Milestone 2: Closed Loop Systems

Nick W, John G
Rowan University

December 3, 2018

Abstract

Milestone 2 utilizes the previous concepts used in labs and Milestone 1. The main objective of this milestone was to create an automatic climate controller. Essentially, a heat source was used to heat up a temperature sensor. The MSP430 controlled a fan pointed at the temperature sensor and through UART, a desired temperature was set. The MSP430 was required to regulate the fan's speed. The circuit was able to successfully pass all four temperature tests.

1 Introduction and Objectives

In the previous milestone, the MSP430 in conjunction with serial communication allowed the control of LEDs through UART protocol. To understand the extent of control the MSP430 provides, the second milestone require students to design a micro-controlled system that can regulate the temperature of a system. Allowed only a computer fan, voltage regulator, a resistive load, and the MSP430 control board, students must use these components to design a temperature control system. Students will be required to ultimately utilize skills covered in class, such as converting analog signals to digital signals and UART communication.

The primary goal of this assignment is to design a micro-controlled system that can regulate temperature based on the input of a user and the current temperature of its surroundings. In order to successfully implement this system, the following objectives must be met:

- Utilize a voltage regulator, power supply, and an appropriate load to generate sufficient heat (at least 65° Celsius).
- Measure the heat and convert it to a voltage using either a PTAT or thermistor.

- Design a set of equations to determine the temperature of the thermistor based on the provided voltage.
- Design a system that can cool down and stabalize the voltage regulator's temperature utilizing PWM and a provided computer fan.
- Implement the ability for users to input a desired temperature and read current temperatures utilizing Realterm and UART interrupts.

Possible applications for this concept would be in home thermostats. These take the homeowner's desired temperature, and control the heater or A/C to bring the internal temperature to the desired value.

2 Background and Relevant Theory

2.1 Using the MSP430's ADC

2.2 Constructing the Circuit

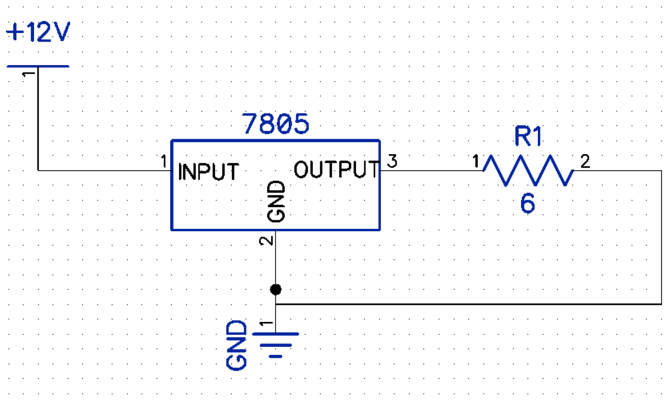

Figure 1: Heating Element Circuit

Once the code has been finalized, the circuit was the next aspect of the milestone to be addressed. Figure 1 shows the schematic of the heating circuit. In order to generate heat, a voltage regulator must draw enough current to dissipate power in the form of heat. Utilizing a 5V LM7805 regulator, the initial goal was to draw at least 1 watt from the power supply. Selecting a 6 Ohm, 50 Watt power resistor, the current drawn from the power supply is based on the following equation below:

$$P = V * I \tag{1}$$

With 12V as VCC and the power resistor equal to 6Ω, the current drawn would be equal to approximately 2A. This means that the power dissipated by the regulator would be:

$$P = \Delta V * I = (12 - 5) * 2 = 14\ Watts \tag{2}$$

However, in this configuration, the voltage regulator tended to go into thermal shutdown. This means that the regulator produces so much heat, it begins to shut itself down to protect it from destruction. At this point, it would draw approximately 400mA. Even in thermal shutdown, the regulator still provided a sufficient amount of heat.

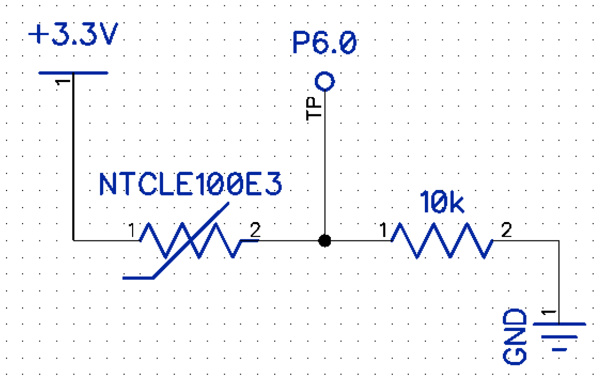

Figure 2: Temperature sensor circuit

Figure 2 shows the thermistor voltage divider used to measure temperature. The center of the divider was connected to pin 6.0 of the MSP430. After the heating circuit was constructed, the thermistor was secured to the voltage regulator using electrical tape. This ensured that the thermistor would maintain a secure connection to the regulator and read the correct temperature.

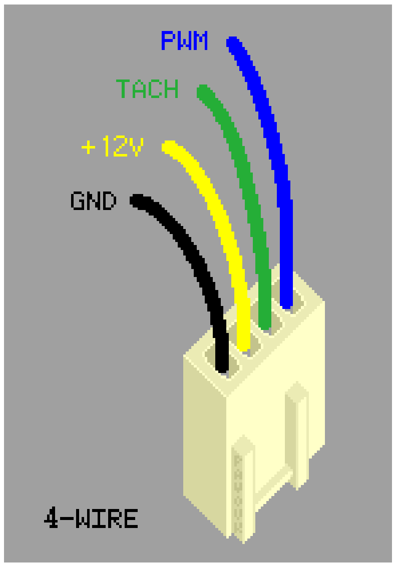

Figure 3: 4 pin computer fan pinout (Courtesy of: '4-Wire fans', $http://www.pavouk.org/hw/fan/en\_fan4wire.html$)

Figure 3 shows the pinouts of the computer fan provided. The fan was controller via the PWM wire (blue). This was connected to pin 1.3 of the MSP430. The fan's VCC and GND pins were connected to +12V and ground, respectively.

2.3 Reading The Temperature

Initially, all groups were given the option to use a thermistor or a PTAT to measure temperature.

For this milestone, a NTCLE100E3 thermistor was determined to be the best choice. The thermistor used is a 10KΩ variable resistor that decreases resistance as temperature increases.

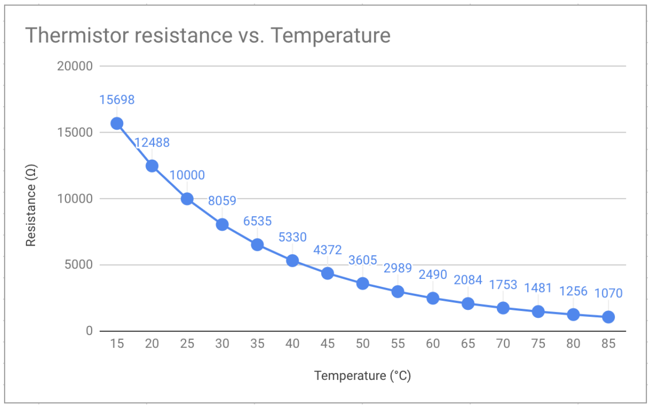

Figure 4: Thermistor Resistance vs. Temperature

Figure 4 shows the thermistor's resistance as temperature changes. When analyzing the shape of the line, resistance exponentially decreases as temperature gets higher.

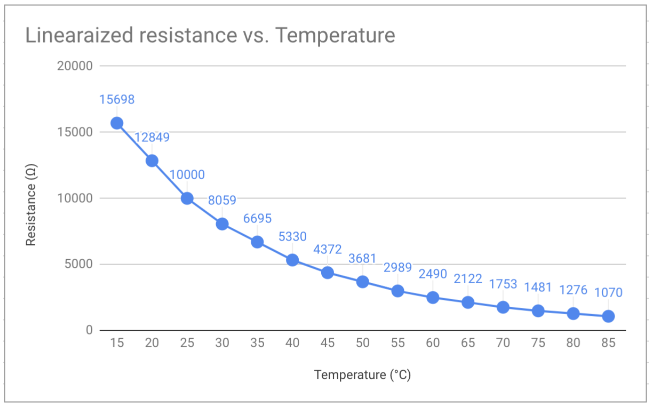

Figure 5: Linearized Resistance vs. Temperature

An issue arises when computing the temperature using an exponential function; it takes a considerable amount of time. In order to reduce computation time, the exponential line is broken up into linear sections (Figure 5). These sections were: 15-30, 30-45, 45-60, 60-75, and 75-90 degrees Celsius. The equations of these lines are taken and solved for R (the Y-axis).

The thermistor is used in a voltage divider. Since all values are known except temperature and voltage, the linear thermistor and voltage divider equations can be combined to form cases.

Temperature Range	Voltage Range	Equation
15-30	Less than 1.74	$t = -\frac{165000-171225V}{2849V}$
30-45	1.74 to 2.22	$t = -\frac{330000-262460V}{2729V}$
45-60	2.22 to 2.59	$t = -\frac{330000-205960V}{1383V}$
60-75	2.59 to 2.84	$t = -\frac{330000-169120V}{737V}$
75-90	Greater than 2.84	$t = -\frac{330000-145640V}{411V}$

This table showcases the five different lines that the thermistor's resistance was broken into. In order to find the thermistor's temperature t, the voltage V needs to be read from the MSP430's ADC. The voltage determines the equation to use. The voltage can then be plugged into its respective equation and the thermistor's temperature is returned.

2.4 The P controller

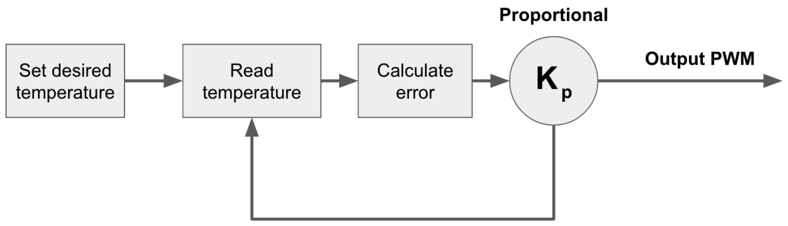

Figure 6: P Controller Block Diagram

Figure 6 depicts a block diagram of a P controller. The P controller is an integral part of the Milestone, and allows for the user to set a desired temperature. Essentially, the P controller takes the user's desired temperature, and finds the quickest way to get to the desired temperature.

Firstly, the temperature is read. This temperature is compared to the desired temperature. The difference between these two values is the error. By the error size, the P controller can set the fan's PWM accordingly. This is done by multiplying the error by a user-specified Kp value. A large P value will reach the desired temperature very quickly, but the read value will tend to oscillate around the desired temperature heavily. A smaller P value will reduce this oscillation, but the response of the system will be slower. Depending on the application of the controller the Kp value needs to be fine tuned for the optimal response.

To better understand the block diagram, consider the following example of an optimal system: If desired temperature is 30°C and the actual temperature is 100°C, the PWM will be set to the maximum and slowly scale down as the actual temperature reaches the desired temperature (the error becomes smaller). This sequence is looped until the error reaches zero, and in this case, the fan turns off.

```
1    currentTemp = temp; // Load temperature
2    error = target - currentTemp; // Find error
3    integral = integral + 1 * error * dt; // Find integral
4    derivative = (error - previousError) / dt; // Find derivative
5    pwm = (kp * error) + (ki * integral) + (kd * derivative); // Find PWM
6    previousError = error; // Store last error
7    if (error > 1){pwm = 1;}
8    if (error == 0){pwm = 1;}
9    TA0CCR1 = pwm;
```

The code above is the P controller section from the final Milestone code. This keeps the temperature of the thermistor at the desired temperature inputted through UART.

A case that had to be accounted for was when the error was positive (Line 7). This is when the desired temperature was higher than the actual temperature. In order to reach a higher temperature, the fan must be turned off so that the regulator can produce heat fast. In this case, the PWM was set to be 1 (low). If this was not set, the fan PWM would be high because the error was high. This would result in the thermistor never reaching the desired temperature.

Another case was when the desired temperature was the actual temperature. Initially, the fan would go to max PWM. This is not desired because it would cool the thermistor away from the desired temperature. This would cause oscillations between temperatures, which is undesired. In order to remedy this case, the PWM was forced to zero if the error was zero (Line 9).

2.4.1 The PID Controller

An important facet to note about the code is that there is a full fledged PID controller that can be used. This uses a integral (I) to bring the error to an absolute zero. This works by measuring where the temperature is in relation to the desired temperature. If there is a little error that the P part does not account for, the rolling sum of the integral will add up and eventually force the error to become zero. The derivative (D) is used to find the slope of the curve as the error decreases. This means that theoretically, the derivative will become smaller as the error approaches zero. This is factored into the outputted PWM value, which would reduce the amount of oscillation as the error approaches zero.

In the case of this milestone, it was deemed unnecessary.

2.5 The Big ”SUCC”

During initial testing, the fan alone was unable to cool the thermistor to temperatures lower than 30°C. In order to cool the thermistor, the 🅱ig SUCC was introduced. This was a paper cone placed over the fan's outlet. As a result, the temperature of the thermistor dropped to as low as 21°C. Additionally, the cooling time response was much quicker.

2.6 UART I/O

In order to process the temperature read by the thermistor, UART communication through Realterm was utilized. The baud rate ultimately chosen for this assignment was 9600.

In order to store the varying voltage values provided by the thermistor, these voltages were stoed in the ADC12MEM0 register for further use. The temperature was then defined by a given a voltage value based on the previously discussed equations. In order to properly read the value stored in the temperature register, the value was assigned the unit8 type. This allows users to perceive the data in terms of a decimal value rather than in binary or hexadecimal. Lastly, the UCA1TXBUF was assigned the value in temperature and relayed back to the computer, displaying in Realterm for the user to read.

In addition to relaying temperature values to the user, UART was used to send in values to regulate the temperature of the voltage regulator. In order to achieve this, a UART interrupt was established in the code. The purpose of this interrupt was to recognize the input of a user through Realterm and assign the target value based on the input of the user (defined by UCA1RXBUF).

3 Discussion and Results

3.1 Testing

The testing sequence consisted of four sequences:

1. Room temp - 65°C
2. 65°C - 35°C
3. 35°C - 45°C
4. Changing power supply and fan parameters (Adding disturbances)

The fan exceeded expectations in all four of the tests, keeping the temperature within 0.75°C of the specified temperature with a slight oscillation. During the fourth test, the supply's power was reduced from 12V to 10V. During this operation, the fan still managed to keep the temperature within 0.75°C of the specified temperature. Furthermore, during initial testing, the fan's intake was blocked and the cone of the fan was moved away from the thermistor. In both these tests, the fan still managed to regulate the temperature.

3.1.1 Temperature ranges

By using a 6Ω 50 watt power resistor, the thermistor was able to reach approximately 70°C. To get the thermistor as hot as possible, it was taped tightly to the heat sink using electrical tape. In order to cool the thermistor and voltage regulator, a small heat sink was placed on the voltage regulator. Without the heat sink, the thermistor could reach approximately 35°C. With the heat sink, the thermistor could reach approximately 22°C.

3.1.2 P controller tuning

As discussed in "The P controller", the P part of the controller operates by multiplying the error by a user-specified Kp value. In this case, the Kp value was set to 100. This proved to provide the fastest response.

3.2 Issues and Solutions

3.2.1 MOSFET Low-side switch

Initially, in order to control the fan, a low-side switch was used. This configuration used an NMOS with the fan on the drain and the MSP430's output on the gate. The MSP430's PWM output would switch the MOSFET on and off to create a PWM signal on the fan. However, no matter the MSP430's output, the fan would be on max speed the entire time. This issue would continue even if the MSP430 was disconnected and the input was grounded; the fan would turn on and off randomly. Precautions were taken to resolve these issues such as placing a drain resistor on the switch, but this did not work. Due to these ghosting issues, this was proven to be an inadequate solution.

Fortunately, the fan that was provided was a four-pin model. This meant that it had a VCC, Ground, Tachometer, and a PWM pin. This PWM pin could be directly fed a PWM signal from the MSP430 and the fan would work properly and as expected.

4 Conclusions

This Milestone utilized and supported previous concepts used in labs and Milestone 1. The fan and MSP430 were able to successfully pass all four temperature tests. Furthermore, the temperature was kept within 0.75°C of the target temperature with little oscillations. In order to obtain these results, the P controller was fine tuned to provide the fastest fan response with the least oscillations. For this, the Kp value was set to 100. Because the P controller was proficient in temperature regulation, a full PID controller was deemed unnecessary.

5 Appendices

```c
#include <msp430.h>
#include "stdint.h"
#include "stdlib.h"

#define testFlag 1  // testing flag

/* Global Variables*/
uint8_t lowBit = 0, highBit = 0;
float ADC_Voltage, temp;
float volt;
/* Init Methods*/
void UART_Init(void);
void ADC_Init(void);
int state = 0;

int currentTemp;
volatile float error;
volatile float previousError;
volatile int integral;
volatile int derivative;
volatile int kp = 10;
volatile int ki = 0;
volatile int kd = 0;
volatile float dt = 0.01;
float clkFrequency = 1000000.0;
int maxpwm = 255;
int minpwm = 1;
int target = 40;
int pwm = 0;

int main(void)
{
  P1DIR |= BIT3;
  P1OUT |= BIT3; //sets PWM on initially

  WDTCTL = WDTPW + WDTHOLD;                  // Stop WDT

  TA0CCTL1 = CCIE;
  TA0CTL = TASSEL_1 + MC_1 + ID_0 + TAIE + TAIFG;  /SMCLK UPMODE, CLEAR
  TA0CCR1 = 64;
  TA0CCR0 = 255;



ADC_Init();
UART_Init();

  while (1)
  {
    ADC12CTL0 |= ADC12SC;

    lowBit = ADC12MEM0;
    highBit = ADC12MEM0 >> 8;
    volt = ADC12MEM0;
    ADC_Voltage = (volt/4095) * 3.3;

    if((ADC_Voltage < 1.74))
    {

       temp = -((165000-(171225*ADC_Voltage))/(2849*ADC_Voltage)); //
    Temperature Range: 15-30
    }
    else if((ADC_Voltage >= 1.74)&&( ADC_Voltage < 2.22))
    {
       temp = -((330000-(262460*ADC_Voltage))/(2729*ADC_Voltage)); //
    Temperature Range: 30-45
    }
    else if((ADC_Voltage >= 2.22)&&( ADC_Voltage < 2.59))
    {
        temp = -((330000-(205960*ADC_Voltage))/(1383*ADC_Voltage)); //
    Temperature Range: 45-60
    }
    else if((ADC_Voltage >= 2.59)&&(ADC_Voltage < 2.84))
    {
        temp = -((330000-(169120*ADC_Voltage))/(737*ADC_Voltage)); //
    Temperature Range: 60-75
    }
    else if((ADC_Voltage >= 2.84))
    {
        temp = -((330000-(145640*ADC_Voltage))/(411*ADC_Voltage)); //
    Temperature Range: 75-90
    }

    currentTemp = temp; // Load temperature
    error = target - currentTemp; // Find error
    integral = integral + 1 * error * dt; // Find integral
    derivative = (error - previousError) / dt; // Find derivative
    pwm = (kp * error) + (ki * integral) + (kd * derivative); // Find PWM
    previousError = error; // Store last error
    if (error > 1){pwm = 1;} // Turn fan off when the desired temperature is
    higher than the actual temperature
    if (error == 0){pwm = 1;}   // When the actual temperature is the desired
    temperature, turn off the fan
    TA0CCR1 = pwm;   // Set output to desired PWM value

#if testFlag == 1
         while (!(UCA1IFG&UCTXIFG));           // USCI_A0 TX buffer ready?
                      UCA1TXBUF = (uint8_t)temp;
#elif testFlag == 0
    while (!(UCA1IFG&UCTXIFG));           // USCI_A0 TX buffer ready?
              UCA1TXBUF = highBit;     // TX -> RXed character

    while (!(UCA1IFG&UCTXIFG));         // USCI_A0 TX buffer ready?
              UCA1TXBUF = lowBit;     // TX -> RXed character
#endif
#if testFlag == 1
              //__delay_cycles(1000000);
#endif
              __delay_cycles(1000);

              __bis_SR_register(GIE);       // Enter LPM0, interrupts enabled
    __no_operation();                          // For debugger
  }
}

#pragma vector = ADC12_VECTOR
__interrupt void ADC12_ISR(void)
{
  switch(__even_in_range(ADC12IV,34))
  {
  case  0: break;                              // Vector  0:  No interrupt
  case  2: break;                              // Vector  2:  ADC overflow
  case  4: break;                              // Vector  4:  ADC timing overflow
  case  6:                                     // Vector  6:  ADC12IFG0
    if (ADC12MEM0 >= 0x7ff)                    // ADC12MEM = A0 > 0.5AVcc?
      P1OUT |= BIT0;                           // P1.0 = 1
    else
      P1OUT &= ~BIT0;                          // P1.0 = 0

  case  8: break;                              // Vector  8:  ADC12IFG1
  case 10: break;                              // Vector 10:  ADC12IFG2
  case 12: break;                              // Vector 12:  ADC12IFG3
  case 14: break;                              // Vector 14:  ADC12IFG4
  case 16: break;                              // Vector 16:  ADC12IFG5
  case 18: break;                              // Vector 18:  ADC12IFG6
  case 20: break;                              // Vector 20:  ADC12IFG7
  case 22: break;                              // Vector 22:  ADC12IFG8
  case 24: break;                              // Vector 24:  ADC12IFG9
  case 26: break;                              // Vector 26:  ADC12IFG10
  case 28: break;                              // Vector 28:  ADC12IFG11
  case 30: break;                              // Vector 30:  ADC12IFG12
  case 32: break;                              // Vector 32:  ADC12IFG13
  case 34: break;                              // Vector 34:  ADC12IFG14
  default: break;
  }
}

// Echo back RXed character, confirm TX buffer is ready first
#pragma vector=USCI_A1_VECTOR
__interrupt void USCI_A1_ISR(void)

{
    while (!(UCA1IFG & UCTXIFG));
    target = UCA1RXBUF;

}
void UART_Init(){
    P4SEL |= BIT4+BIT5;
    P3SEL |= BIT3+BIT4;                          // P3.3,4 = USCI_A0 TXD/RXD
    UCA1CTL1 |= UCSWRST;                         // **Put state machine in reset
    **
    UCA1CTL1 |= UCSSEL_2;                        // SMCLK
    UCA1BR0 = 6;                                 // 1MHz 9600 (see User's Guide)
    UCA1BR1 = 0;                                 // 1MHz 9600
   // UCA1MCTL |= UCBRS_1 + UCBRF_0;               // Modulation UCBRSx=1, UCBRFx
    =0
    UCA1MCTL |= UCBRS_0 + UCBRF_13 +UCOS16;
    UCA1CTL1 &= ~UCSWRST;                     // **Initialize USCI state
    machine**
    UCA1IE |= UCRXIE;// P1.0 output
}
void ADC_Init(){
    ADC12CTL0 = ADC12SHT02 + ADC12ON;         // Sampling time, ADC12 on
    ADC12CTL1 = ADC12SHP;                     // Use sampling timer
    ADC12IE = 0x01;                           // Enable interrupt
    ADC12CTL0 |= ADC12ENC;
    P6SEL |= 0x01;                            // P6.0 ADC option select
    P1DIR |= 0x01;
}

#pragma vector=TIMER0_A1_VECTOR
__interrupt void Timer0_A1_ISR (void) //CONTROLS PWM for the fan
{
switch(TA0IV)
    {
    case 2: //Case 2 defines interrupt on CCR1
            P1OUT &= ~BIT3;      //turn pin off for CCR1 interrupt
            break;
    case 14: //Case 2 defines interrupt on CCR0
        P1OUT |= BIT3;      //turn pin off for CCR0 interrupt
        break;
    }
}
```