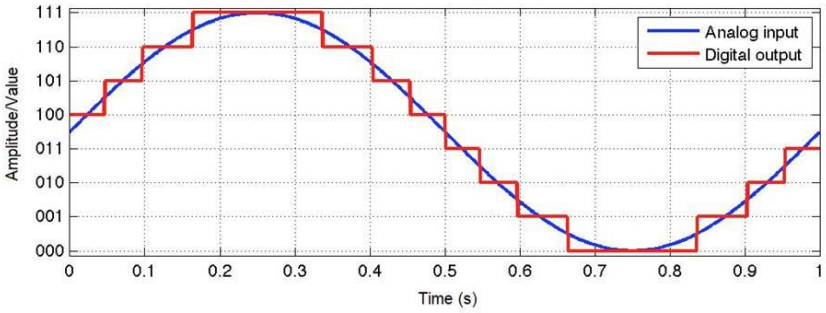

Figure 2 Typical Analog to Digital Conversion Pattern [2]

The sampling process simply collects data and readings on the amount of analog data passing through. The sampling process must occur at double the highest frequency of interest, in accordance with the Nyquist rate, in order to ensure that the signal is reconstructed effectively. Falling below this rate leads to aliasing, a type of intense signal distortion. Aliasing introduces a significant amount of noise and error. Sampling is often done at rates higher than the Nyquist rate, to account for errors and unexpectedly high frequencies [3].

In quantization, the signal is divided into specific intervals depending on the resolution of the ADC. If that ADC is 8 bits, then the resolution is $2^8 = 256$ intervals. These intervals are taken in then the signal itself is given a discrete voltage value.

The MSP430G2553 implemented in this project uses a successive approximation model to quantise and convert the analog data into digital data. Successive approximation starts at a specific voltage then iterates up or down until it reaches the target voltage. The amount in which it iterates up or down is dictated through a binary search, and so it increases the gap between guesses if it has converged, and it decreases the gap between iterations if it has. This allows for an efficient fast way to a continuous convergence of the data [2].

There are several factors such as sampling rate, conversion time, and resolution that could affect the speed of the ADC. ADC speed, within the context of this project, is extremely relevant as it can influence how fast the MATLAB code is able to get the data from the TEC and provide a return signal. If the ADC is slow, then the temperature data would be lagging the actual temperature. This would mean that MATLAB is returning incorrect duty cycles into the MSP, causing inaccuracies within the program. If the ADC is extremely fast, then the data is received quickly, and the appropriate duty cycle is sent back to the MSP.

Pulse Width Modulation

In addition to the information communicated by the ADC data, digital signals are employed to conduct the process of pulse width modulation (PWM) in the system. The process of PWM entails altering the duty cycle of a digital signal to control intensity [4]. In this project, PWM is employed to control the amount of current being passed through the H-Bridge to the TEC unit, thus modulating the intensity at which the unit is being heated or cooled.

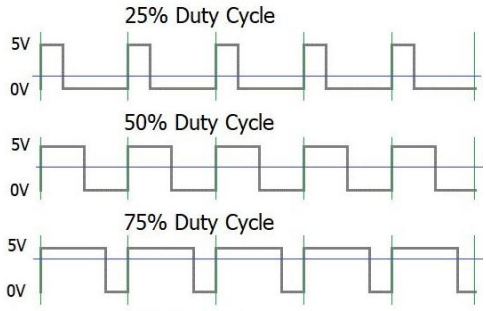

Figure 3 Pulse Width Modulation Examples

H-Bridge

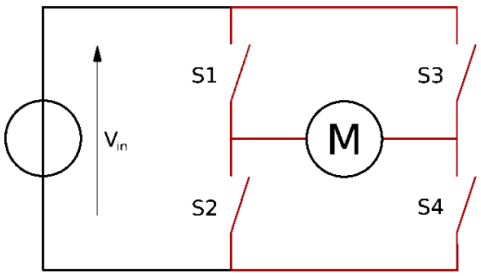

Figure 4 L298N H-Bridge Internal Circuit Diagram

H-Bridges are a type of physical control modules that can be manipulated to pass a desired amount of current to a concurrent device. The internal circuitry of these devices employs four switches – typically transistors - surrounding a centre load, as shown in the adjacent figure [5]. These switches can be strategically closed to produce a desired current output, which can take on several forms. If two adjacent switches (S1 and S3, for example) are closed, the H-Bridge will take on a "static" mode, allowing no current to pass through. If two stacked switches (S1 and S2, for example) are closed, the circuit with be shorted. Thus, to successfully induce a desired current flow, two non-adjacent switches must be closed at once. The "forward" and "backward" modes can be achieved [5]. This project employs a L298N H-Bridge, represented in the schematic above. In this IC, closing S1 and S4 will induce a clockwise (or "forward") force on the centre load, whereas closing S2 and S3 will include a counter clockwise (or "backward") force on the centre load [5].

H-Bridges are typically employed in processes that require a motor to switch directions. In this project, the H-Bridge is responsible for controlling the heating and cooling of the TEC unit. According to the specifications of the TEC unit, a "forward" current induces heating and a "backward" current induces cooling. To introduce sensitivity to the heating/cooling process, PWM is employed in this project to dictate the amount of current being passed in either direction. This is done by holding one of the two switches of focus constantly closed and passing a pulse through the other. As the PWM switch opens and closes, current will only pass while the behaviour of both switches matches (i.e. they are both closed). This implies that a PWM with a high duty cycle (i.e. high for the majority of the time) will allow much more current to pass through than a PWM with a low duty cycle (i.e. high for the minority of the time). An operation that passes a high amount of current will result in rapid heating/cooling, whereas an operation that passes a low amount of current will result in

the opposite. To add additional nuance to this process, a control system can be implemented.

PID Control

A PID control stands for "Proportional-Integral-Derivative" controller, which is a control mechanism that employs feedback from the system to apply a correction, in hopes of getting the system to a set point [6].

The feedback in a PID control system is typically error $e(t)$, which is the difference between the current temperature and the desired temperature. As the name hints, there is a proportional component, an integral component, and a derivative component, all relating to the error. The control function, $u(t)$, is seen below.

$$u(t) = K_p\, e(t) + K_i \int_0^t e(t)dt + K_p \frac{\mathrm{d}\, e(t)}{\mathrm{d}(t)} \quad [6]$$

Whereby K_p, K_i, K_p, are all non-negative constants that are tweaked to amplify to mitigate specific parts of the PID. The main control usually comes from the proportional part, since that can dictate how far the temperature is from the set point. However, it can not account for everything. For example, the integral error is useful since it increases the control function not only based on the current error amount, but for how long it has persisted, as it is summing up the error across a period. Within the context of the project, this is useful for something like cooling where the TEC may struggle to go down in temperature completely, and so the integral amount would account for how long it has struggled to get to the set temperature and increase the control function.

Applying too much of the integral error can lead to overshoot and may end up oscillating. One way to help with overshoot is the derivative error since that aims to bring the value of change of the error to zero. A derivative error acts as a dampening function, continuously flattening the curve in as much possible. This is really useful for maintaining convergence at the set point.

Flowchart

Using the aforementioned components and data transmission methods, the following cycle can be implemented to fabricate a temperature control system.

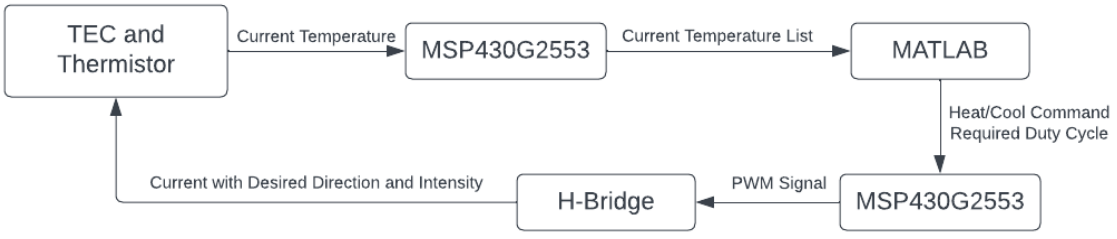

Figure 5 Hardware Flowchart for TEC unit Control System

Methods

Pseudocode

In addition to the physical design outlined in the above flowchart, software must be developed to support data transfer and processing, hardware interactions, and the PID control system. The software components of this project are executed between an MSP command script and MATLAB script, as elaborated in the following pseudocode. All function carried out by the MSP are in green and all functions carried out by MATLAB are in purple.

START

Initialize required 16 MHz clocks and configure watchdog timer

Initialize serial communication via UART

Initialize analog to digital conversion (ADC)

Activate infinite loop

    Activate communication between MSP and MATLAB

    Receive resistance data from TEC thermistor

    Convert TEC thermistor resistance data using ADC

    Send converted TEC thermistor resistance data to MATLAB

    Convert TEC thermistor resistance data to temperature

    Display current temperature on the GUI

    Compare current temperature to goal temperature

    Calculate PID value with determined coefficients

    Determine whether heating or cooling is required to meet goal temperature

    Convert PID value to a related duty cycle

    Send heating/cooling command and duty cycle to MSP

    Receive heating/cooling command and duty cycle

    If cooling command is received

        Activate cooling mode at desired duty cycle in the H-Bridge

    If heating command is received

        Activate heating mode at desired duty cycle in the H-Bridge

    END LOOP

END

MSP430G2553 Software Implementation

This project implements an MSP430G2553 microcontroller unit to facilitate communication between the physical components of the control circuit and the data processing carried out in MATLAB. A full copy of the MSP script is included in Appendix A, and the following section will explore the functions implemented to achieve the final result.

The main function of the MSP script is as follows.

```
void main(void)
{
    Init();
    Init_UART();
    Init_ADC();
    unsigned char dc ;
    unsigned char tc ;
    //cool(900);


while(1)
{
    getc(); //activating
    Sample(NPOINTS);
    Send(NPOINTS);
    tc = getc();
    dc = getc(); // duty cycle
    if (tc == 'h') {
      heat(dc);
      GREEN_LED = OFF;
    }
    else {
        cool(dc);
        GREEN_LED = ON;
    }
}
```

This central function carries out the entirety of the TEC heating/cooling effort and runs until the operations are physically terminated by the user. The first three functions (Init(), Init_UART(), and Init_ADC()) initialize the MSP's built in timers, serial communication method, and built-in analog to digital conversion process, respectively. For this project, data communication was executed without a UART bridge. This was done by enabling self-communication through the assignment of pins P1.1 and P1.2 as input and output pins, respectively, via the following assignment and the use of the MSP pin connectors.

```
P1SEL_bit.P1 = 1;
P1SEL2_bit.P1 = 1;
```

```
P1SEL_bit.P2 = 1;
P1SEL2_bit.P2 = 1;
```
Init_ADC() takes advantage of the MSP's built in 10-bit ADC to carry out the required data conversion. As mentioned before, this ADC applies successive-approximation methods to convert a received signal with, in this case, is the continuous resistance data received from the thermistor affixed to the TEC.

Since the built in ADC outputs a 10-bit signal and the communication method employed in this project can only handle 8-bit (or 1 byte) communication, the converted data must undergo a 2-bit shift in order to be processed. This is facilitated by the Sample(NPOINTS) function, where NPOINTS refers to the desired amount of data points to be collected before being sent to MATLAB.

```
void Sample(int n) // TO COMPLETE
{
 for(i = 0; i<=n;i++)
   v[i] = (ADC10MEM >> 2);
}
```
After this bit shift is executed, the data is sent to MATLAB using the Send(NPOINTS) function, which executes a putc() command to transmit the data through the enabled UART communication port.

```
void Send(int n) // TO COMPLETE
{
 for(i = 0; i<=n;i++)
   putc(v[i]);
}
```
From there, the data is received by MATLAB and processed (elaborated upon below) to determine whether the TEC must be heated or cooled to reach the desired temperature, and the duty cycle required to achieve a necessary current intensity to approach the desired temperature. This process is facilitated by the use of three getc() commands. The first received a character that activates the communication between the MSP and MATLAB. The second receives the heating/cooling command (tc) and the third receives the duty cycle (dc). Once these values are received, the script will enter a decision loop that determines the behaviour of the TEC for that cycle.

The tc variable will either receive an 'h' for heating or a 'c' for cooling. If an 'h' is received, the script will access the heat(dc) function.

```
int heat(int duty) {
    P2DIR = 0xFF;
    P2SEL = 0;
    P2SEL2 = 0;
    P2OUT_bit.P6 = 1;

    P1DIR |= BIT6; //Set pin 1.6 to the output direction.
```

```
    P1SEL |= BIT6; //Select pin 1.6 as our PWM output.
    TA0CCR0 = 255;
    TA0CCTL1 = OUTMOD_7;
    //TA0CCR1 = duty;
    TA0CCR1 = TA0CCR0 - duty;
    TA0CTL = TASSEL_2 + MC_1;
}
```

This function dictates the behaviour of two input pins in the H-Bridge (in1 and in2). In order to carry out a heating command, the H-Bridge must receive a "forward" current. This can be ensured by holding one pin at a constant "high" value and passing a PWM signal through the other. This function assigns P2.6 as the constant high value and P1.6 as the PWM pin, since these pins both run on the same clock type (TA0.0). The actual duty cycle of the signal is determined through a comparison of the TA0CCR0 value (held constant at 255) and the received duty cycle value, dc. In principle, the dc value represents a percentage of the maximum value (255) based on a scaling process that relates a PID value to a percentage of 255. Thus, a received value of 255 induces a 1% duty cycle (ie. very slow heating) whereas a received value of 1 induces a 100% duty cycle (ie. very fast heating). The implications of this process are further elaborated upon in the following MATLAB section, but the general goal of this process is to achieve rapid heating or cooling when the current temperature is very far away from the desired temperature, and much slower heating or cooling as the current temperature approaches the desired temperature to minimize overshoot.

The cool(dc) function operates on the same principle as the heat(dc) function but, instead, induces a backward current in the H-Bridge. Thus, the pin held at a constant value is held "low" and the PWM signal must be inverted, as shown below.

```
int cool(int duty) {
    P2DIR = 0xFF;
    P2SEL = 0;
    P2SEL2 = 0;
    P2OUT_bit.P6 = 0;

    P1DIR |= BIT6; //Set pin 1.6 to the output direction.
    P1SEL |= BIT6; //Select pin 1.6 as our PWM output.
    TA0CCR0 = 255;
    TA0CCTL1 = OUTMOD_7;
    //TA0CCR1 =  TA0CCR0 - duty;
    TA0CCR1 = duty;
    TA0CTL = TASSEL_2 + MC_1;
}
```

MATLAB Implementation

Within MATLAB, a variety of functions were performed. MATLAB functioned like the "brain" of the operation, and that started with defining how many points we were sampling and connecting with the serial comport in order to get the data from the MSP.

```
NPOINTS = 250;
comport = serial('COM5','BaudRate',115200,'FlowControl','none');
```

This allowed for sourcing the data in real time into MATLAB. What followed, then, was a systematic process of taking that data as an input and returning a number or character to IAR as an output. This output is meant to dictate two things: if the TEC should heat or cool, and at what intensity (ie duty cycle). Note that by now all relevant variables have been initialised, and that will not be included here for brevity.

First was an `fprintf` statement to confirm to IAR that the program is running, then the data (which is in the form of voltage readings) is divided into 8 bits.

```
d = fread(comport, NPOINTS, 'uint8');
```

After initialising some variables, a conversion takes place, taking in the voltage reading from the ADC and converting it into a temperature reading. This is the equations used:

Recalling that the voltage is coming from a voltage divider:

$$V_{out} = \frac{R_2}{R1 + R2} V_{in}$$

Gives:

$$R_1 = \frac{V_{in} * R_2}{V_{out}} - R_2$$

Given that the resistance of the thermistor, R_1, has predictable behaviour depending on temperature, R_1 is substituted for a temperature dependent expression. Note that this equation was derived through charting the data of the thermistor from 0 to 40 °C then creating a curve fit in excel.

$$\frac{V_{in} * R_2}{V_{out}} * R_2 = 32.33^{-0.045T}$$

Isolating for temperature:

$$T = -\frac{1}{0.045}\left(\ln\left(\frac{V_{in}}{V_{out}} R_2 - R_2\right) * \frac{1}{32.33}\right)$$

Which, in code, translates to (note that R is simply a placeholder for the calculation):

```
d = 3.3*d/255;
```

```
R = (((R1*3.3)./d)-R1)/1000;
d = log(R/(32.33))/(-0.045);
```

Them, certain variables are calculated and interpreted from the data. These variables include a list of the accumulated error, and current temperature. These then form the basis for the PID calculation, which is done in conjunction with the PID constants. The mean of the PID values is taken and that results in the "PIDval" which is used later for duty cycle calculation.

```
    %Getting accumulated error
    err_list = goal_temp - d;
    %current temp val
    current_temp = round(mean(d),2)

%   Setting PID values
    err = (goal_temp - current_temp);
    slope = mean(gradient(d));
    area = trapz(err_list);
    %Finding the PID val
    PID = Kp*err + Kd*slope + Ki*area
    PIDval = mean(PID);
```

What follows is calculating the requisite duty cycle that is sent to the MSP. This dictates how quickly the temperature of the TEC should change.

A low duty cycle, say 10, means that the heating or cooling intensity will be incredibly low. A higher duty cycle, say 255 (max), is a 100% duty cycle.

The PIDval provides an estimate of how far the temperature was from the set point, in accordance with the PID control. The higher the value, the higher the percentage of duty cycle passed.

The duty cycle was capped from 0 → 255 since any amount over that resulted in overflow when sending to the MSP and the result would not be interpreted well.

```
%Normalisation factor
    fac = 200;

    %calculating duty cycle
    if (abs(PIDval)+1) >= fac
        duty_cycle = 255;
    else
        if (abs(PIDval)+1) > 1
           duty_cycle = round(255*abs(PIDval)/fac);
        end
    end
```

In this case, it is seen that if the PIDval was above an arbitrary number that was picked during the lab, then the duty cycle was 100%. This indicated that the current temperature

is far enough from the set temperature and the TEC needed to catch up as soon as possible. Under that amount, it scales down proportionally. Meaning, that as the number decreases the percentage of the duty cycle also decreases. This is because of the $255 * \frac{\text{abs(PIDval)}}{\text{fac}}$. This simply resulted in the change of the duty cycle sent being related to the change in percentage of the PIDval over the factor. Setting the PIDval to the "absolute" was necessary since we did negative numbers.

After that point, the command of heating or cooling was determined. This was done through checking if the PIDval was negative or positive. The PIDval was only ever negative when the set temperature was below the target temperature, which meant that a negative PIDval meant cooling. The command was instantly sent to the MSP via the command `fprintf`

```
%Sending Heating/Cooling command
if PIDval > 0
    command = 'h';
    fprintf(comport, '%s',command);
else
    command = 'c';
    fprintf(comport, '%s',command);

    if duty_cycle < (256/2)
        duty_cycle = duty_cycle*2
    end
end
```

After that, the duty cycle is sent:

```
%Sending duty cycle
fprintf(comport, '%s',char(duty_cycle));
```

Following this communication was some additional data manipulation to plot the data as a continuously expanding window, and this was done through continuously appending the data to two lists. One list was for displaying the current target temperature and the other list was for displaying the current temperature. The current temperature is list T, and the goal temperature is list Tgoal.

```
%Data manipulation for plotting
T = [T,d'];
Tgoal_append = goal_temp*ones(NPOINTS);
Tgoal = [Tgoal;Tgoal_append];
```

Finally, the results were plotted and the function `smoothdata()` was utilised for effectively displaying the current temperature. This made the result a lot smoother on the graph while maintaining accuracy.

```
plot(smoothdata(T),'b');
```

This concludes the MATLAB section, covering how the data was received form the MSP, and then returning a duty cycle and a heating or cooling command.

Results

Final Circuit

The following is the final physical implementation of the temperature control system.

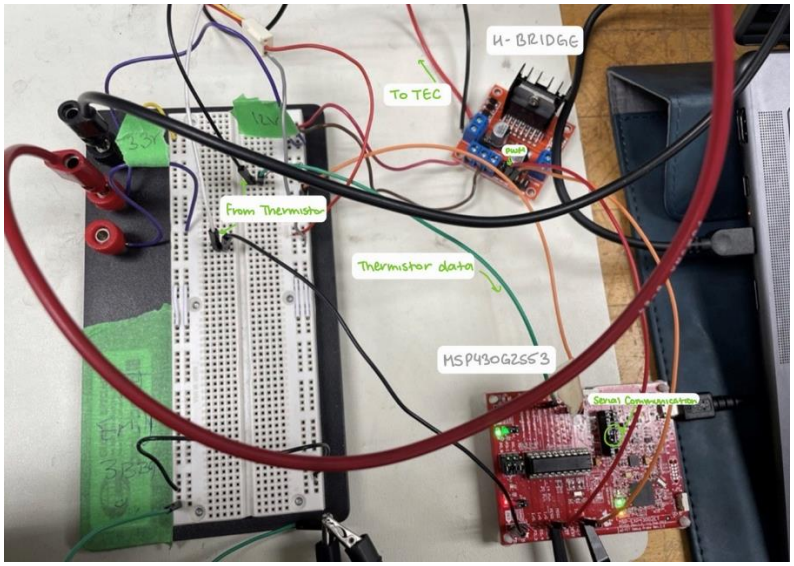

Figure 6 Final Circuit Implementation

Though the L298N TEC unit is not shown, it is directly connected to the H-Bridge via the wires labelled “to TEC”.

Final Results

Upon final implementation, the control system could facilitate heating and cooling within 0.5 degrees of the desired temperature. A general operating range of 13-45 degrees Celsius was observed, the causes of which will be elaborated upon in the discussion section of this report.

The following two pictures show two sample PWM states, the first demonstrating rapid cooling and the second demonstrating slow cooling.

Figure 7 PWM Samples

As expected, the rapid cooling scenario implements a high-duty cycle inverted signal (or low duty cycle non-inverted signal) to maximize the time in which both pins are in a low state. The slow cooling scenario, however, implements the opposite and only holds both pins low for a small amount of time. The small pulse seen in the constant (purple) signal in Figure 2 negligably impacted the functionality of the control system.

Using dynamic signals like those shown above, the following sample results were achieved.

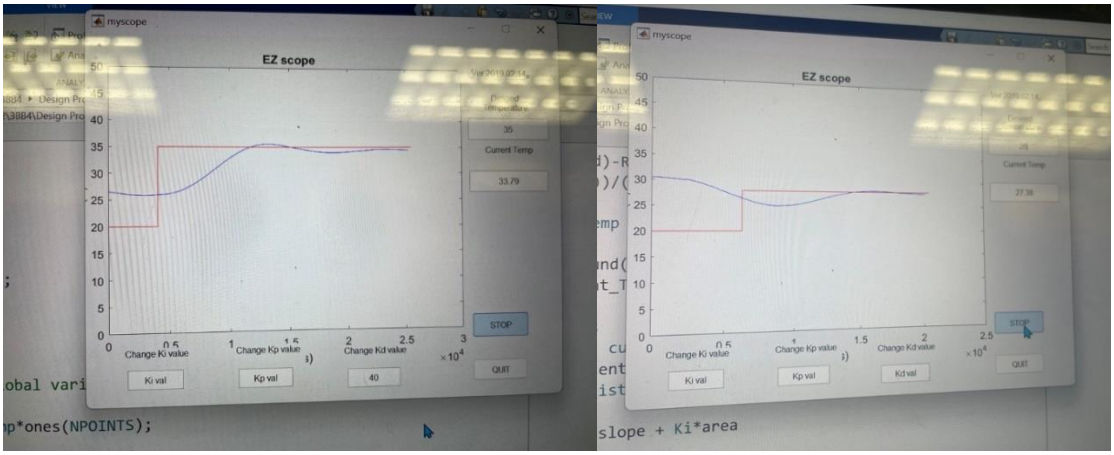

Figure 8 Temperature Convergence Samples

As shown, the final control system is capable of heating or cooling to a desired temperature and maintaining that value within a reasonable degree of accuracy once it is met. Both heating and cooling processes induce an initial overshoot as the current temperature approaches the goal temperature, but this is expected in PID control systems and the degree to which this occurs is not unreasonable.

Discussion

As shown by the results, this project concluded with a reasonably accurate final product capable of maintaining temperatures within the desired 0-50 degrees Celsius range. The result of the system adheres to the expected PID behaviour (ie. overshoots before settling on the desired temperature) and the final logic implemented to control the H-Bridge aligns

with what was expected according to the Theory section of this report. Achieving this result was facilitated by an involved troubleshooting process and could certainly be optimized to achieve further accuracy or expand the operating temperature range.

Troubleshooting

In order to meet the specifications of this project in an optimal way, troubleshooting strategies were implemented throughout. The following represent several notable troubleshooting results, but is not entirely exhaustive.

- The L298N TEC was fastened to a large heat-sink throughout the duration of this project. The heat-sink facilitating the heating/cooling process by capturing the heat emitted by the TEC as it cooled. This, however, meant the heat-sink accumulated heat as the TEC temperature was altered, eventually reaching a temperature that prevented the TEC from being cooled past a certain point. To retain temperature accuracy and satisfy the requirements of this project, a fan was affixed to the base of the heat sink and remained on for the duration of the TEC heating/cooling process, preventing adverse heat-sink interactions.
- For the majority of the project, heartbeat-like pulses would appear in the GUI output as the TEC was heated or cooled. These pulses appeared periodically and prevented accurate maintenance of a goal temperature. Several attempts were made to eliminate these pulses via overflow prevention, limit shearing, and current alteration, to no avail. In the end, it was determined that these pulses were artifacts of the external power supply used to power the TEC thermistor. Thus, the thermistor was hooked up to the internal 3.3V power pin of the MSP430G2553 and the pulses disappeared.
- At the debut of this project, NPOINTS was assigned a value of 400, implying the MSP will send 400 points to MATLAB each communication cycle. Though this many points allowed for high sensitivity, it resulted in severe lag in the GUI and eventual software crashes when >1000 seconds had passed. To prevent these adverse effects, the amount of data points collected and sent were altered until a “sweet spot” was found to be 150 points.

Future Improvements

Though the results achieved at the culmination of this project satisfied the provided requirements, there are several steps that could be taken to further optimize the result or streamline the development process for future implementation.

- Though the temperature was maintained within $\pm$ 0.5 degrees Celsius with the final PID coefficients, a slight oscillation remained present. It is possible that these oscillations could be eliminated completely via further coefficient tuning.
- The coefficients included in the final result were determined via trial and error, guided by visual observation of the system’s behaviour using the GUI. To achieve more optimal results – or to achieve an optimal result more quickly – a script could

be written to alter coefficient values based factors like oscillation, error, and time constant. This would also allow for easier adaptation to different TEC units.

- An additional script could also be written to detect and output the current time constant of the system, allowing for more nuanced visual troubleshooting. Ideally, the time constant should be minimized as much as possible and, though the final result of this project yielded a time constant of less than 30 seconds within the achievable range, there is plenty of room to improve efficiency.
- Since the heat-sinks employed in this project were quite large, high-power fans were proven to be much more capable of cooling the heat sinks in a reasonable time. Implementing high-power fans throughout the development and implementation process would encourage more consistent and accurate results as time goes on.

References

[1] "Thermoelectric cooler (TEC) construction," Kryotherm. Available http://kryothermtec.com/technology.html. Accessed 14 April 2023.

[2] M. Gudino. "Analog-to-Digital Converters Basics," Arrow. Available https://www.arrow.com/en/research-and-events/articles/engineering-resource-basics-of analog-to-digital-converters. Accessed 14 April 2023.

[3] "Nyquist Frequency," Gatan. Available https://www.gatan.com/nyquist frequency#:~:text=When%20a%20component%20of%20the,Nyquist%20before 20sampling%20into%20pixels. Accessed 14 March 2023.

[4] "Pulse Width Modulation: What is Pulse-width Modulation?" Sparkfun. Available https://learn.sparkfun.com/tutorials/pulse-width-modulation/all. Accessed 13 March 2023.

[5] "H-Bridges – the Basics," Modular Circuits. Available https://www.modularcircuits.com/blog/articles/h-bridge-secrets/h-bridges-the-basics/. Accessed 13 March 2023.

[6] "How Does a PID Controller Work?" Omega. Available https://www.omega.ca/en/resources/how-does-a-pid-controller-work. Accessed 14 March 2023.

Appendix A – MSP Code

```
#include "io430.h"

#define ON 1
```

```
#define OFF 0
#define DELAY 20000
#define ASCII_CR 0x0D
#define ASCII_LF 0x0A
#define BUTTON P1IN_bit.P3

#define GREEN_LED P1OUT_bit.P0

#define NPOINTS 250
unsigned char v[250];



void delay (unsigned long d)
{
  while (d--);
}

#pragma vector = PORT1_VECTOR
__interrupt void PORT1_ISR(void) {
    GREEN_LED = OFF;
    P1IFG_bit.P3 = 0;    // clear the interrupt request flag
}
void Init_UART(void)
{

//initialize the USCI
//RXD is on P1.1
//TXD is on P1.2

//configure P1.1and P1.2 for secondary peripheral function

P1SEL_bit.P1 = 1;
P1SEL2_bit.P1 = 1;
P1SEL_bit.P2 = 1;
P1SEL2_bit.P2 = 1;

// divide by  104 for 9600b with  1MHz clock
// divide by 1667 for 9600b with 16MHz clock
// divide by  9 for 115200b with 16MHz clock

UCA0BR1 = 0;
UCA0BR0 = 9;

// use x16 clock
```

```
UCA0MCTL_bit.UCOS16 = 1;

//select UART clock source
UCA0CTL1_bit.UCSSEL1 = 1;
UCA0CTL1_bit.UCSSEL0 = 0;

//release UART RESET
UCA0CTL1_bit.UCSWRST = 0;
}

unsigned char getc(void)
{
  while (!IFG2_bit.UCA0RXIFG);
  return (UCA0RXBUF);
}

void putc(unsigned char c)
{
  while (!IFG2_bit.UCA0TXIFG);
  UCA0TXBUF = c;
}

void puts(char *s)
{
  while (*s) putc(*s++);
}

void newline(void)
{
  putc(ASCII_CR);
  putc(ASCII_LF);
}

void itoa(unsigned int n)
{
  unsigned int i;
  char s[6] = "    0";
  i = 4;
  while (n)
  {
    s[i--] = (n % 10) + '0';
    n = n / 10;
  }
  puts(s);
}
```

```
void Init_ADC(void)
{
// initialize 10-bit ADC using input channel 4 on P1.4Send(NPOINTS);
// use Mode 2 - Repeat single channel
ADC10CTL1 = INCH_4 + CONSEQ_2; // use P1.4 (channel 4)
ADC10AE0 |= BIT4; // enable analog input channel 4
//select sample-hold time, multisample conversion and turn
//on the ADC
ADC10CTL0 |= ADC10SHT_0 + MSC + ADC10ON;
// start ADC
ADC10CTL0 |= ADC10SC + ENC;
}


int i = 0;
void Sample(int n) // TO COMPLETE
{
 for(i = 0; i<=n;i++)
   v[i] = (ADC10MEM >> 2);
}

void Send(int n) // TO COMPLETE
{
 for(i = 0; i<=n;i++)
   putc(v[i]);
}


void Init(void)
{
//Stop watchdog timer to prevent timeout reset
WDTCTL = WDTPW + WDTHOLD;

DCOCTL = CALDCO_16MHZ;
BCSCTL1 = CALBC1_16MHZ;

//P1REN = 0x08; //enable output resistor
P1OUT = 0x08; //enable P1.3 pullup resistor
P1DIR = 0x41; //setup LEDs as output
P1IE_bit.P3 = 1; //enable interrupts on P1.3 input
}

//in1 High, in2 0 is hot
//in1 Low, in2 0 is cold
```

```
int heat(int duty) {
    P2DIR = 0xFF;
    P2SEL = 0;
    P2SEL2 = 0;
    P2OUT_bit.P6 = 1;

    P1DIR |= BIT6; //Set pin 1.6 to the output direction.
    P1SEL |= BIT6; //Select pin 1.6 as our PWM output.
    TA0CCR0 = 255; //Set the period in the Timer A0 Capture/Compare 0 register to
1000 us.
    TA0CCTL1 = OUTMOD_7;
    //TA0CCR1 = duty;
    TA0CCR1 = TA0CCR0 - duty;
    TA0CTL = TASSEL_2 + MC_1;
}


int cool(int duty) {
    P2DIR = 0xFF;
    P2SEL = 0;
    P2SEL2 = 0;
    P2OUT_bit.P6 = 0;

    P1DIR |= BIT6; //Set pin 1.6 to the output direction.
    P1SEL |= BIT6; //Select pin 1.6 as our PWM output.
    TA0CCR0 = 255;
    TA0CCTL1 = OUTMOD_7;
    //TA0CCR1 =  TA0CCR0 - duty;
    TA0CCR1 = duty;
    TA0CTL = TASSEL_2 + MC_1;
}


void main(void)
{
    Init();
    Init_UART();
    Init_ADC();

    unsigned char tc;
    unsigned char dc;


while(1)
```

```
{
    getc(); //activating
    Send(NPOINTS);
    Sample(NPOINTS);

    tc = getc();  //temp command
    dc = getc(); // duty cycle

    if (tc == 'h') {
      heat(dc);
      GREEN_LED = OFF;
    }
    else {
      cool(dc);
      GREEN_LED = ON;
    }
}

}
```

Appendix B – MATLAB Code

```
function varargout = myscope(varargin)
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @myscope_OpeningFcn, ...
                   'gui_OutputFcn',  @myscope_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code
% --------------------------------------------------------------------
% Executes just before myscope is made visible.
function myscope_OpeningFcn(hObject, eventdata, handles, varargin)
global comport
global RUN
global NPOINTS
RUN = 0;
NPOINTS = 250;
comport = serial('COM5','BaudRate',115200,'FlowControl','none');
comport.Timeout = 50;
```

```
fopen(comport)
% Choose default command line output for myscope
handles.output = hObject;
% Update handles structure
guidata(hObject, handles);

%--------------------------------------------------------------------------
function myscope_OutputFcn(hObject, eventdata, handles, varargin)
%--------------------------------------------------------------------------

function Quit_Button_Callback(hObject, eventdata, handles)
global comport
global RUN
RUN = 0;
fclose(comport)
clear comport
if ~isempty(instrfind)
fclose(instrfind);
delete(instrfind);
end

% use close to terminate program
% use quit to terminate MATLAB
close
%--------------------------------------------------------------------------

function Run_Button_Callback(hObject, eventdata, handles)
%Defining basic variables
global comport
global NPOINTS
global RUN
%Defining variables relating to temp and duty cycle
global goal_temp;
global duty_cycle;
global current_temp;
%Defining PID variables
global Ki;
global Kd;
global Kp;
global err;
global PID;
global command; %This is sent to IAR to tell it to heat or cool
global err_list; %This is the accumilated error

%Assigning numbers
command = 'x';
goal_temp = 20;
duty_cycle = 500;
%PID values
Kp = 78;
Ki = 0.0008;
Kd = 35;
err_list = [];

%Initialising
```

```
T = [];
Tgoal = goal_temp*ones(NPOINTS);
PIDval = 1;
if RUN == 0
  RUN = 1;

  set(handles.Run_Button,'string','STOP')
  while RUN == 1
    % send a single character prompt to the MCU
    fprintf(comport,'%s','A');
    % fetch data as single 8-bit bytes
    d = fread(comport, NPOINTS, 'uint8');
    R = [];  %placeholder list to convert to temperature
    PID = [];
    %Value of circuit resistor
    R1 = 10032;
     %element wise operations to get temp
     d = 3.3*d/255;
     max(d);
     min(d);
     R = (((R1*3.3)./d)-R1)/1000;
     d = log(R/(32.33))/(-0.045);
    %Getting accumilated error
    err_list = goal_temp - d;
    %current temp val
    current_temp = round(mean(d),2)
    set(handles.current_T,'string',current_temp);

%   Setting PID values
    err = (goal_temp - current_temp);
    slope = mean(gradient(d));
    area = trapz(err_list);
    %Finding the PID val
    PID = Kp*err + Kd*slope + Ki*area
    PIDval = mean(PID);
    %Normalisation factor
    fac = 200;

%10 is super slow
%250 super fast

    %calculating duty cycle
    if (abs(PIDval)+1) >= fac
        duty_cycle = 255;
    else
        if (abs(PIDval)+1) > 1
           duty_cycle = round(255*abs(PIDval)/fac);
        end
    end

    %Sending Heating/Cooling command
    if PIDval > 0
        command = 'h';
        fprintf(comport, '%s',command);
    else
```

```
        command = 'c';
        fprintf(comport, '%s',command);

        if duty_cycle < (256/2)
            duty_cycle = duty_cycle*2
        end
    end
    %Sending duty cycle
    fprintf(comport, '%s',char(duty_cycle));
    %Data manipulation for plotting
    T = [T,d'];
    Tgoal_append = goal_temp*ones(NPOINTS);
    Tgoal = [Tgoal;Tgoal_append];
   %Plotting
    plot(smoothdata(T),'b');
    hold on
    plot(Tgoal,'r');
    hold off
    title('EZ scope');
    xlabel('Time (s)');
    ylabel('T (C)');
    ylim([0,50]);

    drawnow

   end
 else

  RUN = 0;
  % change the string on the button to RUN
  set(handles.Run_Button,'string','RUN')
end

fclose(comport);
clear comport;

%--------------------------------------------------------------------

function goal_temp_Callback(hObject, eventdata, handles)
global goal_temp
goal_temp = str2double(get(hObject,'String'));
% hObject    handle to goal_temp (see GCBO)
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of goal_temp as text
%        str2double(get(hObject,'String')) returns contents of goal_temp as a double


% --- Executes during object creation, after setting all properties.
function goal_temp_CreateFcn(hObject, eventdata, handles)
% hObject    handle to goal_temp (see GCBO)
% handles    empty - handles not created until after all CreateFcns called

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
```

```
    set(hObject,'BackgroundColor','white');
end

function Ki_Callback(hObject, eventdata, handles)
global Ki;
Ki = str2double(get(hObject,'String'));
% hObject    handle to Ki (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of Ki as text
%        str2double(get(hObject,'String')) returns contents of Ki as a double


% --- Executes during object creation, after setting all properties.
function Ki_CreateFcn(hObject, eventdata, handles)

% hObject    handle to Ki (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function Kd_Callback(hObject, eventdata, handles)
global Kd ;
Kd = str2double(get(hObject,'String'));
% hObject    handle to Kd (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of Kd as text
%        str2double(get(hObject,'String')) returns contents of Kd as a double


% --- Executes during object creation, after setting all properties.
function Kd_CreateFcn(hObject, eventdata, handles)
% hObject    handle to Kd (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function Kp_Callback(hObject, eventdata, handles)
```

```
global Kp;
Kp = str2double(get(hObject,'String'));
% hObject    handle to Kp (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of Kp as text
%        str2double(get(hObject,'String')) returns contents of Kp as a double


% --- Executes during object creation, after setting all properties.
function Kp_CreateFcn(hObject, eventdata, handles)
% hObject    handle to Kp (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function current_T_Callback(hObject, eventdata, handles)
% hObject    handle to current_T (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of current_T as text
%        str2double(get(hObject,'String')) returns contents of current_T as a double


% --- Executes during object creation, after setting all properties.
function current_T_CreateFcn(hObject, eventdata, handles)
% hObject    handle to current_T (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```