AUTOMATED TEMPERATURE CONTROL FOR RAPID HEATING RATES IN AN ELEVATED TEMPERATURE ENVIRONMENT

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Mechanical Engineering

in the

College of Graduate Studies

University of Idaho

By

Victoria R. Kampfer

Major Professor: Robert Stephens, Ph.D.

Committee Members: John Crepeau, Ph.D.; David McIlroy, Ph.D.

Department Administrator: John Crepeau, Ph.D.

April 2015

AUTHORIZATION TO SUBMIT THESIS

This thesis of Victoria R. Kampfer, submitted for the degree of Master of Science with a Major in Mechanical Engineering and titled "Automated Temperature Control For Rapid Heating Rates In An Elevated Temperature Environment" has been reviewed in final form. Permission, as indicated by the signatures and dates below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor: ____________________________ Date: ____________
Robert Stephens, Ph.D.

Committee Members: ____________________________ Date: ____________
John Crepeau, Ph.D.

____________________________ Date: ____________
David McIlroy, Ph.D.

Department Administrator: ____________________________ Date: ____________
John Crepeau, Ph.D.

ABSTRACT

In order to meet the demands of industries and academic research, a cost effective temperature control system was developed to provide accelerated heat up rates between 5-10°C/sec. This apparatus was used to perform tensile tests on a 70XX series aluminum alloy to determine mechanical properties at elevated temperatures.

The automated temperature control system is comprised of two propane torches which heat each end of a tensile specimen during elevated temperature testing. Specimen temperatures are controlled by a PID algorithm which regulates stepper motor position and thus propane torch flame intensity. User inputs to the system are provided via a graphical user interface, with overall system control provided by an Arduino microcontroller.

Successful testing of the 70XX series aluminum alloy occurred at temperatures of 25°C, 225°C, and 425°C and strain rates of 0.05/sec and 0.5/sec. The results clearly show a direct relationship between increased temperatures and material elongation. Yield and ultimate tensile strength, however, decreased in value as temperature increased. Strain rate had an opposite effect on material properties and elongations as elevated temperatures, causing yield strength and ultimate tensile strengths to increase and elongation to decrease.

ACKNOWLEDGMENTS

I would like to express my appreciation for the support and guidance I received from Dr. Robert Stephens, without whom this research and thesis would never have reached completion. I would also like to extend my gratitude to committee members, Dr. John Crepeau and Dr. David McIlroy, for their contributions and recommendations. Many thanks are also extended to the University of Idaho mechanical engineering machine shop manager, Russ Porter, for his untiring assistance.

TABLE OF CONTENTS

AUTHORIZATION TO SUBMIT THESIS ii
ABSTRACT iii
ACKNOWLEDGMENTS iv
TABLE OF CONTENTS v
LIST OF FIGURES viii
LIST OF TABLES x
CHAPTER 1: INTRODUCTION 1
1.1 OBJECTIVES 2
CHAPTER 2: LITERATURE REVIEW 3
2.1 BEHAVIOR AND PROPERTIES OF ALUMINUM ALLOYS 3
2.1.1 FCC ALUMINMUM BEHAVIOR 3
2.1.2 TEMPERATURE AFFECTS ON MATERIAL PROPERTIES 7
2.2 ELEVATED TEMPERATURE TENSILE TESTING 9
2.2.1 TENSILE TESTING AND MATERIAL PROPERTIES 9
2.2.2 TESTING STANDARDS 12
2.3 REVIEW OF AVAILABLE TECHNOLOGY 13
2.3.1 GLEEBLE 13
2.3.2 INDUCTION HEATERS 15
2.3.3 ELECTRIC HEAT 16
2.3.4 PROCESS CONTROL: PID 17
CHAPTER 3: TEMPERATURE CONTROL & TENSILE TESTING APPARATUS DESIGN 23
3.1 MECHANICAL COMPONENTS 23
3.1.1 SPECIMEN GEOMETRY, GRIP GEOMETRY 24
3.1.2 TENSILE TESTING EQUIPMENT 27
3.1.3 THERMOCOUPLES AND SHIELDS 29
3.1.4 TORCH SUPPORTS 32
3.1.5 AUTOMATED CONTROL OF THE NOZZLE 34
3.1.6 ELECTRONIC HARDWARE 39

3.2 SOFTWARE SYSTEM DESIGN 44
3.2.1 PLC OVERVIEW 45
3.2.2 GLOBAL DECLARATIONS 48
3.2.3 VOID SETUP 50
3.2.4 VOID LOOP 52
3.2.5 SUPPORTING FUNCTIONS 53
3.2.6 HMI OVERVIEW 54
3.2.7 QT(PYQT4) 55
3.2.8 EBLIB 56
3.2.9 COM_MONITOR 57
3.2.10 LIVE_DATA_FEED 57
3.2.11 GUI_PID 58
3.2.12 IMPLEMENTATION 60
3.3 DEVELOPMENT COSTS 60
CHAPTER 4: TESTING PROCEDURE INSTRUCTIONS 62
4.1 TEMPERATURE CONTROL SETTINGS 62
4.2 SPECIMEN PREPARATION 63
4.3 THERMOCOUPLE ATTACHMENT 64
4.4 LOAD TRAIN SET-UP 66
4.5 TESTSTAR TESTING PROGRAM 72
4.6 TEST START 73
4.7 POST TEST DATA ANALYSIS 75
4.8 TEST PROCEDURE CHECKLIST 76
CHAPTER 5: TESTING RESULTS 79
5.1 TESTING HEAT RATE PROFILES 79
5.2 TENSILE RESULTS; ϵ = 0.05/s 86
5.3 TENSILE RESULTS; ϵ = 0.5/s 87
5.4 25°C TENSILE TESTING RESULTS 88
5.5 225°c TENSILE TESTING RESULTS 89

5.6 425°c TENSILE TESTING RESULTS 91

5.7 MECHANICAL PROPERTY ERROR BAR PLOTS 92

CHAPTER 6: CONCLUSIONS AND RECOMMENDATIONS 95

6.1 CONCLUSIONS 95

6.2 RECOMMENDATIONS 96

6.2.1 INCREASING SYSTEM CAPACITY 96

6.2.2 VALIDATION FOR ALTERNATE TESTING PLATFORMS 96

6.2.3 MODIFICATIONS TO CURRENT TECHNOLOGY 97

6.2.4 PID IMPROVEMENTS 98

6.2.5 IMPROVED TEMPERATURE MEASUREMENTS 99

APPENDIX A: DRAWING PACKAGE 110

APPENDIX B: HMI SOURCE CODE 134

APPENDIX C: PLC SOURCE CODE 157

LIST OF FIGURES

FIGURE 2.1: FCC UNIT CELL, FROM [8] 4

FIGURE 2.2: EFFECTS OF TEMPERATURE AND STRAIN RATE ON 6061-T6 ALUMINUM, ADAPTED FROM [9] 8

FIGURE 2.3: STRESS STRAIN CURVES OF VARIOUS MATERIALS, FROM [14] 10

FIGURE 2.4: PROCESS CONTROL OVERVIEW, FROM [29] 18

FIGURE 2.5: PID CONTROL IMPLEMENTATION, FROM [29] 21

FIGURE 3.1: SYSTEM SETUP 24

FIGURE 3.2: SYSTEM OVERVIEW 25

FIGURE 3.3: SPECIMEN GEOMETRIES 26

FIGURE 3.4: LOAD TRAIN CONFIGURATIONS 26

FIGURE 3.5: THERMOCOUPLE CLIP ATTACHMENT 31

FIGURE 3.6: COMPARISON OF SHIELD TYPES 32

FIGURE 3.7: TORCH SUPPORTS, (A) PINNED, (B) SEATED 34

FIGURE 3.8: SINGLE STAGE PRESSURE REGULATOR, ADAPTED FROM [31] 37

FIGURE 3.9: MOTOR ASSEMBLY 38

FIGURE 3.10: ELECTRICAL OVERVIEW 40

FIGURE 3.11: SAINSMART UNO MICROCONTROLLER BOARD, FROM [32] 41

FIGURE 3.12: PLC SOFTWARE OVERVIEW 47

FIGURE 3.13: GUI_PID DISPLAY SCREEN 55

FIGURE 4.1: SPECIMEN QUADRANT DIAGRAM 65

FIGURE 4.2: SEATED GRIP SPECIMEN ALIGNMENT 68

FIGURE 4.3: PINNED GRIP SPECIMEN ALIGNMENT 68

FIGURE 4.4: THERMOCOUPLE READER ATTACHMENT 70

FIGURE 4.5: EXTENSOMETER ATTACHMENT 72

FIGURE 5.1: A3 TCS DATA 80

FIGURE 5.2: A4 TCS DATA 81

FIGURE 5.3: B3 TCS DATA 81

FIGURE 5.4: B4 TCS DATA 82

FIGURE 5.5: A5 TCS DATA 82
FIGURE 5.6: A7 TCS DATA 83
FIGURE 5.7: B5 TCS DATA 83
FIGURE 5.8: B6 TCS DATA 84
FIGURE 5.9: HEAT-UP RATE ERROR BAR PLOT.. 85
FIGURE 5.10: STRESS STRAIN CURVES FOR 0.05/SEC STRAIN RATE 86
FIGURE 5.11: STRESS STRAIN CURVES FOR 0.5/SEC STRAIN RATE 88
FIGURE 5.12: 25° C STRESS STRAIN CURVES 89
FIGURE 5.13: 225° C STRESS STRAIN CURVES 90
FIGURE 5.14: 425° C STRESS STRAIN CURVES 92
FIGURE 5.15: YS ERROR BAR PLOT.. 93
FIGURE 5.16: UTS ERROR BAR PLOT 93
FIGURE 5.17: PERCENT ELONGATION ERROR BAR PLOT.. 94

LIST OF TABLES

TABLE 3.1: EXTENSOMETER TEMPERATURE MONITORING 29

TABLE 3.2: ARDUINO UNO REV2 SPECIFICATIONS, FROM [32] 43

TABLE 3.3: TCS SOFTWARE COMPONENTS 45

TABLE 3.4: LIST OF EXPENDITURES 61

TABLE 5.1: PID TESTING PARAMETERS 85

TABLE 5.2: 0.05 / SEC TENSILE TESTING RESULTS 87

TABLE 5.3: 0.5 / SEC TENSILE TESTING RESULTS 88

CHAPTER 1: INTRODUCTION

Industries dependent upon material behavior are continuously seeking new technologies to increase the performance of materials and to reduce production costs. Depending on the material and application, extremely large amounts of research and testing go into product development, with studies focusing on a variety of material properties. For example, Boeing focused on improving aluminum and titanium alloys to reduce weight for the design of the 777. This led to higher strengths, toughness, and corrosion resistance in 7000 and 2000 series aluminum alloys, and increased damage tolerance, corrosion resistance, and temperature resistance in titanium alloys [1].

Of particular interest to the research in this thesis is the testing of materials at elevated temperatures. This testing occurs in a variety of fields, such as the nuclear industry and in structural development. The nuclear industry performs oxidation tests at elevated temperatures to obtain data on advanced oxidation as well as oxidation resistance [2], while other research has investigated the effects of high temperature exposure to static and dynamic mechanical properties of cement [3]. Definition of material properties at elevated temperatures is also critical to the metal casting industry. As technologies improve, analytical models have been created to predict the material behavior of cooling casts to prevent cracking and unwanted residual stresses. These models rely heavily upon research and accurate thermo-mechanical data [4].

Current high temperature testing methods typically rely on resistance heating that can be initiated from a variety of devices. Direct resistance heating passes an electric current

directly through the material to heat it based on the material's resistance. Induction heating however, does not directly contact the subject that is to be heated, but rather uses a magnetic field to induce an electric current in the material which then heats through resistance. Both of these types of heating apparatuses provide quick heat up times, but can be fairly expensive to own and operate. Alternatively electric resistance heaters can be employed, as they are inexpensive and easy to use. The heaters themselves are heated by an electric current through resistance heating, but then transfer this heat to the specimen indirectly. Because of this, electric resistance heaters are less efficient and have slower heat up times.

1.1 OBJECTIVES

Due to barriers in elevated temperature testing, such as low heating rates and high costs, the objective of this research was to:

1. Develop a cost effective, elevated temperature testing apparatus that can be used in conjunction with a servo-hydraulic testing frame. Requirements of the device include maximum temperatures of 470°C, and heating rates between 5-10°C/sec. Costs were to be held to a minimum, with a goal of under $1000.
2. Perform elevated temperature tensile tests to validate the temperature control system and determine mechanical properties of a 70XX series aluminum alloy. Tests were performed at three different temperatures: 25°C, 225°C, and 425°C and two different strain rates: 0.05/sec and 0.5/sec. Specifically, percent elongation, 0.2% yield strength, and ultimate tensile strength values were to be analyzed and evaluated.

CHAPTER 2: LITERATURE REVIEW

This chapter will provide a brief background on the material behavior and temperature affects to material properties of aluminum alloys. Standards and procedures for elevated temperature tensile testing will also be discussed. Because the testing apparatus developed in this thesis focuses on the elevated temperature testing of a 70XX series aluminum alloy, the standards and procedures referenced are those applied to metallic alloys; however, in theory the same testing processes may be applied to non-metals as well. Current elevated temperature testing facilities and technology will also be reviewed, as will concepts pertinent to temperature control systems, such as the implementation of PID process controllers.

2.1 BEHAVIOR AND PROPERTIES OF ALUMINUM ALLOYS

Elevated temperature testing is commonly performed during the development of metallic alloys and aids in the prediction of material behavior in extreme environments [5]. For example, environmental temperature and time spent at that temperature may affect both a material's composition and its microstructure [6]. The following sections will discuss the material behavior of aluminum alloys and the effects of temperature on material properties.

2.1.1 FCC ALUMINMUM BEHAVIOR

Aluminum alloys are polycrystalline structures formed from multiple small crystals arranged in varying orientations. These small crystals are referred to as grains, while the edges between crystals are called grain boundaries. The grains are comprised of a lattice of

atoms occurring in regular, repetitive patterns similar to a grid. There are 14 unique lattice arrangements, which result in specific mechanical properties. For example, aluminum has a face centered cubic (FCC) lattice, as depicted by the unit cell in Figure 2.1, which allows atoms to be packed as tightly as possible in a cubic formation. The FCC arrangement has an atom positioned at each corner of the cube as well as along each face of the cube. For metals, deformation occurs more readily along directions of the cube where atoms are in closest proximity to one another. These directions and planes of atoms can be described for any lattice structure using Miller indices. FCC aluminum may have a close packed plane of the form {111} along the <$\bar{1}10$> direction as depicted in Figure 2.1, as well other planes and directions occurring along the diagonal of the cube [7].

Deformation in polycrystalline metals begins with dislocations, or line imperfections, in the

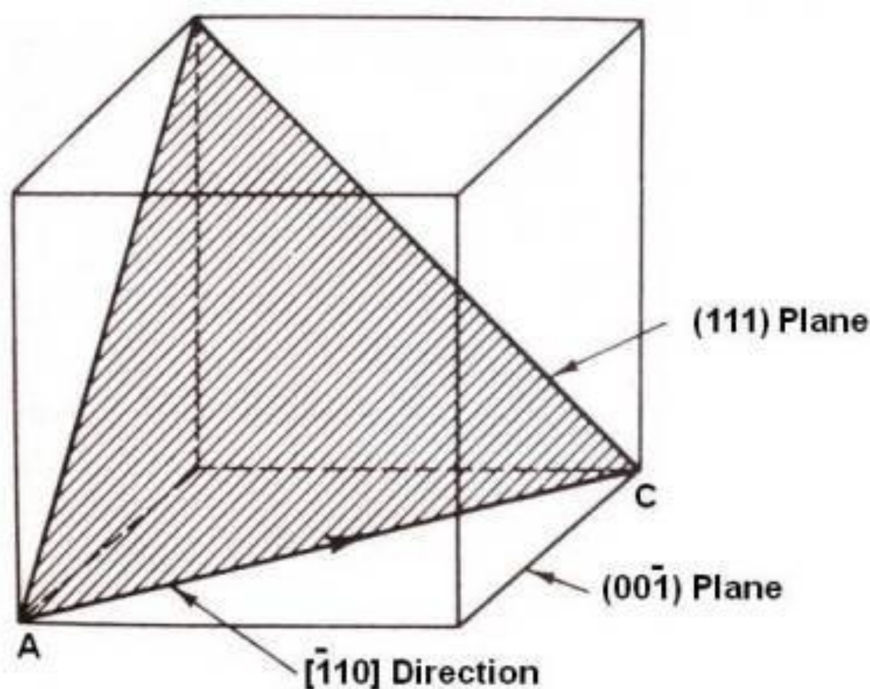


Figure 2.1: FCC Unit Cell, from [8]

crystal. These can occur during solidification of the material and when applied loads are great enough to cause dislocations. The movement of dislocations through crystals is known as slip, and occurs along what are referred to as slip planes. A common slip system for FCC aluminum is the {111} plane along the $<\bar{1}10>$ direction. If slip occurs, only a small number of the metallic bonds throughout the crystal need to be broken to plastically deform the metal and directly affect the material's strength. However, by interfering with slip, the mechanical properties of a metal can be controlled.

One method for preventing the movement of dislocations is through the introduction of obstacles. Obstacles commonly take the form of interstitial defects which are sites in the crystal lattice where an extra atom is inserted. These atoms are typically larger than the interstitial site they occupy and cause compressive stresses in the crystal that resist dislocation movement. Grain boundaries also hinder dislocation movement. Because of this, metals with finer grains and larger grain boundary areas typically exhibit higher strengths. Smaller grain sizes effectively increase the distance that dislocations have to travel to form a void, and are affected by a cast metal's cooling or solidification rate. Faster solidification rates typically lead to smaller grain sizes, while slower rates lead to larger grain sizes [7].

Alternatively various means of hardening can be applied to enhance the strength of aluminum. Manufacturers of aluminum alloys employ a variety of methods such as work hardening (cold working), solid solution hardening, dispersion hardening, and precipitation hardening. Cold working encompasses all rolling, extruding, drawing, bending, etc… of

aluminum products and is performed below the metal's recrystallization temperature. This leads to dislocations on different slip planes interfering with one another's movement, strengthening the material. Solid solution hardening is typical of most aluminum alloys as it involves alloying elements being dissolved in an aluminum base. Alloying atoms occupy positions or empty spaces in the aluminum lattice, causing it to distort and restrict the movement of dislocations, increasing the strength of the alloy. Dispersion hardening occurs when fine particles of an insoluble material are added to the base metal's lattice to obstruct dislocation movement. For aluminum alloys, this may occur by either the addition of an alloy that chemically combines with the aluminum, or each other, to create fine particles that precipitate from the metal, or by combining specific particles with powdered aluminum and then compressing the mix into a solid. Lastly, precipitation hardening consists of a solution heat treatment followed by an ageing process. The solution heat treatment produces a supersaturated condition, and after quenching the material is artificially aged at a temperature above room temperature. Alloys that undergo precipitation hardening must contain enough soluble alloying elements to surpass the room temperature solid solubility limit. They must also be capable of dissolving the excess soluble alloying elements and then later precipitate them as components of the crystal lattice. Care must be given during the precipitation reaction such that the components do not become too coarse and detract from the strengthening potential of the precipitation hardening process [6].

2.1.2 TEMPERATURE AFFECTS ON MATERIAL PROPERTIES

Most materials exhibit temperature dependencies, with yield strength, tensile strength, and modulus of elasticity decreasing at higher temperatures and ductility increasing. At higher temperatures it is common for the brittle nature of a material to be reduced and the ductility of the material to increase. The transition between these two fracture methods occurs at the ductile to brittle transition temperature, which is determined through impact testing. Although impact test results are not always related to tensile test results, materials with high strength and high ductility generally have good tensile toughness. Both test types are important for predicting material behavior in extreme environments [7].

Aluminum alloys at elevated temperatures typically follow the same material property trend as other materials, except when it comes to the ductile to brittle transition temperature. Most FCC metals, such as aluminum, do not exhibit a distinct transition temperature, because the FCC crystal structure allows for higher absorbed energies and no transition temperature [7]. However, FCC aluminum properties such as tensile, shear, compression, bearing and fatigue strengths have been found to follow the established trends and decrease with temperature. These decreases in property values though do not extend to the process of age-hardening, which is performed at specific temperature ranges and for only certain periods of time. The elongation of aluminum is commonly found to increase with temperature. Cold temperatures typically have a reverse effect on material properties, resulting in increased tensile, shear, compression, bearing and fatigue strengths, and decreased elongations [6].

This effect can be seen in Figure 2.2 which depicts stress strain plots of 6061-T6, an FCC aluminum alloy. From the plots it can be seen that 6061-T6 aluminum is not only temperature dependent, but also strain rate sensitive [9]. Additional elevated temperature research has tested 6061-T6 at temperatures of 25°C, 100°C, 200°C, and 300°C and at strain rates of 10^{-4} s^{-1}, 10^{-3} s^{-1}, 10^{-2} s^{-1}, and 10^{-1} s^{-1} with similar results. The FCC aluminum alloy was confirmed to be both temperature and strain rate sensitive with both yield strength and ultimate tensile strength values decreasing with a corresponding increase in temperature. No significant strain rate sensitivity was evident during room temperature (25°C) tests; however, it became much more prominent at higher temperatures. At 300°C the strain rate sensitivity was significant enough to result in a 20% decrease in elongation when the strain rate was decreased from 10^{-1} s^{-1} to 10^{-4} s^{-1} [10].

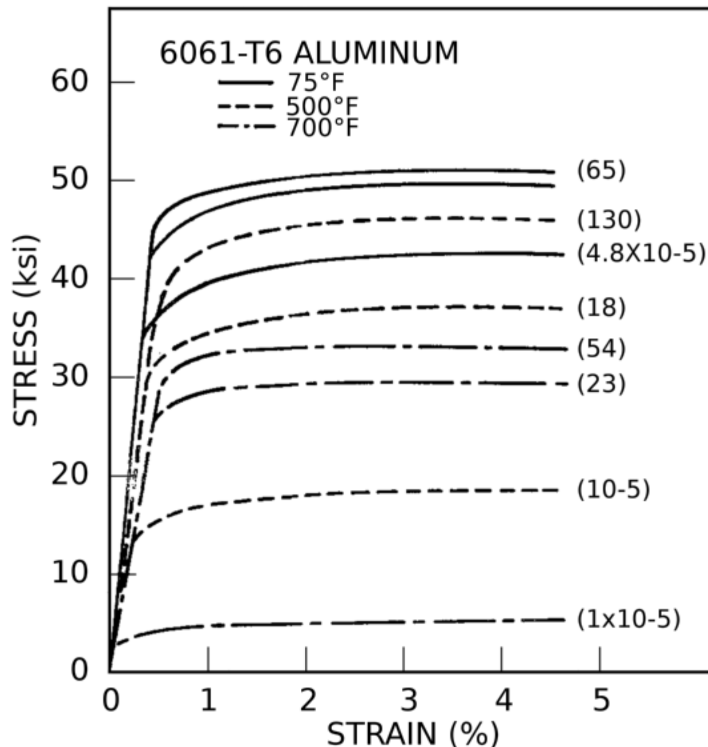


Figure 2.2: Effects of Temperature and Strain Rate on 6061-T6 Aluminum, adapted from [9]

2.2 ELEVATED TEMPERATURE TENSILE TESTING

The following section will discuss in detail tensile testing, stress strain curve generation, mechanical property calculations, and ASTM testing standards. Tensile tests provide the means, while generation of the stress strain curve provides the foundation for calculating mechanical properties. Standards for elevated temperature tensile testing appear in ASTM International E21. Testing standards specific to non-ferrous metal alloys is also discussed in ASTM International E21 [11].

2.2.1 TENSILE TESTING AND MATERIAL PROPERTIES

Tensile testing consists of placing a specimen in axial tension, increasing the load until failure, and recording the corresponding loads and displacements. By measuring the load and displacement stress and strain may be calculated, as defined in (2.1) and (2.2) respectively. In (2.1) and (2.2), σ is engineering stress, ε engineering strain, P the applied load, D_{min} the minimum diameter of the specimen, L the gauge section length, and L_o the original gauge section length.

$$\sigma = P/A = P/(D_{min}^2 * (\pi/4)) \tag{2.1}$$

$$\varepsilon = \Delta L/L_o = (L - L_o)/L_o \tag{2.2}$$

When plotted, these values comprise what is known as the stress strain curve, from which material properties may be calculated. Figure 2.3 illustrates characteristic tensile stress strain curves for mild steel, copper, and aluminum specimen. Strain is plotted on the x-axis, while stress values are plotted on the y-axis. For most metals, the initial portion of the stress strain curve is linear, and is described as the elastic region. This region follows

Hooke's Law, as defined in (2.3), for a uniaxial stress state. The specimen will not experience permanent deformation unless the applied load exceeds the elastic limits.

$$\sigma = \varepsilon * E \tag{2.3}$$

σ and ε have previously been defined, however, E is Young's Modulus of Elasticity. It is defined as the slope of the linear portion, and is a ratio of stress to strain. When the load exceeds the maximum limit, the stress strain curve will start to bend over and the specimen will start to experience permanent deformation. Yield strength, or yield stress, and ultimate tensile strength (UTS) values may also be determined from the stress strain curve after it bends over outside the linear region. For ferrous metals the yield stress is typically a well-defined point, where it can visually be seen that the linear region has reached its maximum limit.

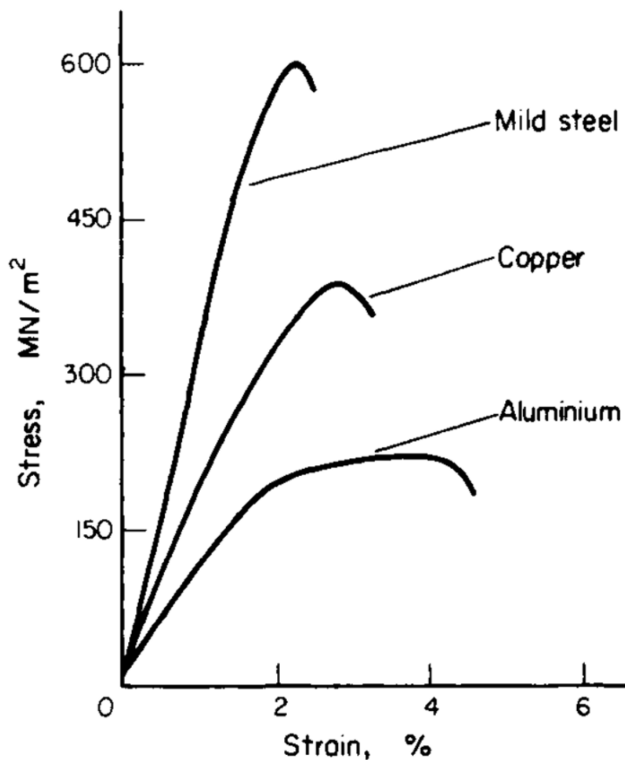


Figure 2.3: Stress Strain Curves of Various Materials, from [12]

However, for non-ferrous metals this point is less evident and is defined by extending and offsetting the linear slope by 0.2%. This is called the 0.2% yield strength (YS). Because the research conducted for this thesis focuses on applications specific to aluminum alloys, YS will refer to the 0.2% yield strength unless otherwise specified. UTS calculations remain the same for both ferrous and non-ferrous metals, and are defined as the maximum stress, experienced by the specimen. For Figure 2.3, the highest point of each curve is used to generate UTS.

Depending on the device used to record displacement measurements, stress and strain calculations may be categorized as pseudo, engineering, or true. The label “pseudo” is used for curves with strain values calculated using displacement values measured by the testing frame’s linear variable differential transformer (LVDT). When the LVDT is used, it is assumed that the majority of displacement in the load train occurs in the specimen gauge section due to its reduction in area, or the material’s response to elevated temperature. However, there is some minimal amount of strain occurring in the load train outside the specimen gauge section. Thus the curve is labeled as “pseudo” stress and strain. Engineering, and true stress-strain curves both measure displacement relative to the specimen gauge section, but calculate stress using different methods. Engineering stress-strain curves make the assumption that reduction in cross-sectional area is minimal and that stress can be calculated using the original, minimum, cross-sectional area of the specimen. A true stress-strain curve on the other hand, requires the instantaneous calculation of stress and strain. In other words, the deformation in cross section must be measured and used to calculate the stress at any given instant in time for the duration of

the test. It is often difficult to obtain these measurements, and as a result a correlation has been developed relating the linear region of the engineering stress strain curve, in which no deformation occurs, to the true stress and strain, as defined in (2.4) and (2.5). σ_t represents the true stress while ε_t is defined as the true strain for the elastic region.

$$\sigma_t = \sigma * (1 + \varepsilon) \tag{2.4}$$

$$\varepsilon_t = \ln(1 + \varepsilon) \tag{2.5}$$

True fracture strength (TFS) may also be of interest and is the stress at which fracture occurs, as shown by (2.6) [13] where σ_{frac} is the true fracture stress, P_{frac} is the load at failure, and D_{frac} is the specimen diameter at failure.

$$\sigma_{frac} = P_{frac}/(D_{frac} * (\pi/4)) \tag{2.6}$$

Lastly, percent elongation measures the percent with respect to the original gauge length that the specimen deforms during a tensile test taken to failure. The following equation describes how to calculate percent elongation:

$$\%Elong = \Delta L/L_o = (L - L_o)/L_o \tag{2.7}$$

Where $\%Elong$ stands for percent elongation, L is the measured specimen gauge length after failure, and L_o is the original specimen gauge length as is recorded before testing.

2.2.2 TESTING STANDARDS

To homogenize results across industries, the American Society for Testing and Materials, now recognized as ASTM International, was formed. ASTM International publishes standards and recommendations for use in research [14]. ASTM International E21 lists the recommended standards for tensile testing of metals and metallic alloys at elevated

temperatures. Key standards taken into account while developing the elevated temperature tensile testing apparatus are as follows [11]:

- When the specimen gauge length is greater than 1” (25.4 mm), temperature should be measured at two separate locations along the gauge length.
- The temperature gradient along the specimen gauge length should not vary more than +/- 3°C
- At set-point (test temperature) the temperature should not vary more than +/- 3°C

2.3 REVIEW OF AVAILABLE TECHNOLOGY

When performing an elevated temperature test, the selection of a heating device is critical. Often, selection is highly dependent upon the device’s primary mode of heat transfer, although all modes need to be taken into account. For example, furnaces or ovens typically used in conjunction with testing frames provide the majority of heat transfer through radiation, but the heating elements are staggered vertically throughout the oven to account for natural convection. Other factors to consider include geometry of the heating device, limits of the mechanical testing apparatus, desired heating rates, process control, and the device’s responsiveness to a change in temperature. The following section introduces several high temperature testing facilities and the technologies available at each.

2.3.1 GLEEBLE

The U.S. Army Armament Research Development and Engineering Center, Benét Laboratories, used a Gleeble to assess the solid/liquid embrittlement of gun steels by copper. Three types of steels in un-notched and notched form were pulled to failure at a

stroke rate of 0.127m/sec. Temperatures ranged between 868°C and 1,100°C and specimen were heated to temperature in 3 seconds and held for a soak period for 10 seconds. The tests found that embrittlement of copper plated steels occurred at 1,100°C, but that there was only minimal evidence for it occurring at lower temperatures [15].

The Gleeble systems is distinct in that it combines separately controlled thermal and mechanical testing systems [16] to provide physical simulations, or "the reproduction, on a laboratory scale and in real time, of the thermal and mechanical parameters of a real-world production process [17]." The Gleeble can reach heat up rates of 10,000 °C/sec to perform elevated temperature tensile, compressive, and torsional tests. These heating rates are a result of the Gleeble's direct self-resistance heating mechanism, which passes an electric current through the gauge section while isolating specimen end sections for attachment to the mechanical test frame. The specimen gauge section heats up due to the resistant nature of its material. Time, applied current, material resistivity, and geometry all influence the heating rate of the specimen [16].

Currently three models of the Gleeble exist. Basic applications covered by the Gleeble 3180 include hot tensile tests, continuous casting simulations, weld HAZ simulations, melting and solidification tests, and heat treatment testing. The Gleeble 3500 and 3800 can facilitate these applications, as well as provide more functions and increased capabilities [18]. With increased functionality comes an increased cost. In 2010 the Department of the Navy estimated a total cost of $878,553.00 for either a Gleeble 3500 or 3800 with hot-torsion and HydraWedge capabilities, a required vacuum pumping system, water chiller, high flow

quench system, mounting gauge spacer clips, strip heaters, heat control shims, installation of the system, and training of personnel [19]. The University of Cape Town in Southern Africa also purchased a Gleeble 3800 in 2012 that cost approximately $900,000.00 [20].

2.3.2 INDUCTION HEATERS

BAM Federal Institute for Materials Research and Testing employs an induction heater for the thermo-mechanical testing of ceramic matrix composites. Specimen can be tested in a vacuum or inert gas at temperatures up 1,700°C or in air up to 1,500°C to determine properties such as tensile strength, stiffness, and elastic/plastic deformation behavior. The testing apparatus consists of a 100 kN servo-mechanical testing machine coupled with an integrated chamber. Specimens inside the chamber are heated by radiation through the induction heating of susceptors surrounding the test piece [21].

Induction heaters are commonly used to heat conductive materials, such as metal, for use in metalworking, heat treating, welding, and melting; however, they can also be modified for the heating of non-metallic materials. The induction heater operates by inducing electrical currents within a metal, or work piece, using an induction coil, metal work piece, and an alternating current power supply. The induction coil is typically made out of copper tubing, to enable water cooling, and is formed in coils around the work-piece. An alternating current is passed through the coil to generate an alternating magnetic field. This field induces an electric current, or eddy current, which raises the work-piece temperature through Joule heating. Joule heating occurs when the work-piece’s natural

resistance to electrical currents produces heat. Ferrous materials, such as iron, respond best to induction heating because they are ferromagnetic [22].

Effectiveness of the induction heater relies heavily upon geometry of the work-piece and induction coil, work-piece material, and magnitude and frequency of the applied ac current. When designing the induction coil no set standards govern the design, and most are formed based from experience. Thus design of appropriate induction coils takes time and can be very costly [22]. Induction heaters themselves are not inexpensive and a powerful heater can cost anywhere from $6000.00 and up [23].

2.3.3 ELECTRIC HEAT

Both the NASA Langley Research Center and Idaho National Laboratory (INL) employ electric furnaces for material property testing [24], [25]. INL uses high temperature furnaces that can achieve temperatures of 3,000°C and autoclaves to test materials used in nuclear reactor core and support structures [25]. NASA's Langley Research Center uses furnaces during elevated temperature tensile testing of foil-gage metals at temperatures of 500°F [24]. Other facilities also use furnaces for elevated temperature testing. A collaborative effort between Texas A&M University at Qatar, American University of Beirut, the University of Michigan, and the University of Lille-North of France was published in *Materials Science & Engineering* detailing a study of AA 6061-T6 under various strain rates and temperatures. The testing process describes the detailed use of a computer controlled MTS Insight electromechanical testing machine equipped with a LBO-series Thermocraft LabTemp laboratory oven (environmental chamber). Tensile testing of the 6061-T6

specimens was conducted at temperatures up to 300°C with specimens kept in the environmental chamber for 30 minutes before starting the test to ensure a homogenous temperature [10].

Electric heaters, such as those used in laboratory furnaces, do not heat the work-piece by induction or resistance heating, but rather transfer heat to the work-piece via one of the three modes of heat transfer. The heater itself is heated by direct resistance heating, and that heat is transferred to the work-piece by some combination of conduction, convection, or radiation [26].

Electric heaters come in a variety of forms, such as strips, cables, and tubes. Band heaters, a type of electric heater designed for extruders, were explored as a possible electric heating source for the developed testing apparatus. The band heaters are cylindrical, meet elevated temperature requirements, and cost around $75.00 [27]. However, due to geometry limitations that restricted placement of the band heaters, responsiveness of the system was significantly limited. Thus, required heating rates and system stability could not be achieved.

2.3.4 PROCESS CONTROL: PID

PID control is a very common process control method used in a variety of industries. Figure 2.4 illustrates a generic feedback control loop block diagram that shows the implementation of a PID controller within a process. Simply explained, the process begins at some initial point and is adjusted by a control element until a sensor indicates that it has reached the set-point (SP). The set-point is some target value for the process, often a

temperature, pressure, or volume, which is continuously monitored by a sensor. Readings from the sensor are known as process variable (PV) readings, and are reported back to the PID controller. The controller then compares the PV values to the SP to calculate an error which dictates the magnitude of change that is to be made to the control element. This progression repeats until the SP and PV are equal to one another or within acceptable bounds [28].

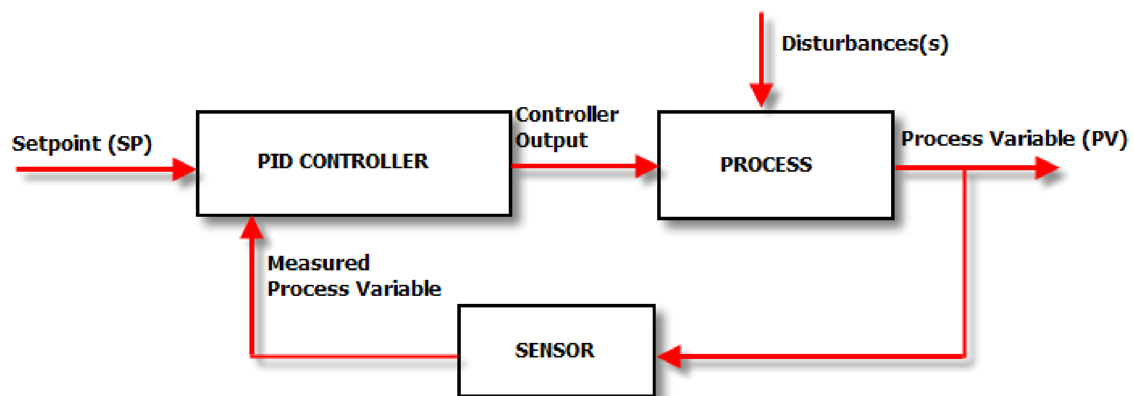


Figure 2.4: Process Control Overview, from [29]

Before designing the PID controller it is important to understand how a process system functions. PID controllers can be designed to provide positive or negative feedback, and be direct or reverse acting. Positive feedback is given by a controller that is programmed to enhance the error between the PV and SP. In other words the PV gets driven further away from the SP until process limits are reached. This type of feedback is not conducive for use with automatic controllers, where negative feedback is preferred. Negative feedback leads the controller to issue outputs that minimize the PV and SP error, increasing the stability of the system. A direct acting controller is one for which an increase in process inputs results

in an increase in the PV. Likewise, a decrease in system inputs should result in a decrease of system outputs. For a reverse acting controller the opposite occurs. When inputs to the system are increased, the outputs see a decrease in value, while for a decrease in inputs the outputs will see an increase in value. Many systems and most PID controllers are direct acting with negative feedback [28].

Looking inside the PID controller it can be seen that there are actually three modes of control: the proportional (P), integral (I), and derivative (D). While P and I control modes can be used alone, the D mode is almost never used in this fashion. P, I, and D can also be combined together. The most common control modes are P, PI, and PID [30].

Proportional control, or gain, calculates a change to the process that is proportional to the error between the PV and SP. It does not rely on past values of the PV nor does it take into consideration the rate of change of the error. Thus P mode is computationally simple and easy to tune as there is only one input to alter. However, it has a downside: an offset typically exists between the SP and PV for most loading conditions. In other words, for a given loading of the system the controller may be able to bring the PV to within the SP bounds. For this same system and loading an additional disturbance within the process may create an offset within the system. This occurs when the controller's abilities are not complex enough to handle the disturbance. Manually we can account for this by adding an output bias to the P controller calculations, as seen in the following equation:

$$MV = K_c * e + b \tag{2.8}$$

For (2.8), MV is the manipulated variable or controller output, K_c is the controller gain, e is the error between the SP and PV, and b is the output bias. When the output bias is a predetermined constant, as is the above case, it is defined as the “manual reset,” and is altered by the user [30].

To automate the process of adjusting the bias, the Integral mode may be introduced. Because the integral term automatically adjusts the bias, it is often referred to as “automatic reset,” or just “reset.” Addition of the integral term to controller calculations allows the controller to account for past errors in its outputs. It does this by summing the past errors to determine whether or not the MV needs to be increased or decreased. If the sum of errors is positive this indicates that the MV needs to increase, while if it is negative then the MV should be decreased. Calculations for the PI controller are shown in (2.9).

$$MV = K_c * (e + \frac{1}{T_I}\int e\,dt) \tag{2.9}$$

In (2.9), T_I is the integral time with units of minutes per repeat. If a fast integral response is desired, mathematically the T_I inputs should be small. Conversely larger T_I inputs will result in slower response times. This can be confusing, which is why some controllers are designed to accept inputs of $1/T_I$, for which a large input results in fast response times, and a small input produces slow response times [30].

Additionally the derivative term can be added to further improve performance of the PID controller as shown in Figure 2.5. Derivative control is based on the rate of change of the product of the controller gain (K_c) and error, allowing for the controller to predict system responses into the future. The mathematical addition of the D mode is shown in (2.10).

$$MV = K_c * (e + \frac{1}{T_I}\int e\,dt + T_D * \frac{de}{dt}) \tag{2.10}$$

The tuning parameter, T_D, is the derivative time with units in minutes. As mentioned above, one of the benefits of the D mode is its ability to predict system responses. It also allows the controller to respond more quickly to disturbances in the system load. Sometimes, however, this can be of disadvantage. For especially noisy systems, D mode will actually amplify the noise causing amplified controller responses and an unstable system [30].

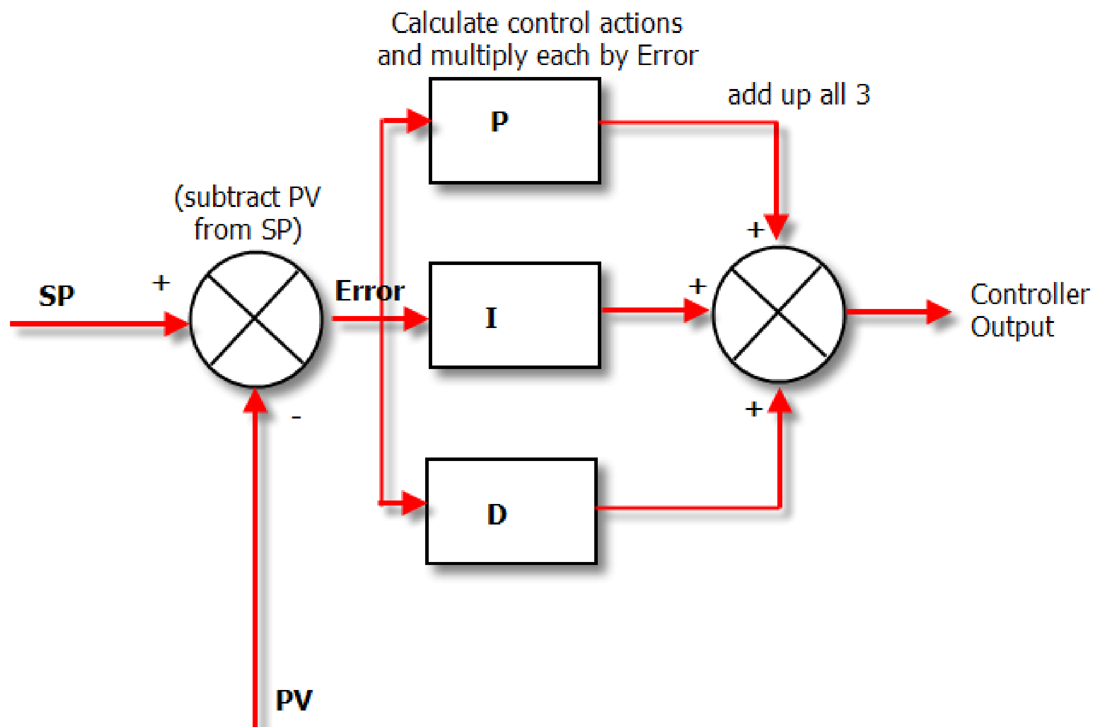


Figure 2.5: PID Control Implementation, from [29]

Equation (2.8) is an example of a traditional PID. This is defined as a controller with a gain that is multiplied to through all three modes of control. Alternatively, the "parallel" form of PID allows an independent gain for each mode of control as shown in (2.11).

$$MV = K_P * e + K_I * \int e\,dt + K_D * \frac{de}{dt} \tag{2.11}$$

This allows the user to fine tune each parameter. A correlation between the traditional and parallel forms of the PID can be seen as follows:

$$K_P = K_C \tag{2.12}$$

$$K_I = \frac{K_C}{T_I} \tag{2.13}$$

$$K_D = K_C * T_D \tag{2.14}$$

Where K_P is the proportional gain, K_I is the integral gain, and K_D is the derivative gain. These variables have perhaps less physical meaning with respect to their units, but they do allow for a more simplified tuning process [30].

CHAPTER 3: TEMPERATURE CONTROL & TENSILE TESTING APPARATUS DESIGN

Components of the overall testing apparatus design will be discussed in the following sections. Section 3.1 will detail the interaction of the system's mechanical components, 3.2 will discuss software design, and 3.3 will compare the costs associated with the developed system to those required by other elevated temperature testing platforms. Mechanical components include specimen geometry, grip geometry, tensile testing equipment, displacement measurement, thermocouples, shields, propane torch support fixtures, stepper motors, the coupling device, torque arm, pressure regulator, and the specific electronics incorporated into the design. Design of the software includes the use of a programmable logic controller set to respond to a PID control loop, and the implementation of a graphical user interface (GUI). Costs for the entire system are limited to hardware purchases, and manual labor associated with manufacturing.

3.1 MECHANICAL COMPONENTS

The mechanical components of the system ensure heating of the test specimen for the duration of a tensile test. Specimens are suspended in the servo-hydraulic testing frame through use of a multi-axis gripping system while heat is applied, via flame, by two propane torches aimed at opposite ends of the specimen. Temperature is varied by individually changing the flame intensity of each propane torch. This is accomplished by activating a stepper motor to regulate the single stage pressure regulator housed within the torch handle. Each stepper motor is operated by the temperature control system which receives

inputs from two thermocouples attached to the specimen. An image of the laboratory set-up is shown in Figure 3.1, while Figure 3.2 displays a diagram of system interactions.

3.1.1 SPECIMEN GEOMETRY, GRIP GEOMETRY

Specimen and grip geometry are interdependent aspects of the tensile testing process: specimen geometry may dictate grip design or grip design may dictate specimen geometry. For the purpose of this thesis, specimen geometry was selected first, and grips were designed second. Figure 3.3(a) illustrates the selected round specimen geometry. The round specimen has button head ends measuring 12 mm in diameter, a stepped section with a diameter of 8 mm, and a gauge section diameter of 6 mm that is gradually reduced by 0.2 mm to ensure failure at the center of the specimen. Note that the button head ends are filleted down to the reduced section. This fillet was designed to seat into a

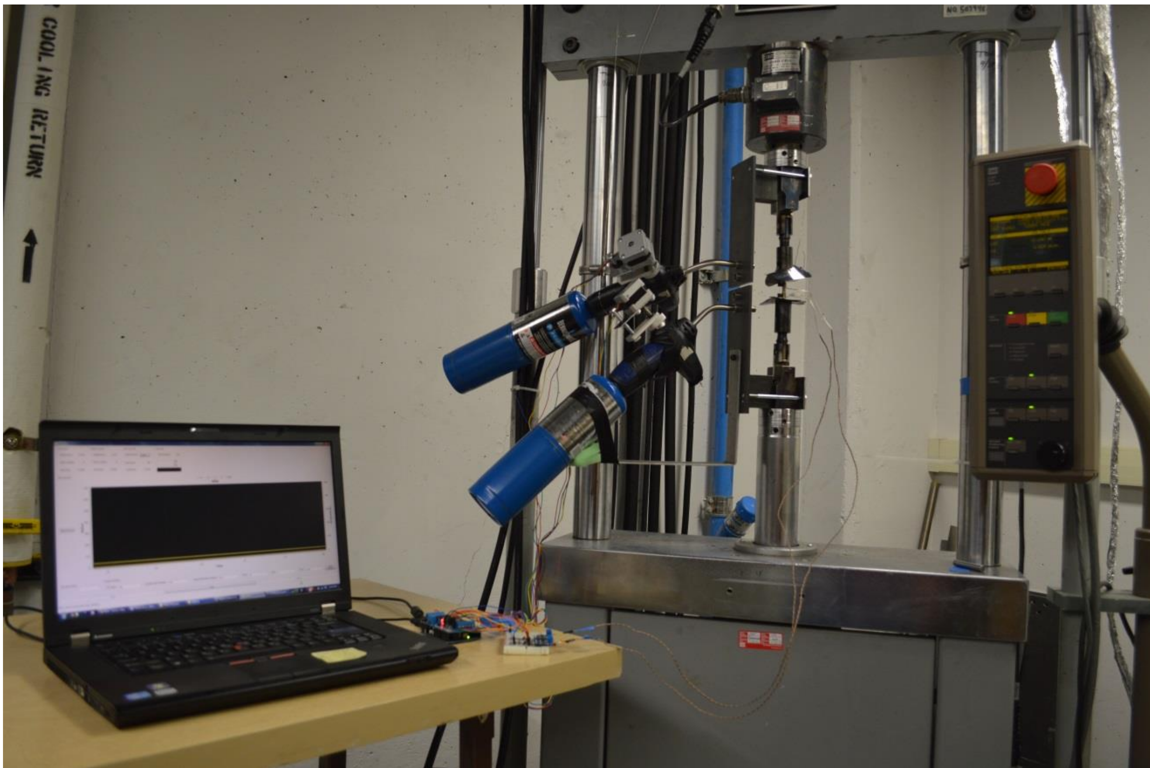


Figure 3.1: System Setup

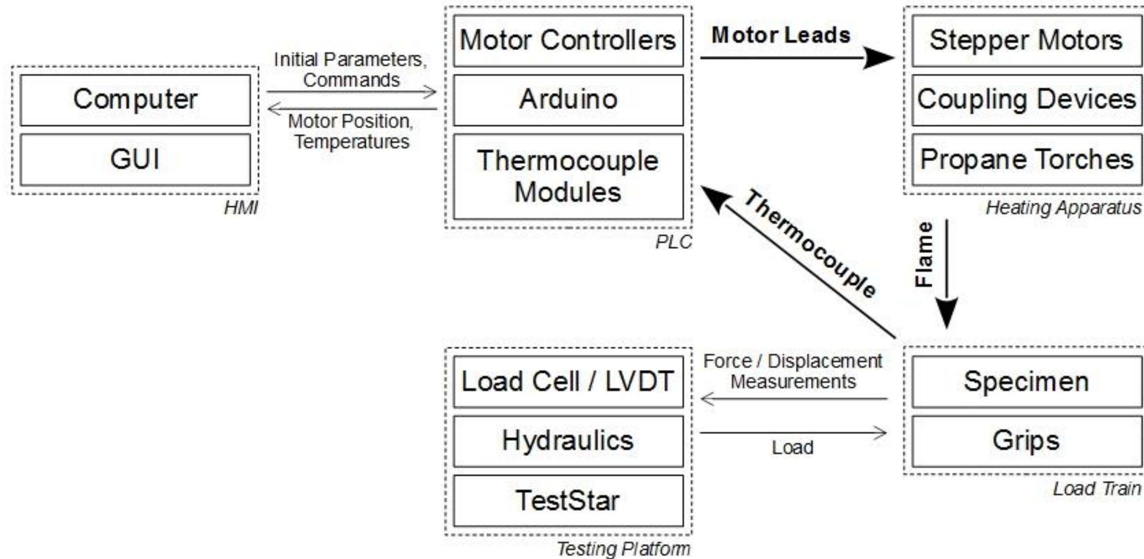
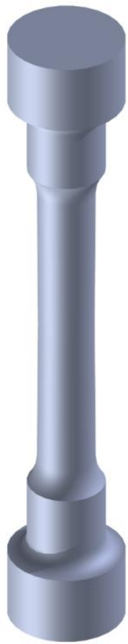
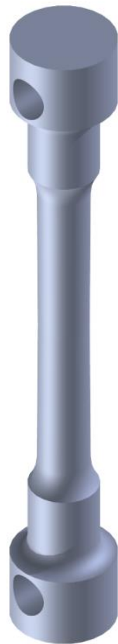


Figure 3.2: System Overview

correspondingly filleted grip. Alternatively, the button head ends may be drilled through to accommodate a pinned connection as shown in Figure 3.3(b). The load train implementing a seated grip connection is illustrated in Figure 3.4(a), while the pin connection grip option is illustrated in Figure 3.4(b). Figure 3.4(a) and (b) also show a threaded connection part, a clevis grip, and a square pinned connection at the top and bottom of the load train. The square components thread into the MTS servo-hydraulic testing frame, while the threaded connection part allows for vertical adjustment of the specimen position, and the clevis grip adds extra degrees of freedom. These extra degrees of freedom help rectify any minor misalignments that may exist in the load train. If displacement measurement devices are going be used, grip geometries should also be checked to avoid interference with the device's attachment point and measurement probes.

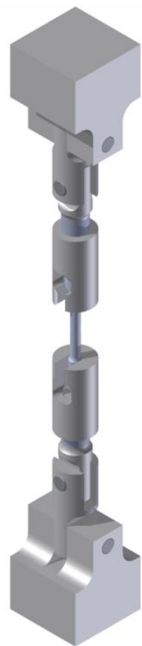


(a)



(b)

Figure 3.3: Specimen Geometries



(a)

(b)

Figure 3.4: Load Train Configurations

3.1.2 TENSILE TESTING EQUIPMENT

During development, all materials testing and validation was performed on an MTS servo-hydraulic test frame, model 312.21. This test frame has an actuator with a total stroke of 200 mm, and includes a MTS model 661.21 load cell with a maximum operational capacity of 100 KN (10 metric tons). The MTS frame is operated by a TestStar II control system in either load or displacement mode, with testing procedures written and performed via Testware SX V4.0A software (TWSX). Displacement of the actuator is measured by the LVDT. It is important to note that measurements obtained using the LVDT refer to the displacement of the entire load train and are not specific to the specimen gauge section. If necessary, displacement of the specimen can be measured to provide more accurate strain calculations. This is accomplished using an MTS model 633.11B20 extensometer with a range of +/- 0.15” or 3.81 mm, and a circuitry housing temperature limit of 200°C. To avoid damaging the extensometer, TestStar interlocks may be set to trigger a shut off of the test frame hydraulics when the extensometer reaches a reading of +/- 2.5 mm. Thus the extensometer can safely accommodate a total displacement of 5 mm, which correlates to a 14.3% elongation of the specimen described in Section 3.1.1 . Depending on the elongation of the material being tested, failure may or may not occur before the MTS model 633.11B20 extensometer reaches its limits. Other extensometers that accommodate greater elongations may be implemented, however, were not available for use during the research performed in this thesis.

Tensile testing performed for this thesis measured displacements using the LVDT; however, the testing apparatus was also designed to accommodate an extensometer. Due

to the extension and temperature limits of the available extensometer, a set of extended arms were designed for the device. These arms doubled the extension range of the extensometer, as well as moved the extensometer circuitry away from the heated region. However, increasing the extension range halved the number of recorded data points. When using the extended arms, Equation (3.1) should be used to convert extensometer readings to actual values:

$$Ext_{Actual} = 2.5626 * Ext_{Reading} - 0.0008 \tag{3.1}$$

Where Ext_{Actual} is the actual displacement of the extended arms, and $Ext_{Reading}$ is the corresponding measurement recorded by the extensometer. The extended arms work well if the elastic region of the material being tested is of sufficient length to record enough data points for an accurate linear fit correlation. However, this was not the case during testing of the 70XX series aluminum alloy, due to its low yield strength. Also of concern were the effects of momentum and inertia acting on the extensometer circuitry housing when using the extended arms at high strain rates. This is because the mass of the housing is located far enough from the attachment point that it cannot withstand the momentum generated and causes inaccurate data collection.

Due to these failings the original extensometer set-up was reexamined. It was found that the circuitry was far enough away from the heated zone around the specimen to be used for tests up to 400°C. Monitored temperature tests were performed at 300°C and 400°C with results displayed in Table 3.1. It should be noted that shield design also contributed to the extensometer temperature. The cone shaped top shield used with the seated grips

directed heat from the flame towards the extensometer more than the pinned connection grip shields. Because the temperature recordings reaching upwards of 150°C during ramped heating, it was determined that testing specimens above 400°C while using the original extensometer set-up would be detrimental to the extensometer circuitry.

Table 3.1: Extensometer Temperature Monitoring

Temperature (°C)	Time (sec)	Extensometer Housing Temperature (°C)	Notes
300	120.0	55.0	At some points during the heating process, both tests displayed extensometer temperatures of 150°C
400	120.0	87.0	

3.1.3 THERMOCOUPLES AND SHIELDS

Temperature regulation is monitored through the use of type K thermocouples and is essential to the operation of the temperature control system. In accordance with ASTM standard E21, the thermocouples were attached at each end of the specimen to measure the temperature across the gauge section of the specimen. The thermocouples were attached using clips made of 301 stainless steel strips, 0.008" thick x 0.25" wide. These were fastened by 2-56 x 0.25" socket head cap screws and nuts as shown in Figure 3.5. The design of these clips was selected to emphasize a low profile, producing a minimal heat sink. However, if the material being tested elongates and causes a significant reduction in cross-sectional area, the diameter of the specimen can become smaller than that of the attachment clip, which in turn can result in the thermocouple "popping out" of the clip or a reduction in contact with the specimen. Alternatively, welding thermocouples to the specimen was considered. However, this was deemed too extensive a task due to the poor welding characteristics of aluminum and the quantity of welds necessary for completing all

of the tensile tests. A secondary method consisting of an alligator clip with a potted thermocouple was also considered. The alligator clip was filed to have a curved attachment point so as not to mar the specimen surface, and was potted with Resbond 906 High Expansion Adhesive, an electrically resistant adhesive/potting agent. Although the alligator attachment devices addressed the reduction in cross-sectional area, they proved to be too great of a heat sink and increased heating rates. A solution was found by moving the thermocouples and clips further out of the specimen gauge section until they were in contact with the 5 mm radius shoulder. This solution worked as long as deformation occurred in the center of the gauge section.

Testing was performed with thermocouples located 27 mm from each end of the test specimen. This ensured enough room in the specimen gauge section for attachment of an extensometer, if desired, while still monitoring temperatures at the ends of the specimen gauge section. As seen in Figure 3.5 the 27 mm was measured from the end of the specimen button head to the closest edge of the thermocouple clip, with thermocouple beads positioned at the approximate mid-point of the clips. Symmetry of thermocouple placement is important if a symmetric and accurate temperature profile is to be accomplished. Because the specimen is orientated vertically, natural convection affects the gauge section heating profile. With symmetric thermocouple placement this discrepancy in temperature becomes significantly apparent, and changes to the system that would offset the temperature discrepancy are more easily monitored. These changes to the system may include torch flame position relative to the specimen button heads, and flame intensity. Flame position may be altered by moving the torch nozzle closer or farther away from the

load train, changing the torch nozzles vertical position, or adjusting how many threads are exposed at the grip connection point in the load train. Flame intensity may be varied by changing the maximum motor positions, as well as adding fans to remove heat at lower temperature settings.

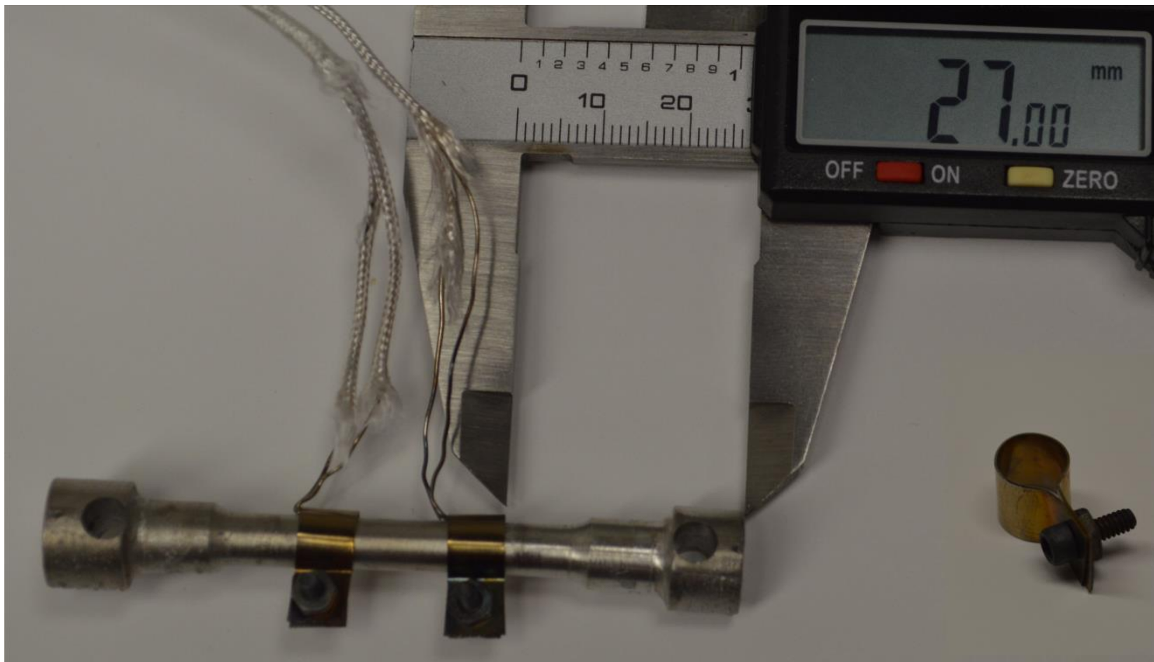


Figure 3.5: Thermocouple Clip Attachment

The thermocouples were particularly sensitive to direct flame, resulting in the addition of shields to the load train. Implementation of the shields helped guard against erroneous temperature readings, as well protected the specimen gauge section from direct flame and hot spots. Shields were located between the propane torch flame and nearest thermocouple attachment point and were supported by the grips. For the seated connection grips, the top shield was manufactured out of 0.006" shim stock which wrapped around the grip in a cone shape and was secured by a pinned connection. It was initially tight enough to support itself, but became loose with use and eventually required a

wire hook to hold it up. The lower shield consisted of two 0.010" flat, square shim stock pieces with slots cut to fit around the specimen. These shields rested flat on the top of the bottom grip. A second set of shields was designed for the pin connection grips. A comparison of the three shields can be seen in Figure 3.6 and Appendix A displays the drawings for these shields.

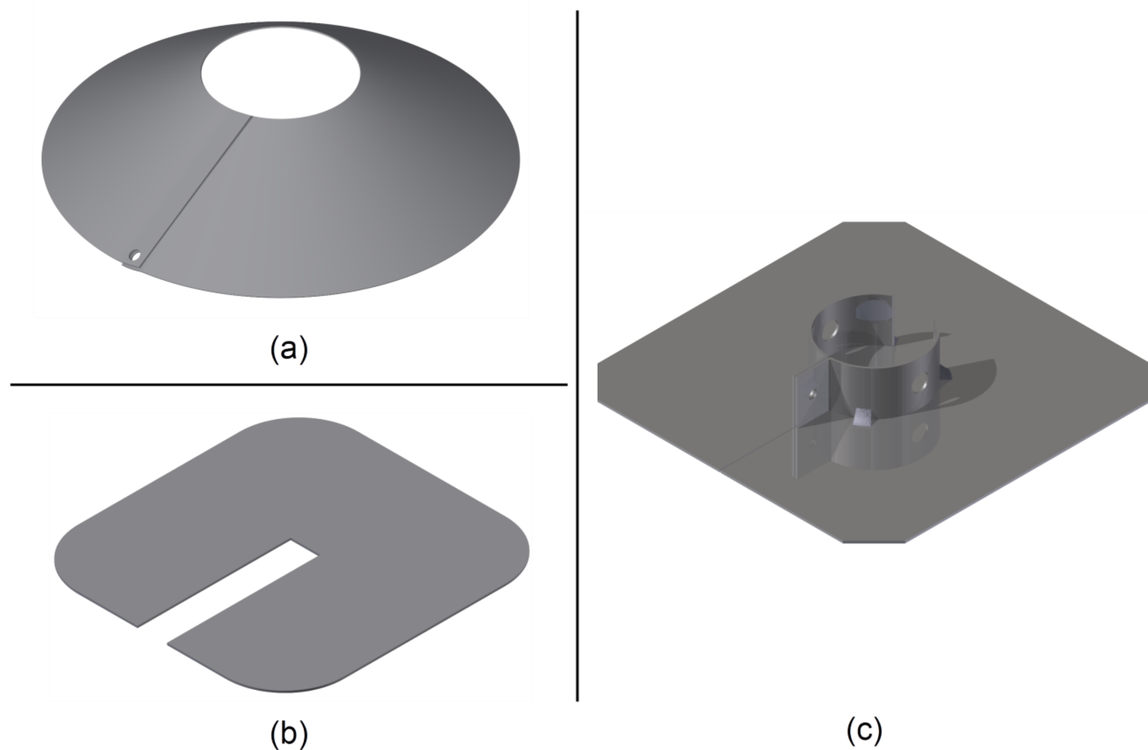


Figure 3.6: Comparison of Shield Types

3.1.4 TORCH SUPPORTS

As mentioned in the previous section, location of the propane torch nozzles is critical to achieving a uniform temperature distribution along the specimen gauge section. The purpose of the torch supports is to provide an adjustable fixture for the torch nozzles so that heat may be applied to the specimen button heads. For both the seated and pinned connection grips, torch nozzles are radially positioned in line with the center of the grip

openings. Location of the torch nozzle relative to the button head in the z and x direction can greatly affect temperatures in the gauge section, and may be adjusted to account for natural convection. A uniform gauge section temperature was accomplished for the seated grips by locating the top torch nozzle at a distance of 1.68” horizontally from the top grip opening, and a distance of 0.60” vertically from nozzle center to the bottom of the top grip. The bottom torch nozzle was located 1.44” horizontally from the bottom grip opening and 0.48” vertically from nozzle center to the top of the bottom grip. For the pinned grip set-up the top torch nozzle was located 1.68” horizontally from the top grip opening and 0.445” vertically from nozzle center to the bottom of the top grip. The bottom torch nozzle was located 1.44” horizontally from the bottom grip opening and 0.92” vertically from nozzle center to the top of the bottom grip. Appendix A contains drawings detailing the locations described above

The torch nozzle locations were thoroughly tested to ensure that the temperature profile along the length of the specimen was within the acceptable limits of +/-3 degrees. Once these locations were found, fixtures were manufactured to support the torch nozzles. These are comprised of steel plates press fit with stainless steel bushings through which the torch nozzle can slide and adjust position. The steel plates are bolted to the square grips of the MTS test frame—one just below the load cell and another located on the actuator. These attachment points allow both torches to stay centered on the button head as long as the load train is not rotated out of alignment. An L shaped bracket was added to the bottom torch support to create a rigid member in support of the propane cylinder. This cautionary feature was added to prevent damage that could occur from the momentum

and impact of the actuator moving at high strain rates. A safety wire sling was added to the top torch as a secondary support point to relieve stress from the top nozzle. Figure 3.7 provides a detailed illustration of the torch supports, while drawings can be viewed in Appendix A.

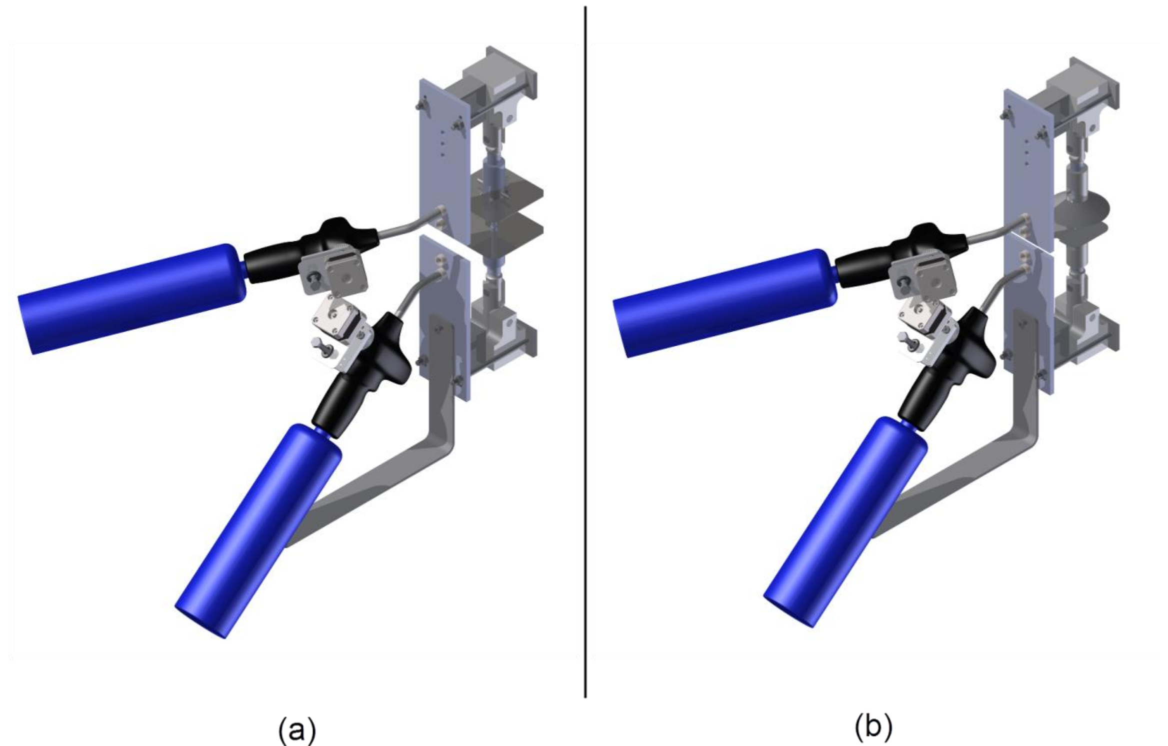


Figure 3.7: Torch Supports, (a) Pinned, (b) Seated

3.1.5 AUTOMATED CONTROL OF THE NOZZLE

Automated control of the propane torch nozzle consists of a stepper motor, torque arm, coupling device, and a single-stage pressure regulator (SSPR, also known as a diaphragm valve). The stepper motor is used to open the SSPR by turning a threaded connection within the coupling device to open the valve. The torque arm was implemented to align and maintain motor position. Figure 3.9 provides an illustration of the torch/motor assembly.

Ignition of the propane torches requires fuel flow at or above a certain level. On an original Bernzomatic TS3000 this is accomplished by turning the torch handle knob clockwise until the ignition button can trigger a sustained flame. Flame size can then be regulated by the torch handle knob; however for the purpose of this research it has been replaced by a coupling device and stepper motor to automate the process. A CanaKit STM100 stepper motor was selected for automation because of its relatively high torque, positional feedback, and unbounded shaft movement. The use of a DC or servo motor was also considered, but servo motors are typically limited to only 180° of shaft rotation and DC motors cannot provide positional feedback without additional hardware. The CanaKit motor is rated for 0.23 N-m of torque, requires a 12 V power supply, and has a 1.8° step angle (200 steps/revolution). The 200 steps/revolution provides enough resolution to satisfy the temperature control system requirements, and the torque is great enough to operate the torch valve without stalling at low fuel flows. One downfall of the stepper motor is that if a torque greater than what it is rated for is applied to the motor, it will stall out and lose its position relative to the valve. Because of this feature it is necessary to define an origin, a minimum, and a maximum motor position. The origin refers to the zero step point at which there is no flame, while the minimum point has been defined as the location where the torch can maintain the smallest flame possible without blowing out. This position is necessary for setting lower motor limits in the system software code. For implementation in this research, the bottom motor minimum position was set at 203 steps; while for the top motor was set at 204 steps. The maximum motor position is a user defined input to the GUI so that it may be changed to satisfy varying heat rates. However,

it is important to acknowledge that an upper limit for the motors exist at the position where the motor begins to stall out. This position will vary depending on motor ratings, and the resistance of the valve compression spring. This location was found to be at 700 steps for both the top and bottom motor, however, a 50 step buffer was reserved and upper limits were set at 650 steps. A fourth position, the ignition point, was defined at 600 steps, or 3 rotations of the motor shaft, for both motors. This location provides enough propane for a successful ignition mixture. At the start of the temperature control program, the motors ramp up to a position of 600 steps, at which point the user is signaled to ignite the torches. After ignition the motors will ramp up or down to the GUI defined maximum motor positions until the PID loop takes action.

By turning the torch handle knob the SSPR can be adjusted. The SSPR allows for the regulation of fluid flow through a valve. Typically this fluid is a gas, as is the case for the Bernzomatic TS3000. The inner workings of a SSPR can be seen in Figure 3.8(a), and consist of a loading mechanism, sensing element, and control element. The loading mechanism is comprised of a pressure adjustment knob which applies a force to the compression spring. As the spring compresses it transfers this force to the sensing element, or diaphragm. If the force is great enough the control element will be activated, opening the orifice by unseating the poppet. This allows gas to flow from the high pressure inlet through the low pressure outlet until equilibrium is reached. In total, four forces act on the diaphragm and contribute to the regulation of fluid flow. As Figure 3.8(b) depicts, a downward force is applied to the diaphragm by the compression spring, while the inlet gas, outlet gas, and bonnet all apply upward forces. When the compression spring force is greater than the

combined gas pressures and bonnet force, it deflects the diaphragm downward. This forces the bonnet off its seat, opening the orifice, and allows fluid to flow from the high pressure region to the low pressure region as seen in Figure 3.8(c) [31].

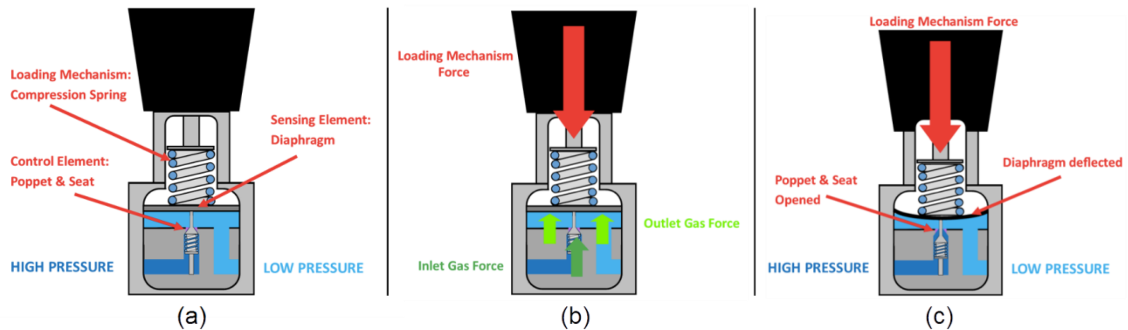


Figure 3.8: Single Stage Pressure Regulator, adapted from [31]

Diaphragm valves are designed to regulate outlet pressure, despite decreasing high-side pressures, such as would be the case in the propane cylinder. During the course of this research, it was found that outlet pressure and flame size remained constant for a given valve setting, unless the propane cylinder was nearing empty. This only became problematic during prolonged testing, and solutions and improvements for this aspect of the design are provided in CHAPTER 6:.

Another issue presented by the valve is that vertical movement is required to compress the valve spring against the diaphragm. Although the motors provide adequate rotational movement, they do not provide linear movement to compress the valve spring. This was solved by designing an aluminum coupling device to interface between the stepper motors and pressure regulator. A flat spot was ground into the section of motor shaft that slides into a hollowed out section of 0.375” OD all-thread. The shaft is secured by a set screw, and the all-thread is inserted into the center of the coupling device. Thus, rotation of the

motor shaft threads the all-thread in and out of the coupling device. Also housed in the center of the coupling device is the original SSPR compression spring which interfaces with the all-thread via a ball bearing to reduce friction and torque on the motor. The other end of the spring is fitted with a brass cap that presses against the diaphragm and bonnet when the spring is compressed. The coupling device connects securely onto the valve chamber by threaded connection, replacing the original brass fitting and plastic knob of the torch handle. Figure 3.9(b) provides an exploded view of the motor assembly for which drawings can be viewed in Appendix A.

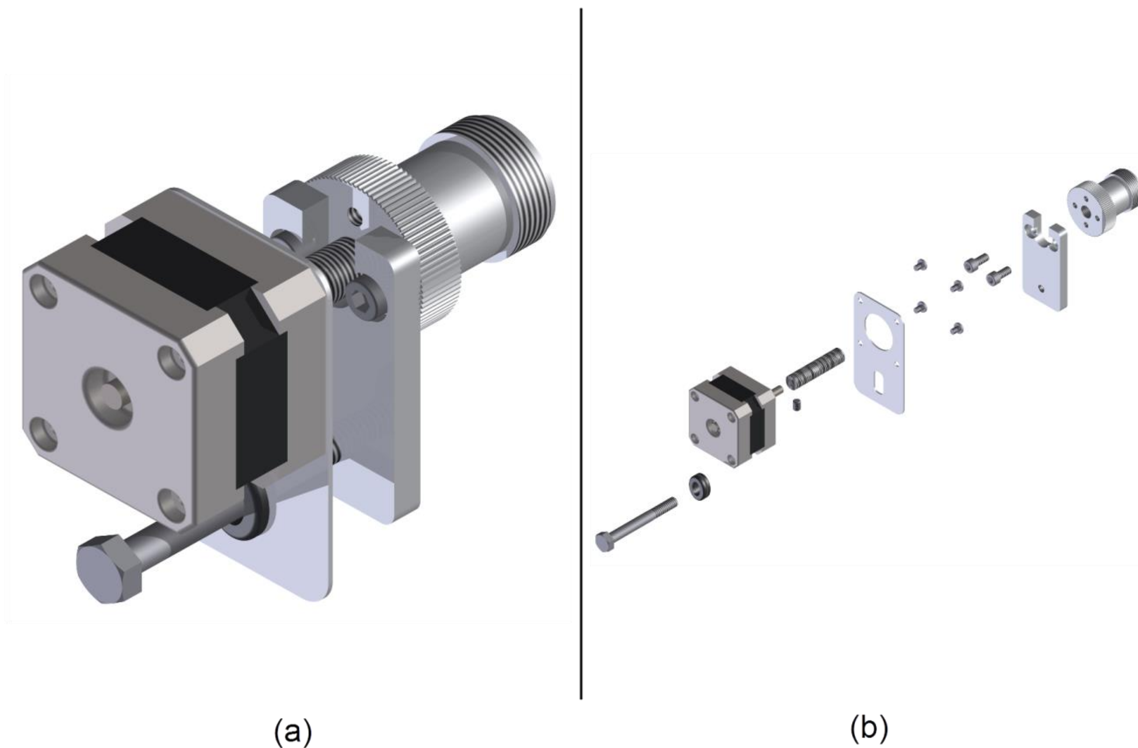


Figure 3.9: Motor Assembly

Also attached to the motor assembly is a torque arm. This feature was designed to keep the motors aligned to the torch nozzle handle, allowing the motor shaft to rotate relative to the motor housing, rather than letting it spin freely. The torque arm keeps the motor

housing fixed with respect to the torch handle while still allowing for vertical movement of the valve without stalling the motor. It is comprised of a 3” long, 1/4”-20, hex bolt that fits through a slotted plate attached to the motor housing, and threads into the coupling device. This set up of the torque arm was found to work nicely for tests run in LVDT mode; however, it created too much vibration for accurate testing results with the extensometer. To dampen vibrations, a 0.3125” ID rubber grommet was added to each torque arm set-up.

3.1.6 ELECTRONIC HARDWARE

The electronic hardware is responsible for obtaining and communicating inputs and outputs to and from the PID control loop. Propane torch motor control is achieved by coupling a SainSmart Uno with a SainSmart L239D Motor Drive Shield. Temperature control is regulated by coupling the Uno with two Adafruit Thermocouple Amplifier MAX31855 Breakout Boards. A set-up of the electronic hardware is shown in Fig Figure 3.10.

Simply put, the SainSmart Uno, as shown in Figure 3.11, is a small computer that can be programmed to perform a variety of tasks. It is a clone of the Arduino Uno, an open-source computing platform, comprised of a microcontroller board and an integrated development environment (IDE) for writing software to control the board. Arduinos were originally developed to provide a simple, inexpensive microcontroller platform for students and teachers; however, it has grown in popularity due to the fact that all of the technology behind the hardware and software is available for anyone to view, use, or develop. This expands the device’s target audience from beginning to advanced users, as it allows for the

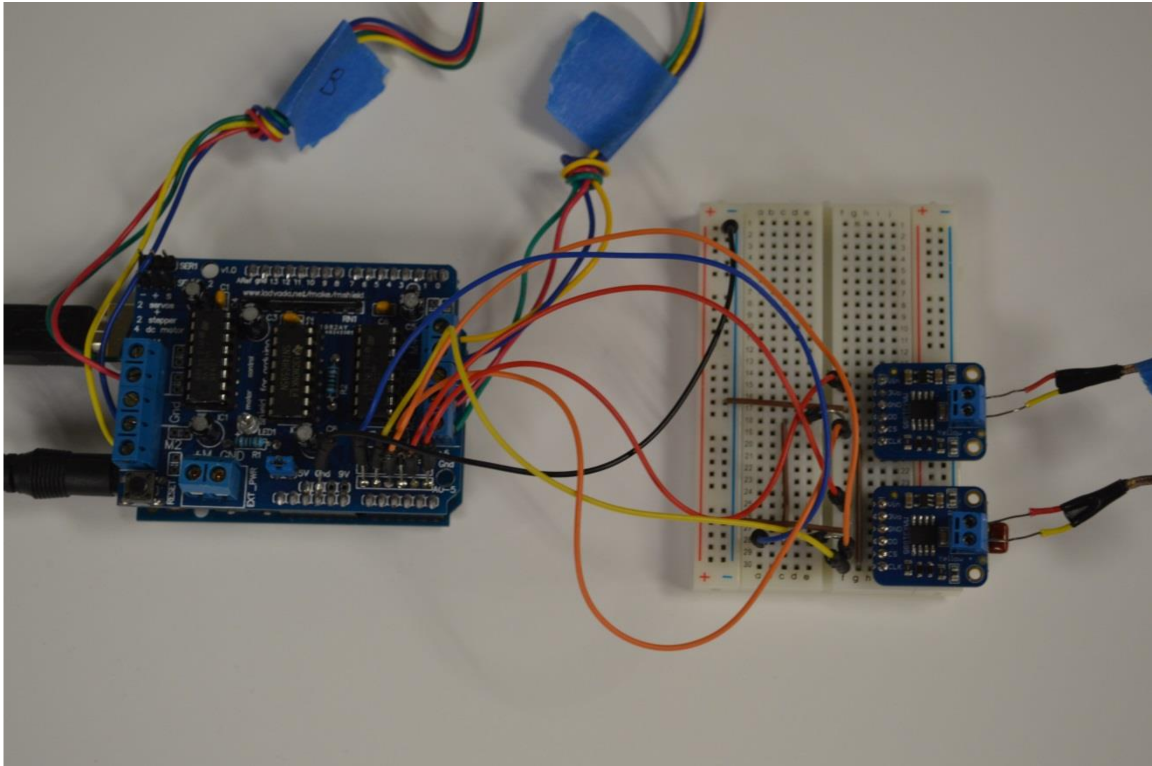


Figure 3.10: Electrical Overview

modification, improvement, and adaptation of the microcontroller to the user's specific needs. Another benefit is that its IDE is compatible on multiple operating systems, requiring only a USB port and cable to program the microcontroller. This makes the Arduino plug-and-play capable, eliminating the need for an external programmer.

Arduino produces a variety of boards, the most common of which is the Arduino Uno. Currently three revisions of the board have been released, however the board used for work in this thesis is a clone of the Uno R2, and it is this version that will be discussed. The Uno R2 features an ATmega328 microcontroller, and has a 16 MHz ceramic resonator, 14 digital input/output (I/O) pins, 6 analog inputs, several pins for ground and power, an in-

circuit serial programming (ICSP) header, USB interface, power jack, and reset button. The USB interface is used for programming, serial UART communication with the computer,

Figure 3.11: SainSmart Uno Microcontroller Board, from [32]

and can be used to power the board. The UART is also available to the digital pins 0 (RX) and 1 (TX). Six of the digital pins can be used as pulse-width modulation (PWM) outputs.

The SainSmart Uno was selected to operate as a programmable logic control (PLC) due to its ease of use and minimal cost. Specs for the SainSmart Uno match those of the Arduino Uno listed in Table 3.2, except for one minor difference: the SainSmart Uno board has a 16

MHz crystal oscillator instead of a 16 MHz ceramic resonator. The SainSmart Uno is also 100% compatible with the Arduino IDE. Because the Arduino programming language is C/C++ based, it allows for sharing within the open source community, expanding resources available to the programmer. Several open source libraries were used to program the temperature control, including one for motor control, thermocouple readout, and PID control. The implementation of these libraries will be discussed later on in the chapter. Another advantage of using the Uno is that the board has USB overcurrent protection and a voltage regulator. USB overcurrent protection provides an extra level of safety in the form of a reusable polyfuse. This fuse protects the user's computer USB ports from shorts and current greater than 500 mA. The Uno board also incorporates a voltage transformer that can step power up or down from the input jack to an operating voltage of 5V. The recommended power input ranges from 7-12V, although the Uno can handle 6-20V. Operating at power levels less than the recommended range can result in the 5V pin outputting below 5V, while operating above recommendations can overheat the voltage regulator, damaging the board [33].

A Sainsmart L239D Motor Drive Shield was purchased to operate the stepper motors. It is capable of running two stepper motors, two servo motors, or up to 4 bi-directional DC motors at once. The motor shield incorporates a L293D 4-channel driver, or H-bridge, to control motor direction. The H-bridge is an electric circuit that allows a voltage to be applied across a load. It contains 4 switches which are activated 2 at a time to apply a positive or negative voltage to the motor which causes the motor shaft to rotate clockwise or counterclockwise [34]. Connection to the Uno occurs by stacking the motor shield on

top of the Uno as shown in Figure 3.10, snapping it into place. Activation and speed control of the motors are linked to the Uno's digital pins. Pins 11 and 3 correspond to the first stepper motor and pins 5 and 6 connect to the second stepper motor. Digital pins 4, 7, 8, and 12 drive the stepper motors via the 74HV595 serial-to-parallel latch, while pin 9 is for servo control of the first motor and pin 10 provides servo control of the second motor [35]. Connection to the motor shield ties up nearly all of the Uno's pins. This could be alleviated by communicating over serial, requiring the implementation of a different motor shield as will be discussed in CHAPTER 6:.

Table 3.2: Arduino Uno Rev2 Specifications, from [32]

Characteristic	Value	Units	Notes
Microcontroller	ATmega328		
Operating Voltage	5	V	
Input Voltage	6 - 20	V	7 - 12 recommended
Digital I/O Pins	14		6 of which can be used for PWM output
Analog Input Pins	6		
DC Current per I/O Pin	40	mA	
DC Current for 3.3V Pin	50	mA	
Flash Memory	32	KB	0.5 KB used by bootloader
SRAM	2	KB	
EEPROM	1	KB	
Clock Speed	16	MHz	

For accurate temperature readings, thermocouples require a good amplifier with a cold-compensation reference such as is provided by the Adafruit Thermocouple Amplifier MAX31855K breakout boards. The MAX31855K Breakout Board was selected for use because it provides both of these functions, and is specifically designed to read the voltage

across K-type thermocouple leads. Voltage is measured across the thermocouple leads, amplified, and converted to digital form to be sent via serial communication to the PLC. However, if the thermocouples are surrounded by a noisy environment, a 0.01μF may need to be added across the thermocouple leads to reduce the noise. Space on the motor shield is limited, and the MAX31855K board is connected to the Uno via jumper cables and a breadboard. Extra circuitry was added to the breadboard to account for the fact that the MAX31855 boards are not compatible with grounded thermocouples. Further discussion on this topic is included in later in the chapter as it directly affects software programming methods [36].

3.2 SOFTWARE SYSTEM DESIGN

The Temperature Control System (TCS) is comprised of two subsystems, the PLC and the Human Machine Interface (HMI). The PLC is responsible for reading temperatures and setting motor positions through the use of two separate PID control loops. The HMI is comprised of a standalone computer running a custom GUI written in Python using the PyQt graphical toolkit. Additional modules are also used to assist with communication and data management. The HMI's primary responsibility is to control the overall process of the TCS, and provide the user with a real time graphical display of time, temperature, and motor position data. After initial communication is established between the HMI and PLC, the TCS is designed to carry out a sequence of events, culminating in the termination of motor movement. Programming of the TCS code is discussed in the following sections, with an overview of the software components listed in Table 3.3

Table 3.3: TCS Software Components

Refs.	Cat.	Name	Description
	TCS_PLC	PID_Motor_Control	PLC sketch
		AFMotor	A class used to control the servo-motors
		EEPROM	Library to manage reading and writing data to EEPROM
		Adafruit_MAX31855	Arduino Library designed to ease the use of the MAX31855 thermo-couple breakout boards
		PID_v1.h	PID control library
	TCS_HMI	GUI_PID.py	Defines the GUI of the HMI.
		com_monitor.py	Defines the ComMonitorThread() class, which resides in its own thread and reads data from the serial port and posts it to LiveDataFeed()
		live_data_feed.py	Defines LiveDataFeed() class. Stores/returns the most recent data from ComMonitorThread()
		PyQt4	Python bindings for Qt4 – a powerful cross platform GUI tool-kit
		eblib	A package that provides access to the computers short and full port names, and all the data in a queue.
	Python 2.7 Standard Library	__future__	Provides future Python 3 changes to Python 2
		sys	Provides access to system parameters used by the Python Interpreter
		Queue	Python's built in Queue which is thread safe. Used for storing data from TCS_PLC
		collections	Python's High performance container datatypes. deque was used for storing plotting data
		time	Used to keep track of time.
		PyQt4	Cross platform graphical toolkit originally designed for C++

3.2.1 PLC OVERVIEW

In order to regulate test temperatures via motor position, an Arduino compatible microcontroller was implemented as a PLC to manage the PID algorithm, motor control, and temperature readings. The selected SainSmart Uno is 100% compatible with the

Arduino platform, and is programmed through the Arduino IDE [32]. The Arduino program, or sketch, utilizes the availability of several open source libraries, as will be discussed. Code for the sketch is broken into three sections: global declarations, the setup function, and the loop function. The following is a discussion of this code, with excerpts from the sketch appearing in bold. A copy of the sketch is included in Appendix C.

For ease of understanding, Figure 3.12 provides an overview of the PLC code layout. The first section of the sketch is labeled global declarations. This section is technically not a function, but rather provides support for the operation of later functions. Figure 3.12 illustrates the importation of libraries, assignment of pins, reading of motor positions to the EEPROM, declaration of variables, assignment of constants, and creation of objects for the global declaration section occurring in a sequential manner. In reality the sequence of events does not affect the program; however, it is important that these actions occur before any other functions are implemented. The setup function follows, and Figure 3.12 depicts the necessary steps to prepare the PLC for application of the PID control loop. Unlike the global declaration section, the setup function does require sequential organization as the order of events is important. Lastly the loop function is shown. As its name implies, the contents of the loop are repeatedly cycled through. This section contains logic for merging motor control, temperature reading, and PID control. The following 4 sections will walk through the PLC Arduino code step by step, which can be viewed in its entirety in Appendix C.

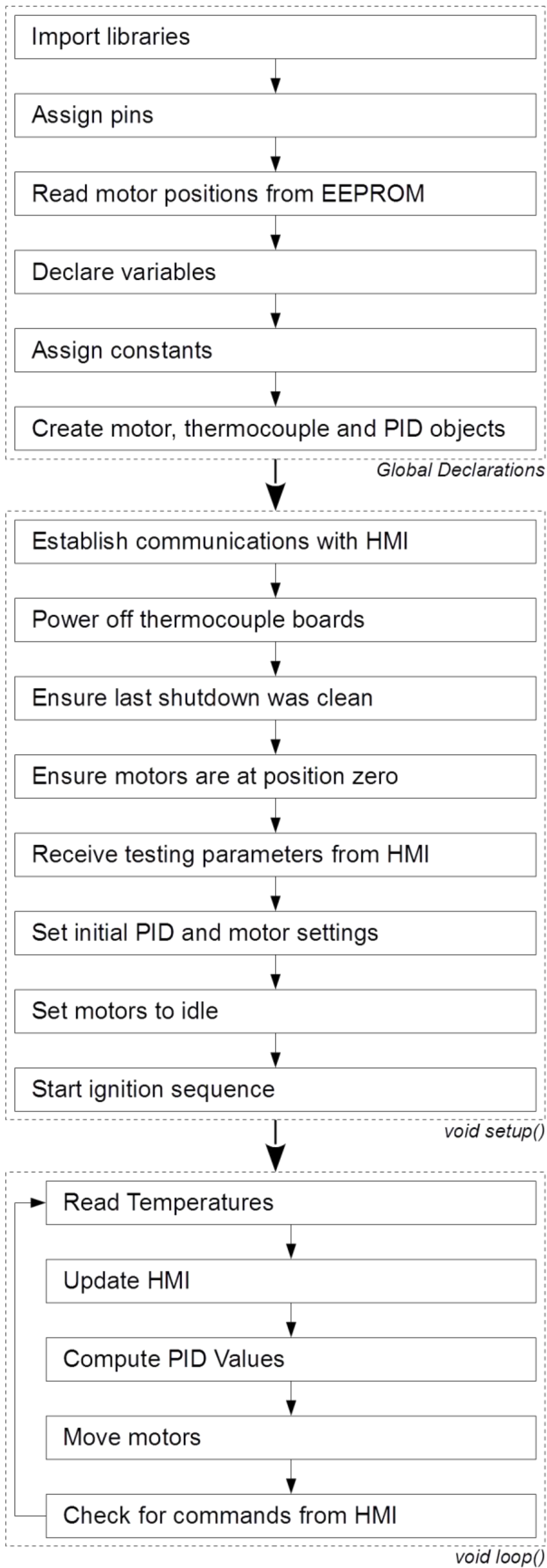


Figure 3.12: PLC Software Overview

3.2.2 GLOBAL DECLARATIONS

The first part of an Arduino sketch declares global items. Items which are global can be accessed by all other functions within the sketch, eliminating the need to pass variables between functions. This section includes the importation of libraries, and declaration of variables, constants, and objects.

Currently the sketch utilizes the AFMotor.h, EEPROM.h, Adafruit_MAX31855.h, and PID_v1.h libraries which are initialized by the keyword **`#include`**. The EEPROM.h library is a standard Arduino library whose purpose is to read and write data from the Electrically Erasable Programmable Read-Only Memory (EEPROM). AFMotor.h and Adafruit_MAX31855.h libraries were both obtained from the Adafruit website and interface with the motors and thermocouples respectively. The AFMotor.h library allows for simplified control of the L293D motor shield. Note that the library has been slightly changed to support simultaneous movement of two motors. The Adafruit_MAX31855.h library is used to read temperatures from the thermocouple breakout boards via the **`readCelcius()`** function, while the PID_v1.h library defines a robust PID algorithm that accepts a series of inputs and parameters, and calculates outputs for the control process.

Once the libraries have been initialized, pin assignments are created. The L293D motor shield does not require pin assignments as its pins are physically constrained to mate with those of the Uno board. However, the MAX31855 board does require pin assignments. Digital output and clock assignments are shared over the same pins for both boards, while chip select and V_{in} pins are assigned separately.

After pins are assigned, temporary motor position variables are initialized, and stored values from the EEPROM are read to them. The EEPROM maintains its memory even when the board is turned off, providing an ideal form of storage for motor position data. A flag which indicates if the motors were moving during shutdown is also stored to the EEPROM.

Next variables are initialized and defined. As previously mentioned, the variables in this section of code are global, and do not have to be passed into functions throughout the entirety of the program. The values of these variables can be changed as long as they maintain the type to which they are cast. Those variables cast as constants take up less program memory, but their values cannot be changed.

Objects are then created based on the imported libraries and include **`AF_Stepper`**, **`Adafruit_MAX31855`**, and **`PID`**. The **`AF_stepper(steps, num)`** motor object is initialized with the value of 200steps/rev and the specific motor's position on the motor driver shield. The **`Adafruit_MAX31855(SCLK, CS, MISO)`** object is created with the previously assigned clock, chip select, and digital output pins as attributes with respect to the corresponding thermocouple. The **`PID(*Input, *Output, *Setpoint, Kp, Ki, Kd, ControllerDirection)`** object is initialized with three pointers and four parameters. The **`*Input`** pointer, *temps*, stores the most current temperatures. The **`*Output`** pointer, *bottomOutput* and *topOutput* is where the PID object stores the next calculated motor positions. The **`*Setpoint`** pointer, *desiredTemp*, stores the target temperature that the PID is trying to maintain. **`Kp`**, **`Ki`**, and **`Kd`** are the initial tuning

parameters, and **ConrollerDirection** is set as *DIRECT*, indicating that an increased input will lead to an increased output.

3.2.3 VOID SETUP

The function **void setup()** is a function that is ran at the beginning of the sketch and will only run once per power cycle of the Uno. Because the function is only ran once, it is a convenient location to place commands which only need to be ran at startup, such as setting the values of variables, pin modes, and setting object parameters. Inside the function, communication is established with the HMI, and the data rate is set to 9600 bits per second. **pinMode** for the MAX31855 boards sets the power pins to ***OUTPUT*** so that they may be independently powered on or off. This becomes necessary later in the program when ground looping must be accounted for. The function also checks the EEPROM for indications of an unclean shutdown of the Uno.

Next, **moveMotors(0)** is issued and the motors are moved to a starting position of zero steps if they are not already there. Note that the input *0* to the function does not signal a move to zero steps, but is an operational mode. The **moveMotors()** function begins by checking to see if the motors are currently at the predetermined desired location. If they are not, it calculates the number of steps the motor needs to move, and constrains it within a starting position of 0 and an end position of 1200 steps. This prevents the motors from attempting to exceed the system's mechanical limits. The function then checks to see if motor speeds have been changed, and recalculates the speed at which the PLC needs to control the motors. The motor direction is determined, and finally the motors are signaled

to move. This part of the **moveMotor()** function code was implemented from the AFMotor.h library and revised to control the simultaneous movement of two motors. Before the function ends, the new motor positions are saved by **saveMotorPosition()**, a function that will be discussed later, and the motors are "released" or powered off so as to prevent overheating.

After the **moveMotors()** function is issued, **receiveParameters()** is called and waits to receive parameters specific to the temperature set-point, maximum motor positions, and P, I, and D parameters as entered by the user from the HMI. The receiveParameters function waits in a while loop until serial from the HMI is detected. The while loop calls the **printStatus()** function to continually update the HMI with current motor and temperature data. Upon exit from the while loop, values of the parameters are read.

After successful reading of the input parameters, both PID loops are configured using functions from the PID_v1.h library. **SetTunings(Kp, Ki, Kd)** updates the P,I, and D gain values, while **SetMode(Mode)** defines the PID algorithm mode. The mode can either be *AUTOMATIC* or *MANUAL*. Automatic mode describes a closed feedback loop where the feedback is used to adjust the outputs, while manual mode is open looped without feedback and the user manually adjusts the outputs. **SetOutputLimits(Min, Max)** defines the range for the PID output in terms of motor position limits, and **SetSampleTime(NewSampleTime)** sets the PID computation time in milliseconds.

The **`setSpeed(rpm)`** function from the AFmotor.h library allows us to set the motor speed in RPM. However, the function has been modified to return the required delay, in microseconds, between motor step movements. This value is saved as **`usperstep`** and is of significance because it is required by the custom function **`moveMotors()`**, as discussed above.

Lastly, **`lightFires()`** is called to start the heating process. This function begins by sending a message to the HMI indicating testing is ready to start. The function then remains in a while loop, continuing to update the HMI with **`printStatus()`**, until it receives the start trigger *"s"* from the HMI. At this point the **`moveMotors()`** function is called, and the motors are moved to their ignition locations at 600 steps. Starting temperatures are also noted, and after the bottom temperature rises 3 degrees the PLC sends the command *"CMD: Started"* to notify the HMI that ignition has occurred. This concludes the **`void setup()`** function and the sketch moves on to the **`void loop()`**.

3.2.4 VOID LOOP

After the **`void setup()`** function, the sketch continuously runs the **`void loop()`** function until the PLC is power cycled. The loop begins by calling the **`printStatus()`** function to report the motor positions and temperatures to the HMI. Next the **`bottomPID.Compute()`** and **`topPID.Compute()`**, as defined by the PID_v1.h library, are called. When these functions are called, they reference the recently read temperature values and evaluate the PID algorithm to compute new motor positions. Next

moveMotors(0) moves the motors from their current position to their newly computed positions. Lastly **parseSerial()** is called to check if more data is available to read.

3.2.5 SUPPORTING FUNCTIONS

saveMotorPositions(bottomDesired, topDesired) records the bottom and top motor positions to the EEPROM. Because the motors are allowed to move to a maximum position of 1200 steps, and the EEPROM is comprised of 8 bit bytes, two bytes of EEPROM must be used to save each motor position. The task of saving the integer value to two bytes is presented as follows. The first byte stores the number of times the motor position is completely divisible by 256, or the maximum value of one byte. The second byte stores the remainder. After each value is computed, they are saved to the EEPROM. The motor position may then be determined by multiplying the first byte by 256, and adding it to the second byte. In the **moveMotors()** function, a flag is set to indicate that the motors have started to move. This flag is then set to low in **saveMotorPositions()** after the values have been successfully saved. This provides the ability to determine if a power loss/cycle occurs while the motors are moving. If a power loss/cycle occurs during motor movement the position of the motors could be recorded incorrectly, whereas if it occurs while the motors are stationary, their positions are known, and the PLC may safely return the motors back to a position of zero.

printStatus() is called to communicate the motor positions and temperatures over serial. Motor positions should already be stored to a variable and are directly referenced by the **printStatus()** function. Temperatures, on the other hand, are determined by

calling **`readTemps(&temps[0])`**, which saves both temperatures to an array. The **`readTemps()`** function is responsible for reading both temperatures through the breakout board. Because of limitations of the breakout boards, ground looping issues occur. These issues are accounted for by quickly powering on and off the MAX31855 boards, resulting in a minimum cycle time of approximately 300 ms.

3.2.6 HMI OVERVIEW

The HMI is a GUI designed to display a central real-time plot displaying motor positions and temperatures with respect to time. Above the plot, information pertinent to the test is displayed and periodically updated throughout the test as needed. These fields include temperatures, motor positions, and heating rates for the top and bottom heaters; com port information; and the total test time with a 3 second countdown timer for when both temperatures are within set-point boundaries. Figure 3.13 provides an illustration of what the GUI looks like. The HMI is programmed in Python, and is comprised of a PyQt GUI file, a thread safe serial com monitor, and several other utilities. It is derived from a Python PyQt real-time graphing demonstration program named **`plotting_data_monitor`** [37] accessible at GitHub [38]. The following sections will follow the Python code as found in Appendix B and discuss the independent HMI components, and how they interact as a whole.

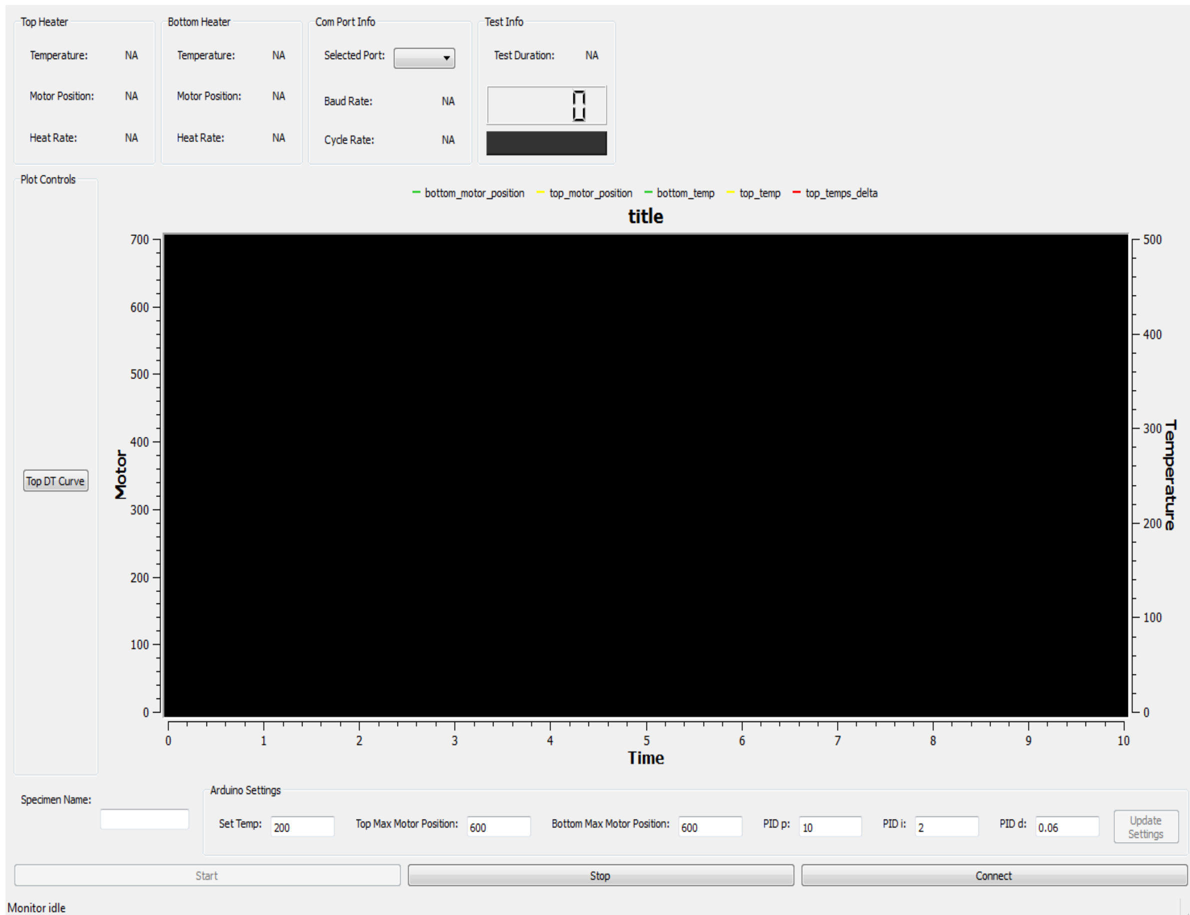


Figure 3.13: GUI_PID Display Screen

3.2.7 QT(PYQT4)

Qt is an open-source framework specific to developing graphical user interfaces (GUI), that can be implemented across many platforms [39]. This framework was selected for use because it provides high level graphing capabilities, a well-documented application programming interface (API), an extensive online development community, and is actively developed. Qt is currently on its fifth version; however the developed GUI was implemented using Qt4 due to package dependencies for Qwt, a compilation of classes and widgets for plotting. Qt is written in C++, but bindings have been developed for various programming languages, including Python.

Python's bindings for Qt4 are handled by a package known as PyQt4. PyQt4 is divided into modules, with the modules QtGui and QtCore being used. The QtGui module contains most GUI classes, while QtCore contains essential non-GUI related classes. Especially important is its implementation of signals and slots for dealing with events [40].

Signals and slots are used to communicate between objects when an event occurs. A signal is emitted when a certain event happens, while a slot is typically a Python callable which is called when the signal connected to it is emitted. The version of PyQt4.5 as programmed for use in this thesis, introduced a new style API which implements the key phrase **`QtCore.SIGNAL('clicked()')`** to emit a signal. The concept of signals and slots was instrumental in the creation of the HMI as it allowed for the user to interact and select buttons to ease the control of certain aspects of the temperature control program [41].

3.2.8 EBLIB

The eblib is a package of utilities [42] used by the **`plotting_data_monitor`** [37]. The serialutils and utils modules from eblib define four functions, **`full_port_name()`**, **`enumerate_serial_ports()`**, and two communication thread-safe functions **`get_all_from_queue()`**, **`get_item_from_queue()`**. The **`full_port_name()`** function gets the full serial port name based off the provided short name. **`enumerate_serial_ports()`** uses the windows registry to return a list of serial com ports (shorthand names). The **`get_all_from_queue()`** function gets all the items from any queue without hanging up the application when there are no items in the queue. It is used in conjunction with the **`data_from_arduino`** queue, which is shared with the

com_monitor object and runs in its own thread. The **get_item_from_queue()** function attempts to get single items from the queue, however, if 0.01 sec passes without the function finding anything, it returns *none* indicating an empty queue. It is also used to check for errors from the **error_queue** which is also shared with **com_monitor**. In summary the eblib utilities are important because they allow for the acquisition of the short and full serial port names, as well as provide functions that can access one or all items from a queue [43].

3.2.9 COM_MONITOR

The **com_monitor** module is included to provide communication over the serial port using PySerial, by implementing a run function to continuously read and write to the serial port in a thread safe manner [38]. PySerial is an imported module for the purpose of accessing the computer's serial port. It automatically selects the appropriate backend for Python, depending on which operating system is in use. PySerial is known for its cross-platform ease of use, as its syntax remains the same despite being used with different operating systems [44]. The **com_monitor** module [37] was originally designed to only receive information; however, two-way communication was desired between the HMI and PLC. A small modification was made to the code, resulting in the successful sending and receiving of messages.

3.2.10 LIVE_DATA_FEED

The **live_data_feed** module contains **LiveDataFeed()**, a class that houses new data and allows the user to post or read data from it. It also keeps track of whether the

new data has been previously read [38]. With respect to **GUI_PID**, **live_data_feed** provides a location for the GUI to store data and allows the function **update_info_and_plot()** to quickly decide if new data exists and whether the real-time plot needs to be updated.

3.2.11 GUI_PID

The **GUI_PID** is the script file that creates a GUI for the HMI. It consists of a central widget and status bar. Functionality is brought to the widget by adding labels, fields, and displays which are organized by boxes. Placing the boxes in the widget arranges the overall look of the GUI, and allows for customization. The GUI also communicates with the PLC to populate specific fields or displays. To achieve these results, **GUI_PID** defines five classes as discussed below.

The **CoupledBox(QtGui.QWidget)** class combines a label and a type of field to create an information display. The field may be defined as a **QLabel**, **QLineEdit**, or **QComboBox**. **QLabel** displays text; **QLineEdit** displays a single line text editor, which is commonly used as an input box; and **QComboBox** displays a dropdown list. These fields are used to accommodate user inputs and show relevant test information. This class was created to simplify the main GUI display and standardize formatting.

The **RealTimePlot(QWidget)** class produces a real-time plot of selected data vs. time. Its main purpose is to set the graph backgrounds, axis properties scales, and legends. Motor positions and bottom and top temperatures vs. time are plotted by the **RealTimePlot(QWidget)**. The **RealTimeCurve(Qwt.QwtPlotCurve)** class

defines curves based on the available data as well as determines such parameters as solid lines for the curve's display, colors of the curves, and setting y-axes on both the left and right sides of the graph.

The **MyLCDCounter(QLCDNumber)** class creates a countdown timer with LCD-like digits. The timer counts down from three seconds when the top and bottom temperatures are within bounds of the temperature set-point. If the temperatures slip below or rise above the bounding temperatures the countdown timer will wait until the limits are again met to start over. The timer is based off the **QtCore.QTimer** class and automatically updates the LCD numbers.

The **PlottingDataMonitor(QtGui.QMainWindow)** class is the main class of **GUI_PID** and is responsible for creating and managing graphical entities of the GUI. This class uses the data types dequeues and lists to store data, initializes **com_monitor** and **live_data_feed**, and finally creates and manages GUI specific components. The **PlottingDataMonitor(QtGui.QMainWindow)** class is comprised of four sub functions: **create_status_bar()**, **create_top_info()**, **create_mid_plot()**, and **create_bottom_inputs()**. **create_status_bar()** creates the status bar for displaying text messages from the PLC. **create_top_info()** creates and populates a box at the top of the GUI widget to display information about the top motors and temperatures, bottom motors and temperatures, communication of the program, and test information such as total time and status. The **create_mid_plot()** function creates a real-time plot displaying motor positions and temperatures vs. time. It also includes a

button for turning a curve displaying the delta between the set-point and top temperature on and off. The **create_bottom_inputs()** function creates and populates two boxes at the bottom of the GUI for user interactions. The topmost box has entry fields for specimen name, set-point temperature, top and bottom maximum motor positions, PID parameters, and a button for updating the settings. The bottom box contains three buttons: start, stop, and connect. Various buttons are toggled on or off depending on the sequence of the test process.

3.2.12 IMPLEMENTATION

The modules described above all contribute to the functionality of the **GUI_PID** script. They can be found throughout the various classes defined in **GUI_PID**, and provide critical functionality. As mentioned, **PlottingDataMonitor(QtGui.QMainWindow)** is the main class for **GUI_PID**, and it utilizes all the other classes in the script, except for **RealTimeCurve(Qwt.QwtPlotCurve)** class which is called by the class **RealTimePlot(QWidget)**. **PlottingDataMonitor(QtGui.QMainWindow)** is the name of the script which calls the **main()** function. This is the main function of the script and is triggers the creation of the GUI.

3.3 DEVELOPMENT COSTS

A driving force behind the design of the testing apparatus was overall cost. As described in Chapter two, systems capable of maintaining ASTM standards and achieving acceptable heating rates already exist. However, these systems are extremely expensive. Gleeble

Systems can cost up to $900,000.00 [20] and used induction heaters can cost around $6,000.00 [23].

In comparison the developed testing apparatus' temperature control hardware totaled around $200, with raw material costs estimated to be approximately $100.00. It is important to note that the developed system is intended as an add-on to existing test frames such as the MTS servo-hydraulic frame used for the research displayed in this thesis. Also, the prices displayed do not account for the cost of manual labor associated with manufacturing, such as for the machining of grip parts, or time associated with tuning the PID control settings. ***Table*** 3.4 provides a detailed list of system expenditures which total approximately $300.

Table 3.4: List of Expenditures

Component	Vendor	Cost ($)
MAX31855 Thermocouple Breakout Board (x2)	Adafruit Industries	29.90
Breadboard + Jumper Cables	Amazon	9.69
SainSmart Uno	Amazon	17.69
SainSmart L293D Motor Shield	Amazon	11.18
Canakit Stepper Motor (x2)	Amazon	37.90
Type K Thermocouples (5 pack)	Omega	35.00
Bernzomatic TS3000KC (x2)	Home Depot	49.94
Raw Materials	U of I Machine Shop	100.00
	Total:	**291.30**

CHAPTER 4: TESTING PROCEDURE INSTRUCTIONS

The following sections detail and lay out the steps necessary to perform testing using the elevated temperature testing apparatus. It is important to note that the following series of instructions assumes that the testing platform has been set up according to the methods laid out in this thesis. This includes specimen and grip geometry, thermocouple locations, nozzle locations, and upper and lower motor limits for the temperature control system. It also assumes that the torch supports, and automated motor control assemblies have been attached to the test frame.

4.1 TEMPERATURE CONTROL SETTINGS

Before an actual test can be run, the PID settings and maximum motor positions must be defined and entered into the GUI. One of the downfalls to using a PID control algorithm can be the tuning of input parameters. For the set-up described in this thesis it is necessary to tune the PID settings before testing begins. This means one has to set up a secondary, expendable test specimen with thermocouples attached and connected. Validation of the PID settings should be performed on a load train set-up as similar to the real testing environment as possible; this includes starting each tuning test with a room temperature specimen and grips.

When first tuning a PID control loop, it is advised to approach the situation in a logical manner. Most literature recommends beginning by tuning the proportional value (setting I=0, and D=0) to the best of ones abilities. Start by setting P to a low value such as 1, and take note of the system's response to each tuning test performed. At a low P value the

system should struggle to reach the temperature set point. Next increase the P setting to a value that causes the system to oscillate around the set point. This will be an indicator that the P value is too great, creating an unstable system. Now decrease the P value until the system starts to steady out. At this point an offset like that described in CHAPTER 2: may occur. In order to compensate for this offset an integral value may be introduced. Repeat the same series of steps for the integral value as were followed for the P value, taking note of how the system responds to each set of inputs. CHAPTER 2: also mentions that the PI controller is one of the most common types of controllers used in industry. Depending on the system, this may be all the control necessary to meet one’s requirements. If not, the derivative value can be introduced. The D value will help to accelerate the rate at which the system approaches the set point. However, the D value should only be altered by small amounts, such as on the order of tenths of a point. Just as the D value can cause the system to quickly reach equilibrium, if entered incorrectly, it can also cause the system to become very unstable. Tuning the PID values may require keeping a log of system responses to various P, I, and D values. By acquiring this knowledge, an adequate set of values can be found that satisfy the system temperature testing requirements. It is also important to keep in mind that PID settings and the stability of the system may be affected by empty propane cylinders, incorrectly assembled load trains, unsymmetrical thermocouple placement, and altered maximum motor positions.

4.2 SPECIMEN PREPARATION

Before testing can begin the specimen geometry must be measured to record the minimum gauge section diameter and length of the gauge section. The minimum diameter

is necessary for calculating the cross-sectional area of the specimen for generating stress strain plots, while gauge length is used in percent elongation calculations. Percent Elongation can be measured off of the shoulders of the specimen, but more accurate measurements are preferred. This is accomplished by scribing two small lines at a set distance apart in the reduced section of the specimen. The original distance between marks is the initial gauge section length and should be recorded for later use in percent elongation calculations. These marks should be deep enough that they can still be seen after testing, but small enough to not create inclusions or imperfections where cracks may start. The validation testing performed in CHAPTER 5: used original gauge section lengths of approximately 25.00 mm and were created using calipers and a razor blade. Measurement of the elongation gauge section needs to be accurate, and should be measured using an appropriate device.

4.3 THERMOCOUPLE ATTACHMENT

It is important to determine thermocouple position and attachment technique before testing begins because both attributes can greatly affect PID values, as well as nozzle position and flame intensity. As stated in Section 3.1.3 , thermocouples are attached 27 mm from each end of the specimen, using clips made from stainless steel strip. The clip and thermocouple attachment location can be seen in Figure 3.5.

When attaching the thermocouples it is important to not place the thermocouple weld bead near the gauge marks. This is because the thermocouples could scratch or disfigure the gauge marks, causing an inaccurate or unreadable final gauge length measurement. It

is also important to locate the torch nozzles and displacement measurement device on separate sides of the specimen as well. Figure 4.1 depicts a recommended orientation for these components around the specimen. In it the round cross section of the specimen is reduced to quadrants, each of which is reserved for one of the previously mentioned functions. In Figure 4.1 it can be seen that the torch nozzles and thermocouples occupy opposite quadrants as a supplementary means of preventing the thermocouples from picking up inaccurate temperature readings. The elongation gauge marks are also located perpendicular from the torch nozzles and thermocouples, and opposite the displacement measurement device. This is particularly important if an extensometer is used because the knife edge attachment points could leave marks on the specimen which would disfigure the gauge section marks.

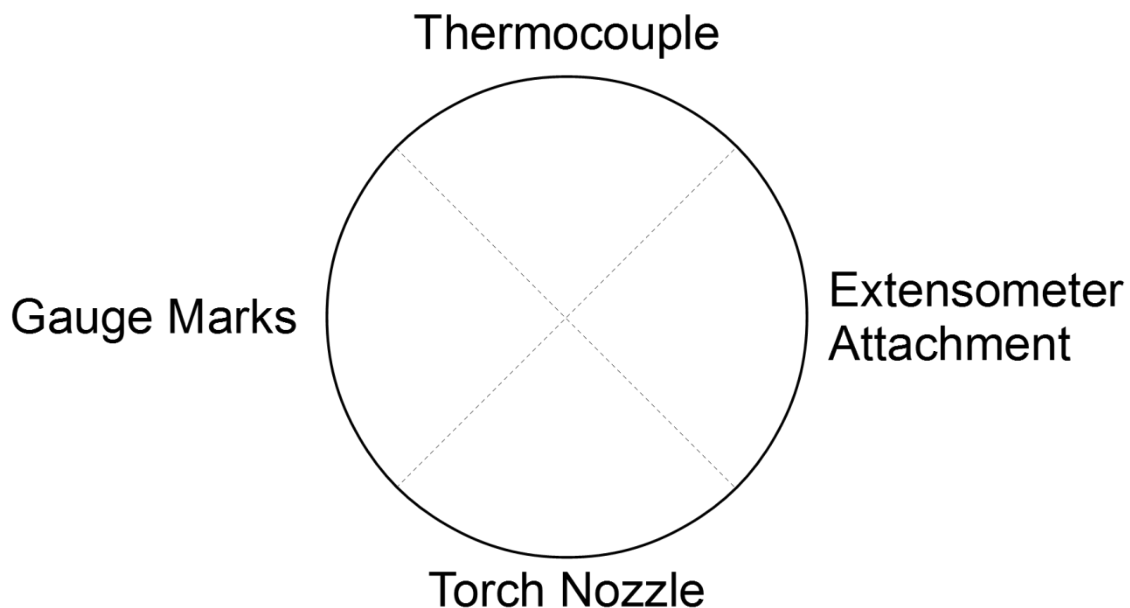


Figure 4.1: Specimen Quadrant Diagram

4.4 LOAD TRAIN SET-UP

Once the specimen measurements have been taken and thermocouples attached, it is time to place the specimen in the load train. As mentioned in Chapter 3, the research in this thesis developed two different types of grip connections, however if the seated connection grips are employed the process becomes much simpler and can be accomplished more quickly. When using the seated connection grips, lower the test frame actuator so that there is enough room to hang the full length of the load train without any interference. Next assemble both the top and bottom halves of the load train. Attach the top half of the load train first, using a 0.375” x 2” dowel pin to connect the load train to the test frame. Ensure that the seated connection grip is facing the torch nozzle and insert the specimen’s top button head into the grip’s opening. Pull down so that the specimen seats securely in the grip. This may be difficult to maintain until the bottom grip is attached due to the thermocouple wires pulling on the specimen. Next attach the bottom half of the load train in the same manner that the top was attached, confirming that the bottom grip opening also faces the torch nozzle as seen in Figure 4.2. The load train should be fairly balanced now, but if it is not check to make sure that the specimen is seated securely in both grips. Bring the actuator up so that the bottommost connection of the load train can slide into the slot on the actuator grip, but do not insert the pin. It was found that the extra step of raising the actuator helps provide more stability when connecting thermocouple leads.

Alternatively, the pin connection grips may be used. In contrast with the seated connection grips, it was found that assembling the entire load train before attaching it to the test frame was easiest, as seen in Figure 4.3. Assemble both the top and bottom halves of the

load train separately from the test specimen. Then check that the 0.1875" x 2" dowel pins slide easily through the holes at each end of the specimen. If they do not, run a 0.1885" reamer through the hole by hand until the pin slides nicely through. Next pin the specimen to the top and bottom grips, making sure to match the top to the top and bottom to the bottom. For repeatability mark the sides of the grips that are to face the torch nozzles, and ensure that the opposite side of the specimen is where the thermocouples are connected. At this point a slight gap between specimen edges and grip sides may be noticeable. If this is the case, it may be necessary to add shims. The shims seen in Figure 4.3 were made from a variety of shim stock thicknesses, cut in the shape of a tab, and punched with either a 0.375" or 0.25" hole. Add the shims to either side of the specimen, as symmetrically as possible to take up slack between the specimen and grip sides at both the top and bottom connection points. The load train is now ready to be attached to the test frame. Again ensure that the appropriate grip and specimen side are facing the torch nozzles, and lower then raise the actuator to add stability when attaching the thermocouples.

Once the load train is in place, verify that the specimen is oriented correctly and matches the diagram shown in Figure 4.1. The thermocouples should be located opposite the propane torch nozzles while the elongation gauge marks should be located midway between the two. When this has been accomplished connect the thermocouples to the MAX31855 break out boards. When the electronic components of the control system were set up, it should have been noted which MAX31855 board corresponded to which temperature read out. Thus, depending on how the Arduino code is written, one

Figure 4.2: Seated Grip Specimen Alignment

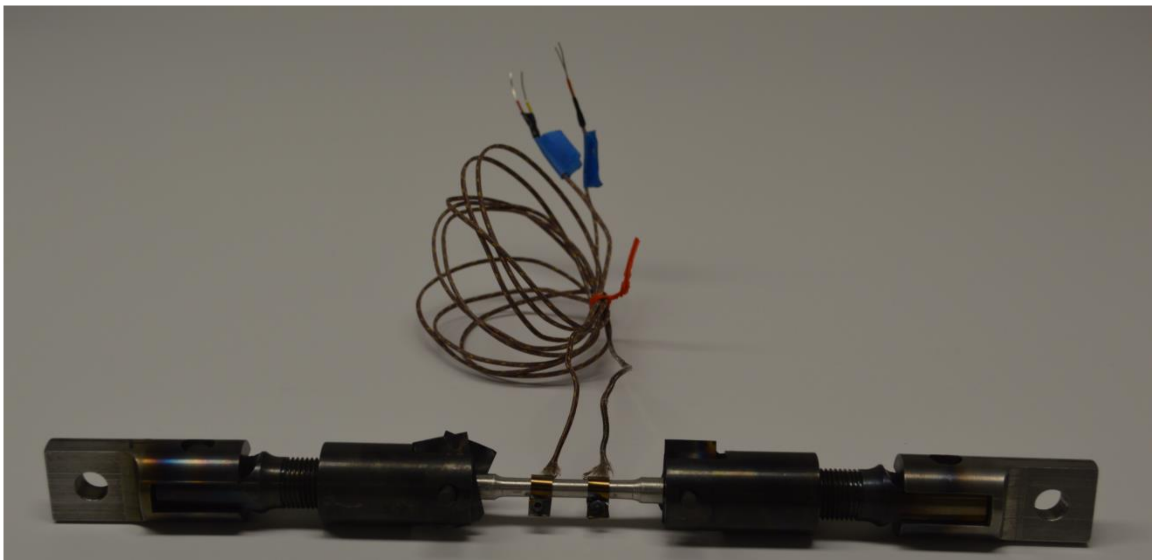


Figure 4.3: Pinned Grip Specimen Alignment

board should be designated for connection to the thermocouple located at the top of the specimen, and one should be designated for connection to the thermocouple at the bottom of the specimen. The boards used for testing in this thesis were labeled “T” or “B” for top and bottom so as to reduce the possibility of error when connecting thermocouples. Tags with similar labels were also taped to the thermocouple wires near their leads to further simplify the process. A proper connection between the board and thermocouple can be seen in Figure 4.4. If it should occur that thermocouple positions get swapped and that the top is connected to the bottom readout while the bottom is connected to the top readout, the GUI will most likely display feedback where the temperature for one thermocouple remains constant while its corresponding motor position increases towards its maximum value. Alternatively, the other thermocouple readout will display a temperature that is constantly increasing while its corresponding motor position remains at the minimum value. Thus it is imperative to correctly connect the thermocouples to the corresponding boards. When connecting the thermocouples it is also critical to correctly match leads to the appropriate connection. As seen in Figure 4.4 the board is already labeled with a + or - and the corresponding thermocouple lead is colored, yellow (+) and red (-). When the thermocouple leads are reversed the GUI will respond by displaying the temperature of the corresponding thermocouple decreasing into negative values, rather than increasing.

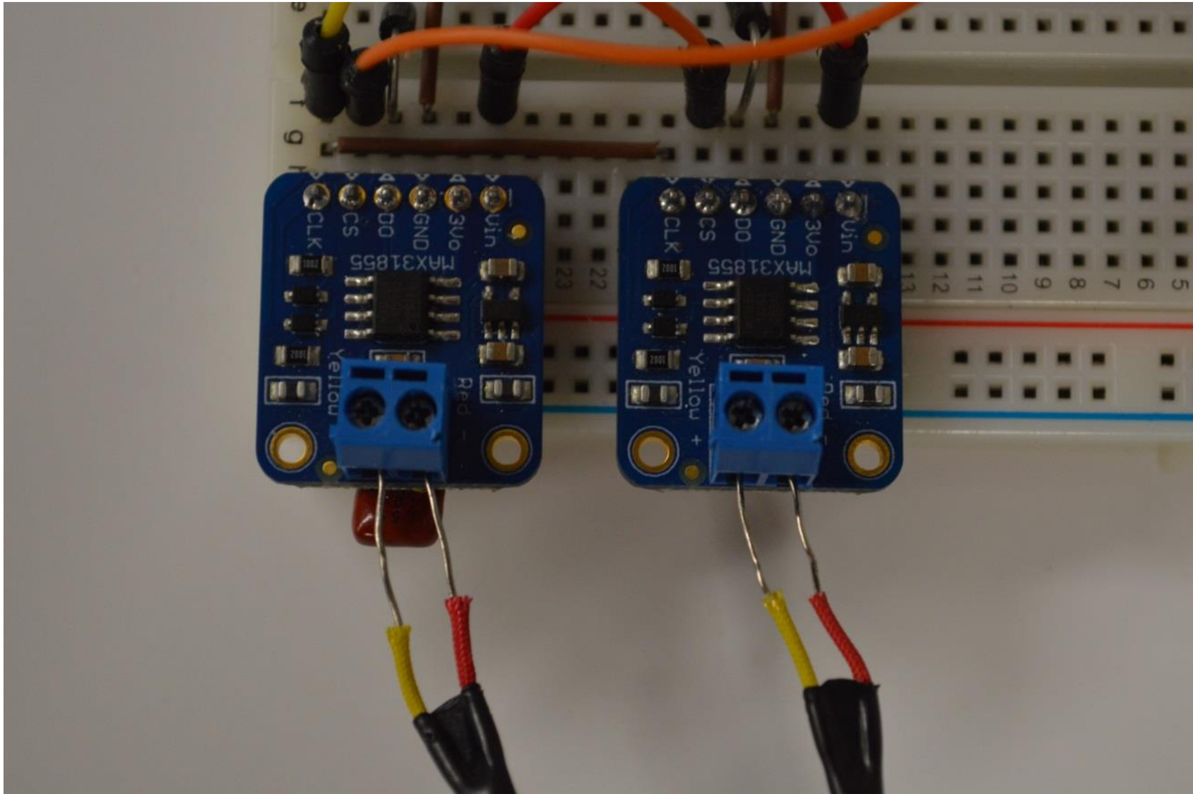


Figure 4.4: Thermocouple Reader Attachment

It is the author's opinion that attaching the shields after putting the load train in place and connecting the thermocouple leads is easiest, however, the order of operations is insignificant to testing, as long as care is taken and no parts are damaged. That being said, as previously mentioned, the seated connection grips are easier to work with and this continues to be true when attaching the shields. The shields used in conjunction with the seated connection grips are described in Section 3.1.3 and can be seen in Figure 3.6(a) and (b). Note that a "tongue" formed out of 0.006" shim stock was used to deflect the torch flame from shooting down the top grip opening, directly onto the specimen gauge section. For the pinned connection grips a different type of attachment provides for the same level of protection, and allows for symmetry between the top and bottom shield geometry.

These shields are designed to be supported by the pin connecting the specimen to the load train, and they come in two parts which effectively surround the top quarter of the grip. A flat bottom plate on the shield has a central hole, big enough to allow the specimen to fit through with room to self-align, but small enough to block any heat from the torch flame that may impact thermocouple readings. The two halves are held together in the front and back by straps fashioned out of shim stock or by a screw and nut. Curved sections perpendicular to the flat plate provide some alignment, and match the grip geometry, as shown in Figure 3.6(c).

It was also found that heating rates and profiles are not affected despite the difference in shield geometry and attachment techniques. This is attributed to changes in heat sinks due to the varied grip geometries. If an extensometer or other displacement measuring device is used it is advisable to attach it to the specimen after attaching the grips. As previously mentioned, the extensometer should be positioned on the side opposite the specimen gauge marks, and can be attached with stainless steel springs. The springs shown in Figure 4.5 are 0.125” in OD, 0.45” long, and formed of 0.017” 302SS wire. When the knife edges have been securely fastened, zero the COD reading and remove the extensometer pin. The load train should now be ready for the testing process to begin.

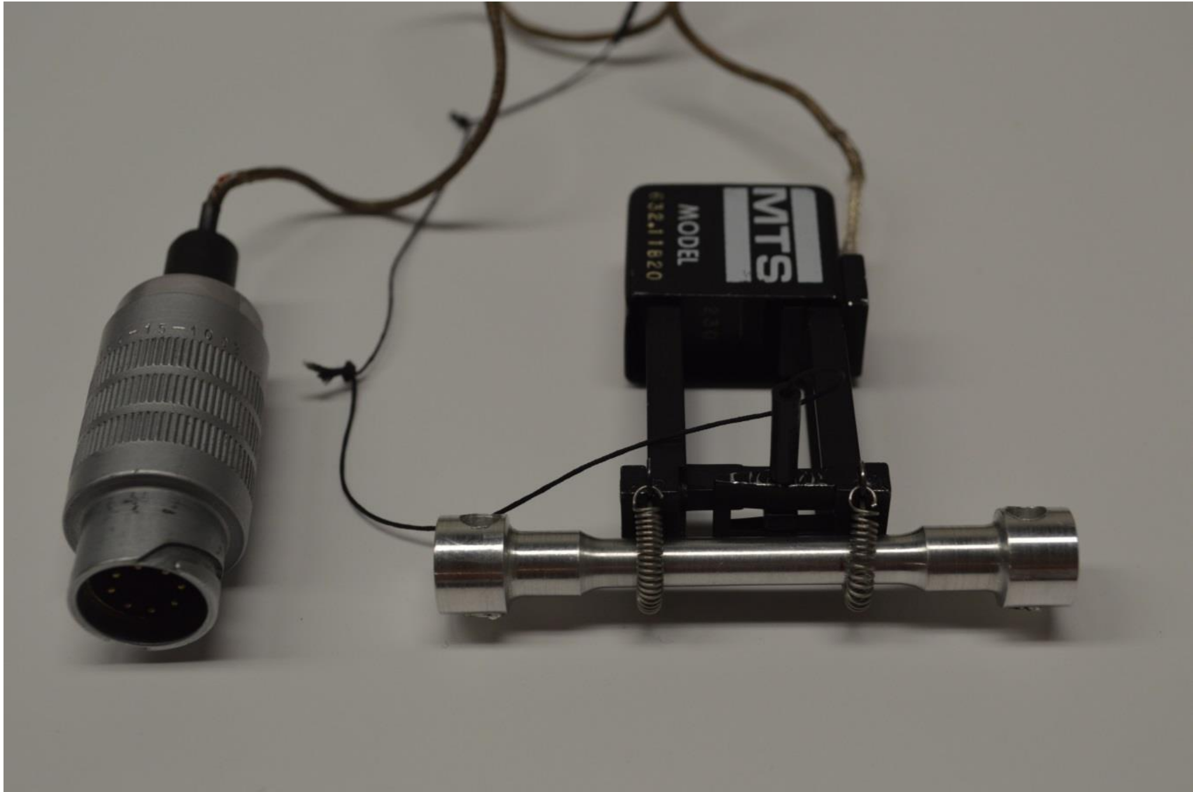


Figure 4.5: Extensometer Attachment

4.5 TESTSTAR TESTING PROGRAM

After the load train and thermocouples have been set up, testing parameters should be entered into the temperature and testing control programs. Depending on the type of testing platform in use, the control software will vary; however, the system available for use during the development of this thesis will be used as an example. Section 3.1.2 provides an overview of the MTS test frame, control system, and sensors. For the TWSX software in use, procedures developed for tensile testing typically operate at a constant strain rate, which was determined based on the following equations:

$$\dot{\epsilon} = \frac{\Delta L/_{sec}}{L_0} = \frac{1.75\ mm/_{sec}}{35\ mm} = 0.05/sec \tag{4.1}$$

$$\dot{\epsilon} = \frac{\Delta L/_{sec}}{L_0} = \frac{17.5\ mm/_{sec}}{35\ mm} = 0.5/sec \tag{4.2}$$

Where based off a 35 mm gauge section, 1.75 mm/sec and 17.5 mm/sec are the required LVDT rates to achieve the desired strain rates of 0.05/sec and 0.5/sec respectively. In the Testware SX procedure it is important to make sure that the correct LVDT rate value and units are entered before beginning a test, as well as the correct data acquisition rates. Again, depending on the limits of the testing device, this value may need to be altered. It was determined that collecting data every 0.01 mm of actuator movement was sufficient to provide accurate data, and was within the limits of the TestStar controls. The MTS testing frame has data collection limits set at a maximum of 5 KHz, as found in the user's manual. Data collection may also be defined in terms of extensometer movement, for which the limits vary depending on the strain rate at which tests are performed. When considering the mechanical limits of the system, it may also be useful to initialize bounds based on LVDT or COD readings. These limits may be implemented from inside the Testware SX program, or externally by placing interlocks on the TestStar platform.

4.6 TEST START

Once the load train has been set-up and correct settings entered into the control programs, testing can begin. Briefly double check the entire system to make sure everything is correct before raising the actuator to a level where a pin can easily slide through the actuator grip and bottom load connection point. Do not actually pin the connection, but leave the test frame hydraulics running. The load train should not be connected at the bottom because the specimen could elongate during heating. If elongation were to occur while the load train was pinned at both ends then the specimen could become unseated or placed under a compressive load, neither of which creates a desirable test starting condition. If an

extensometer is in use, zero the COD and remove the extensometer's pin. Next verify that the TWSX program is in "Execute" mode and click the C*onnect* button on the GUI. Wait for the temperature and motor positions to start plotting on the GUI graph, and the message "Serial Connected, Enter Settings" to be displayed in the message bar in the bottom left corner of the screen. Enter the temperature settings, completing the *Specimen Name*, *Set Temp*, *Top Maximum Motor Position*, *Bottom Maximum Motor Position*, *PID p*, *PID i*, and *PID d* fields. When these have been filled out select the *Update Settings* button, at which point the message, "Press Start to Begin Test" should appear and the *Start* button should become activated. When it is time to begin the heating portion of the test, click the *Start* button to move the motors and propane torch valves to the ignition position. When this occurs, the hiss of propane being released from the cylinders should be audible and "Ignite Propane Torches" will be displayed in the message bar. Simultaneously pull the ignition triggers on the propane torch handles to begin heating of the specimen. Once the torches are lit and the bottom thermocouple has sensed a change in temperature greater than 3°C the message bar will change to read "Fires Lit, Waiting for SS Target," or in other words, the PID control is waiting until temperatures are close enough to the set point, or steady state, to make adjustments.

When the temperature reads approximately 50°C below the set point begin connecting the load train and zeroing controls. This reference point was found through trial and error, and may need to be adjusted depending on the user's abilities. The first step in readying to start the tensile test is to zero the load cell. This may be accomplished using the Actuator Positioning Control (APC). Next adjust the actuator and slide in the pin so that the load

train is connected at both ends. Gloves may be necessary at this point to prevent burns. Finally turn off the APC and wait for the temperature to reach the set point. Depending on the desired heating rate and selected conditions, the GUI will notify the user when both the top and bottom temperatures are within +/-3°C of the set-point temperature by displaying "GO!" in a yellow box near the top of the GUI. If a hold period is desired, a timer counting down from 3 seconds is displayed above the yellow box. This 3 second hold may be adjusted by changing values in the GUI code. The 3 seconds is reset every time the temperature falls or rises outside the acceptable +/-3°C limits. When the clock reaches 0.0 seconds, the yellow box changes to green and displays the message "SUCCESS." At the users discretion the tensile test may be started as soon as the yellow message appears, or they may wait for the green message to be displayed. Tensile testing is started by pressing the "Run" button on the APC or on the computer display screen. The actuator should immediately move downwards at the specified rate, and will continue to do so until the program limits or mechanical limits are reached. Alternatively, the test may be ended by hitting the "Stop" button on the APC or computer screen. The propane torches may also be turned off by hitting the "Stop" button on the GUI.

4.7 POST TEST DATA ANALYSIS

Once the specimen has been tested to failure stop the test, but DO NOT touch any part of the load train. Depending on the set point temperature of the test this could result in severe burns. Let the load train cool down before collecting the specimen pieces for measurement—this should take anywhere from 5 to 15 minutes, depending on how cool the specimen is desired to be. During this time it is advisable to review the data collected

from the test and verify that it follows acceptable trends. As described in Section 2.2, this requires creating a stress strain plot from the load-displacement data generated during the test. Depending on if displacement was measured using the LVDT or extensometer, the corresponding plot will either be labeled “pseudo” stress strain, or “engineering” stress strain respectively. Typically a decrease in material properties can be expected for a corresponding increase in specimen temperature. This trend may also be expected for a decrease in strain rate as well. When the specimen has cooled enough, detach the thermocouple leads from the MAX31855 boards and remove the specimen pieces from the load train. It is advised at this point to mark the ends of the two halves so that they match one another and will not be lost or mixed up with other specimen. Re-assemble the specimen by matching up the fracture surface so that the gauge marks are in line with one another. Fastening the two halves together in an acceptable fashion, re-measure the distance between gauge marks using the same approach as when the marks were created. The percent elongation can now be found using (2.1). Typically percent elongation increases with temperature and decreases relative to strain rate. This is opposite of the trends seen for YS and UTS.

4.8 TEST PROCEDURE CHECKLIST

1. Attach specimen to corresponding grips
    a. For a pinned connection, add shims around the specimen if necessary
    b. Check thermocouple and gauge mark orientation relative to the grips
2. Pin the grips and specimen to the top of the test frame
    a. Lower actuator out of the way, and check that the LVDT settings will allow for enough travel during the current test

   b. Check that the grips are oriented correctly with respect to the torch nozzles
3. Attach shields, and extensometer if necessary
   a. Minimize any gaps in the shields
   b. Check that the extensometer springs are tight enough to hold the knife edges in place
4. Attach the thermocouple leads to the break out boards
   a. Check that the leads are attached correctly (colors and +/- symbols match)
   b. Check that the top and bottom thermocouples are attached to the correct break out board
5. Review the TWSX program
   a. Check LVDT rates and displacement limits
   b. Verify that the COD or LVDT data collection rates are within limits
   c. Check that the correct data file name has been entered
   d. Zero the TWSX display meters
   e. Check that the program is in "Execute" mode
   f. Enable interlocks if desired
6. Set up the temperature program
   a. Check the physical motor start position
   b. Check that the GUI is connected to the desired COM port
   c. Click on the "Connect" button to ensure that the GUI can connect to the microcontroller
   d. Enter the specimen name, temperature set-point, maximum motor positions, and PID settings
   e. Click on the "Update Settings" button and verify that the plot is updating
7. Raise the actuator up to level with the bottom pin connection
   a. Check that the LVDT is in the appropriate mode
   b. Zero the LVDT position if desired
   c. If an extensometer is being used, zero the COD reading and remove the extensometer pin

8. Double check that all aspects of the system are ready for test start
9. Click the “Start” button on the GUI
    a. Pull the torch triggers to ignite flames
    b. Monitor the testing temperatures via the GUI display
10. Approximately 50°C before the set-point temperature, ready the system for tensile test start
    a. Zero the load cell
    b. Adjust the actuator and pin it to the bottom of the load train
    c. Turn off the APC
11. Hit the “Run” button on either the APC or TestStar program to begin the tensile test

CHAPTER 5: TESTING RESULTS

Elevated temperature tensile testing was performed to evaluate stress strain curve reproducibility, material property trends, heating conditions, and to verify effectiveness of the developed propane torch and temperature control system. A series of six different tensile tests were performed at three different temperatures and two different strain rates, with duplicate tests performed for each set of criteria totaling twelve tests. Specimens made of a 70XX series aluminum alloy were tested at 25°C, 225°C, and 425°C at strain rates of 0.05/sec and 0.5/sec. Evaluated material properties include YS, UTS, and percent elongation. Heat up rates, and temperatures at tensile test start were also recorded. Modulus of elasticity values were not evaluated as testing was performed using the LVDT. Further discussion of these properties and trends is included in the following sections.

5.1 TESTING HEAT RATE PROFILES

Temperature heating profiles in relation to motor positions for both the top and bottom positons can be seen in Figure 5.1 through Figure 5.8. These plots are generated when temperature data is sent from the Arduino to the GUI and are displayed for the user; however, the Arduino is only capable of doing so when the motors are stationary. Thus the plots are not 100% real time, but provide a very good representation of what is happening with the system. To remedy this, a secondary Arduino would need to be implemented. Improvements to this aspect of the system are further discussed in CHAPTER 6:. In Figure 5.1 through Figure 5.8 the horizontal light blue dashed lines indicate the set point temperature, while the vertical green dashed lines indicate the tensile test start time. From

the plots it can be seen that all tests, except A5, were conducted when the temperature had reached a steady state at the target temperature. Figure 5.5 provides data for the testing of specimen A5. This test was conducted at temperatures above the set point due to an incorrectly entered Integral value in the PID settings. The effects of this error will be discussed later in the chapter. From the lower motor position plots it can be seen how the PID algorithm reacted to temperature changes via motor position. When the temperature is significantly far from its target temperature the motors are at their maximum positions. However, the closer to the set point that the temperature gets, the more the PID becomes active. This is evident in the lower positioned peaks on the motor position plots, as the PID tries to prevent and counteract any significant changes in temperature.

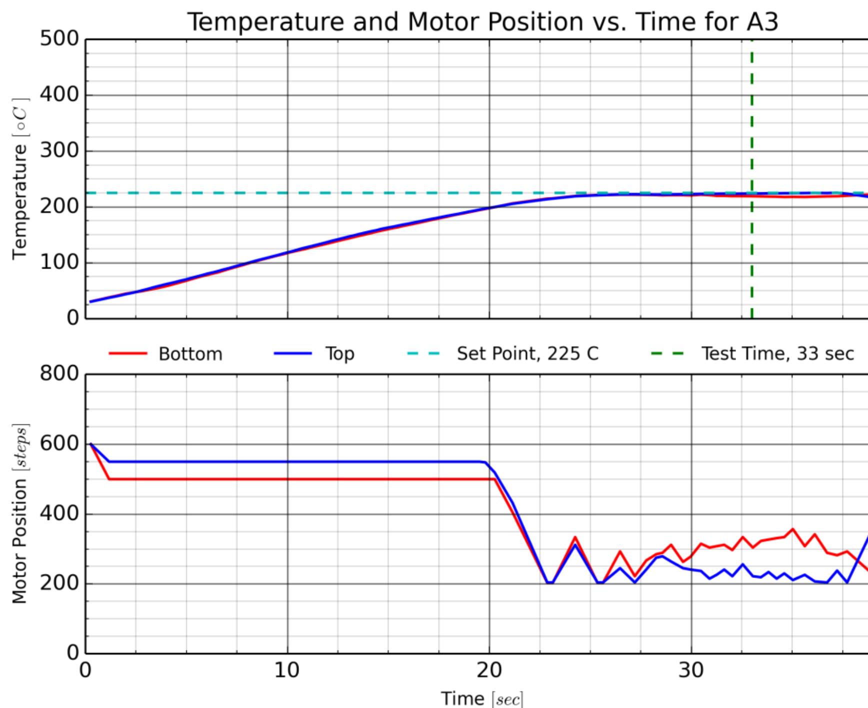


Figure 5.1: A3 TCS Data

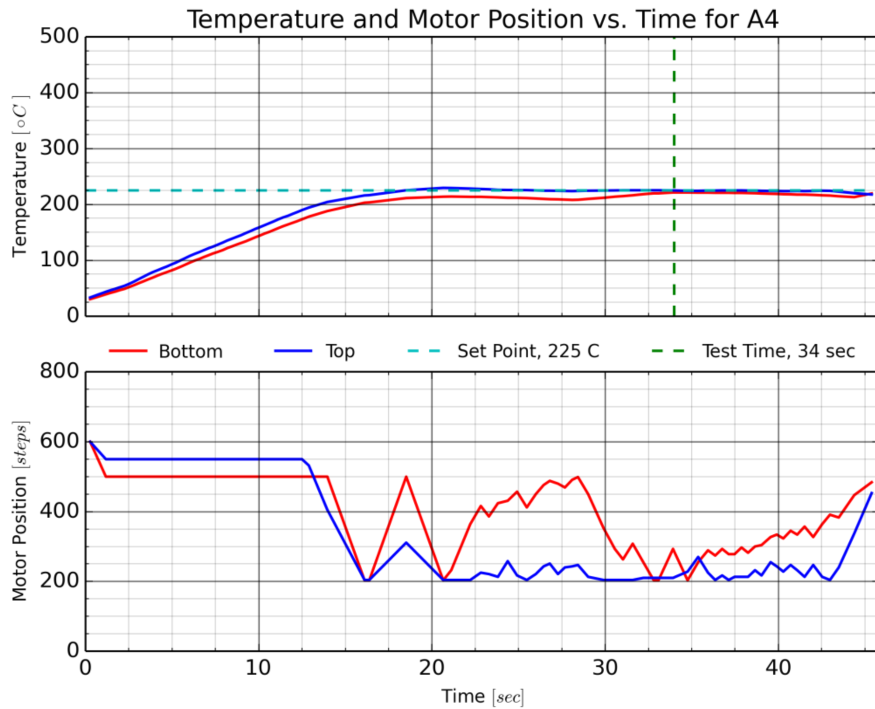


Figure 5.2: A4 TCS Data

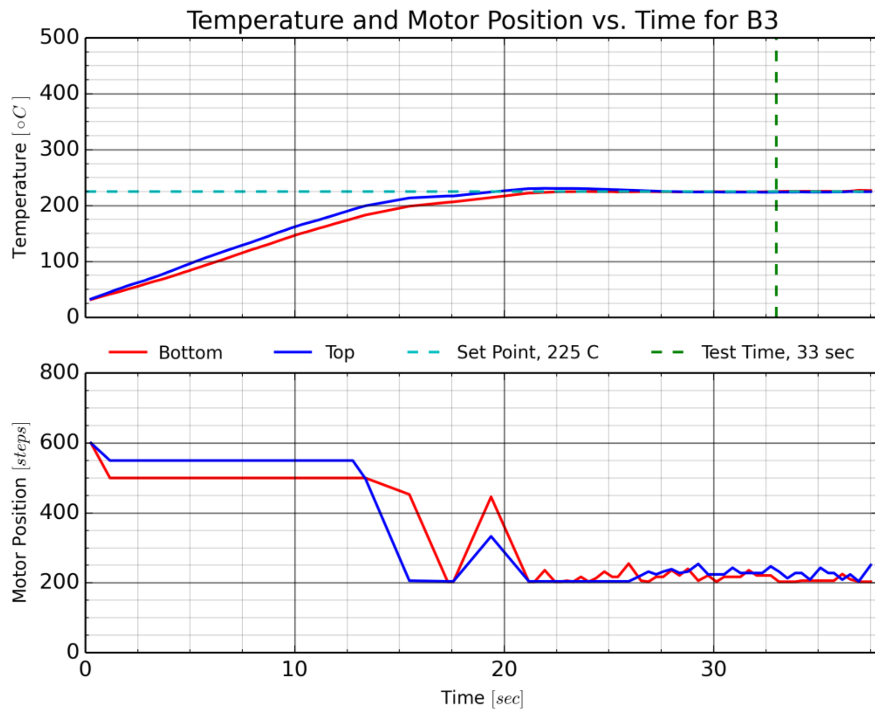


Figure 5.3: B3 TCS Data

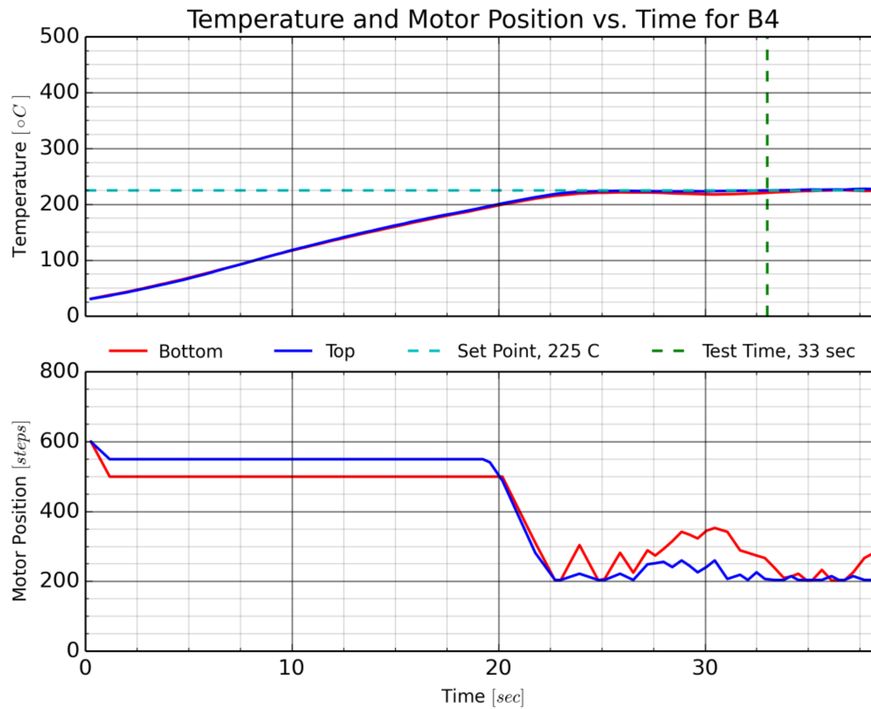


Figure 5.4: B4 TCS Data

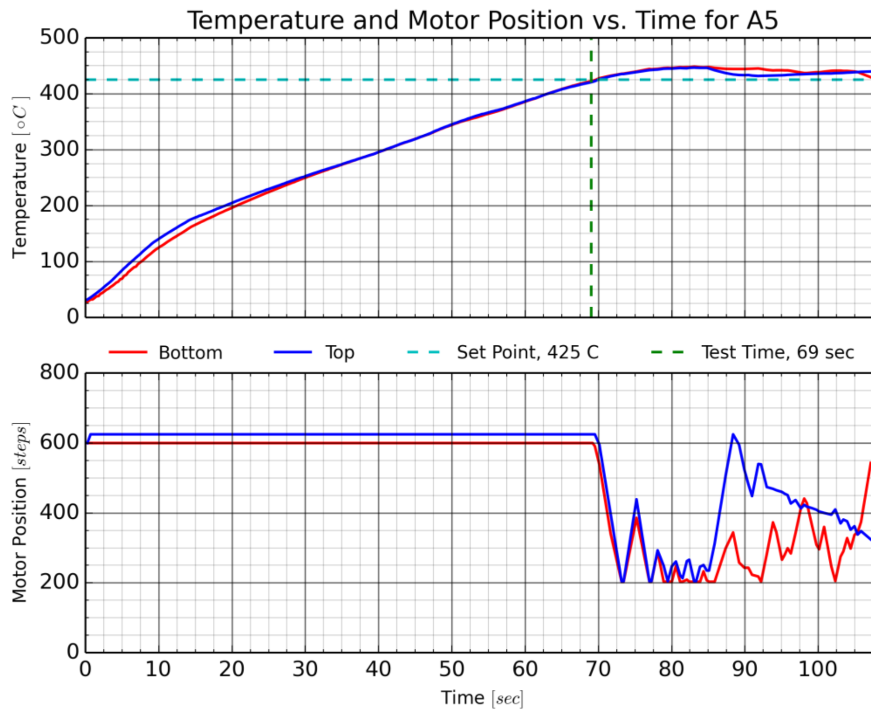


Figure 5.5: A5 TCS Data

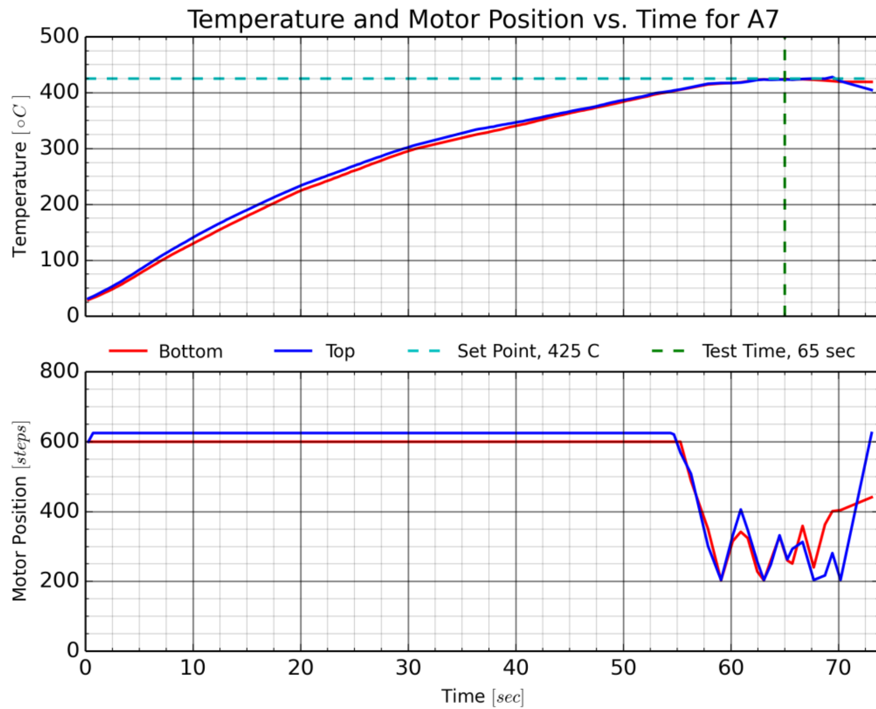


Figure 5.6: A7 TCS Data

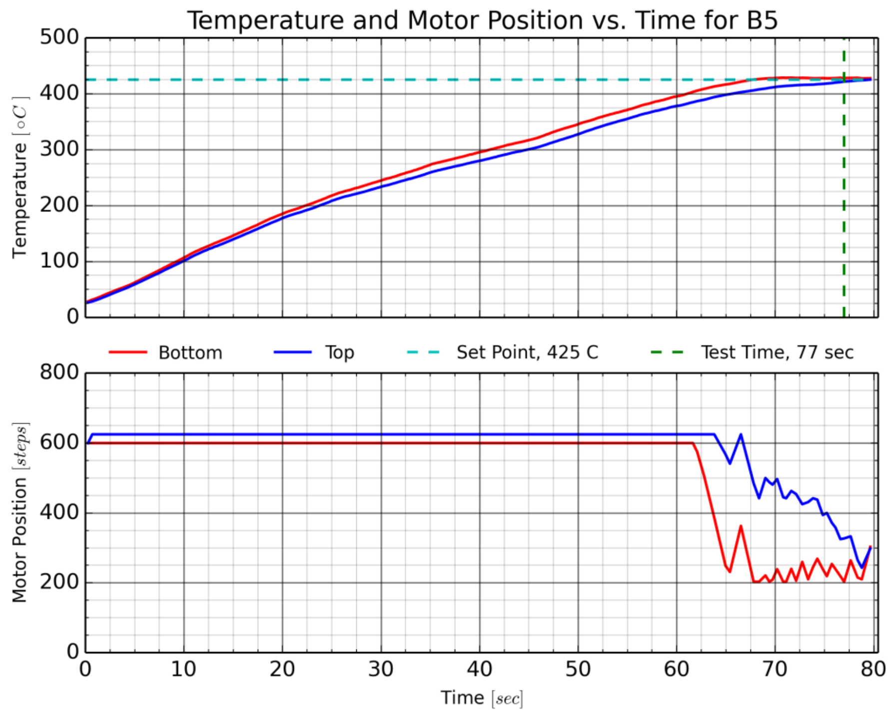


Figure 5.7: B5 TCS Data

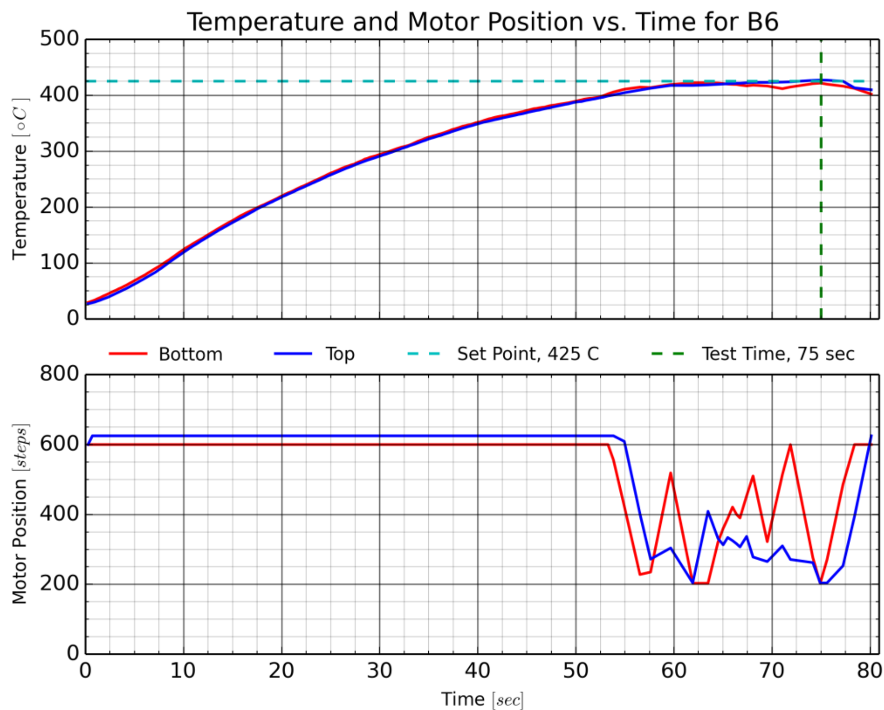


Figure 5.8: B6 TCS Data

Table 5.1 provides an overview of minimum and maximum motor positions, PID values, and average heat up rates for the 225°C and 425°C tests. Minimum motor positions remained the same for testing at both temperatures, while the maximum motor positions were shifted up 100 steps on both the top and bottom motors for the 425°C tests. This was done to initiate faster heating rates, due to the fact that heat up rates start to slow as the overall system capacity becomes hotter. As discussed in CHAPTER 2:, increasing P values results in faster system response times as well, however caution must be taken as this can also result in an unstable system. Thus the P value was increased from 15.00 to 18.00 for the 425°C tests. These changes produced an average heat up rate of 6.0°C/sec for the four tests performed at 225°C, while the four 425°C tests averaged a heat up rate of 5.6°C/sec.

Table 5.1: PID Testing Parameters

Temp. (°C)	Min motor position Top (steps)	Bottom (steps)	Max motor position Top (steps)	Bottom (steps)	P	I	D	Average Heat Rate (°C/sec)
225	203	204	550	500	15.00	2.00	0.06	6.0
425	203	204	650	600	18.00	2.00	0.06	5.6

Figure 5.9 displays a standard deviation error bar plot for heating rates during the 225°C and 425°C tests. It can be seen that the 225°C tests exhibited a much tighter, repetitive range of heating rates than the 425°C tests. 225°C heat up rates ranged from 5.9-6.1°C/sec, while 425°C heat up rates ranged from 5.2-6.1°C/sec. Overlap of the two error bars indicate that mean heating rates for both temperatures have a likely propensity to fall within the range of 5.9-6.1°C/sec.

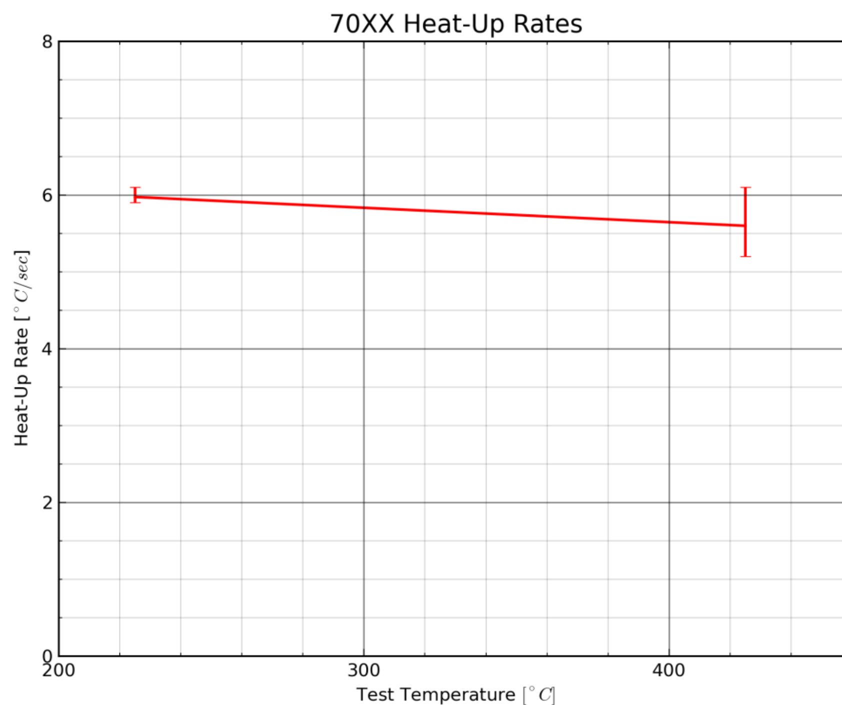


Figure 5.9: Heat-Up Rate Error Bar Plot

5.2 TENSILE RESULTS; $\dot{\epsilon}$ = 0.05/S

Stress strain curves for a strain rate of 0.05/sec at 25°C, 225°C, and 425°C are displayed in Figure 5.10. The plot serves to illustrate the effects temperature has on material properties. This is can be seen in the steepness of the elastic region, UTS magnitudes, and strain values. Note that specimen A5 displays a greater strain than the other tests. This is due to an incorrectly entered integral value and resulting temperature overshoot as mentioned in Section 5.1. Mechanical properties and other test specific information for the 0.05/sec tests are tabulated in Table 5.2. Looking at the table it is clear that an increase in temperature causes a decrease in YS and UTS values, and an increase in percent elongations. This is particularly evident when comparing UTS values. For 25°C tests the average UTS is 157.2 MPa, 101.9 MPa at 225°C, and 39.4 MPa at 425°C.

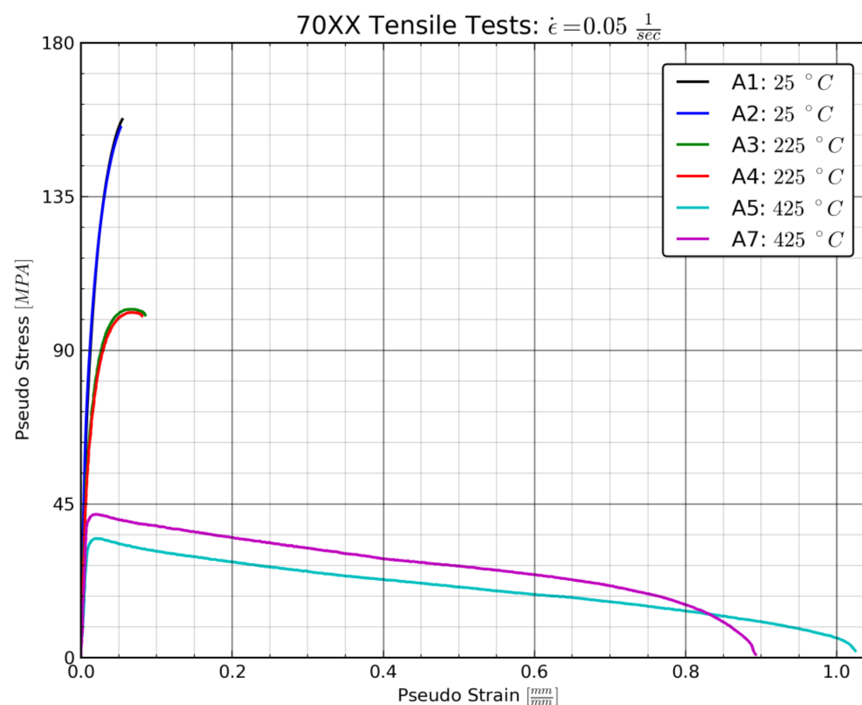


Figure 5.10: Stress Strain Curves for 0.05/sec Strain Rate

Table 5.2: 0.05 / sec Tensile Testing Results

Specimen	Test Temp. (°C)	0.2% YS (MPa)	UTS (MPa)	% Elongation	Heating rate (°C/sec)	Temp. at Tensile Test Start Top (°C)	Bottom (°C)
A1	25	71.6	158.2	5.4			
A2	25	76.6	156.2	5.6			
A3	225	60.6	102.2	8.6	6.1	224.00	219.00
A4	225	58.2	101.5	8.4	5.9	225.00	221.25
A5	425	33.0	35.3	126.2	5.8	422.50	424.25
A7	425	40.2	42.4	131.9	6.1	423.50	424.25

5.3 TENSILE RESULTS; $\dot{\epsilon}$ = 0.5/S

Stress strain curves for a strain rate of 0.5/sec for 25°C, 225°C, and 425°C are shown in Figure 5.11. This plot also displays the effect of temperature on material properties. Table 5.3 displays the mechanical properties and other information relevant to the elevated temperature portion of the tests. From both the stress strain curves and tabulated data it can be seen that that an increase in temperature causes a decrease in YS and UTS values, while increasing percent elongation. Again this is most visible when comparing UTS values. 25°C tests averaged a UTS value of 158.5 MPa, a UTS value of 109.0 MPa was averaged for the 225°C tests, and an average UTS value of 63.8 MPa was calculated for the 425°C tests.

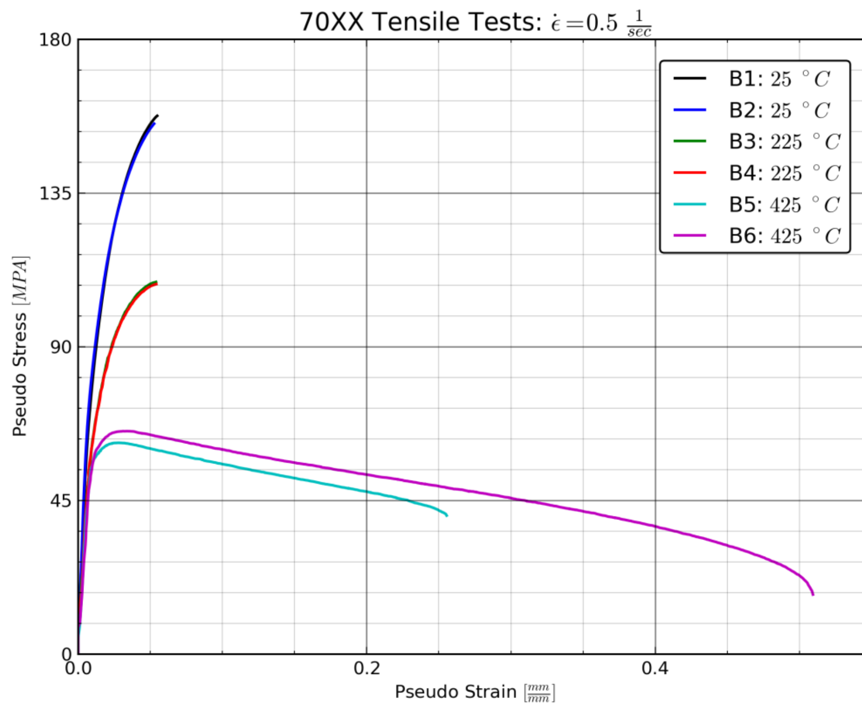


Figure 5.11: Stress Strain Curves for 0.5/sec Strain Rate

Table 5.3: 0.5 / sec Tensile Testing Results

Specimen	Test Temp. (°C)	0.2% YS (MPa)	UTS (MPa)	% Elongation	Heating rate (°C/sec)	Temp. at Tensile Test Start Top (°C)	Bottom (°C)
B1	25	75.3	157.1	5.4			
B2	25	72.9	159.9	4.8			
B3	225	60.2	109.3	5.2	6.0	224.00	225.25
B4	225	60.1	108.7	5.6	5.9	225.25	222.25
B5	425	56.8	62.1	49.7	5.2	421.75	428.00
B6	425	57.9	65.6	61.8	5.3	427.25	421.75

5.4 25°C TENSILE TESTING RESULTS

Stress strain plots for the 25°C tests are shown in Figure 5.12 and exhibit minimal to non-existent differences in material properties between the two strain rates. For a strain rate of

0.05/sec an average YS of 74.1 MPa, average UTS of 157.2 MPa, and an average elongation of 5.5% were calculated. At a strain rate of 0.5/sec the material exhibited an average YS of 74.1 MPa, an average UTS of 158.5 MPa, and an average elongation of 5.1%. Of the 25°C tests, specimen B2 shows the greatest pseudo strain, but has the smallest measured elongation at 4.8%. This is due to the specimen experiencing deformation outside of the gauge section in the shoulders where it seats into the grips.

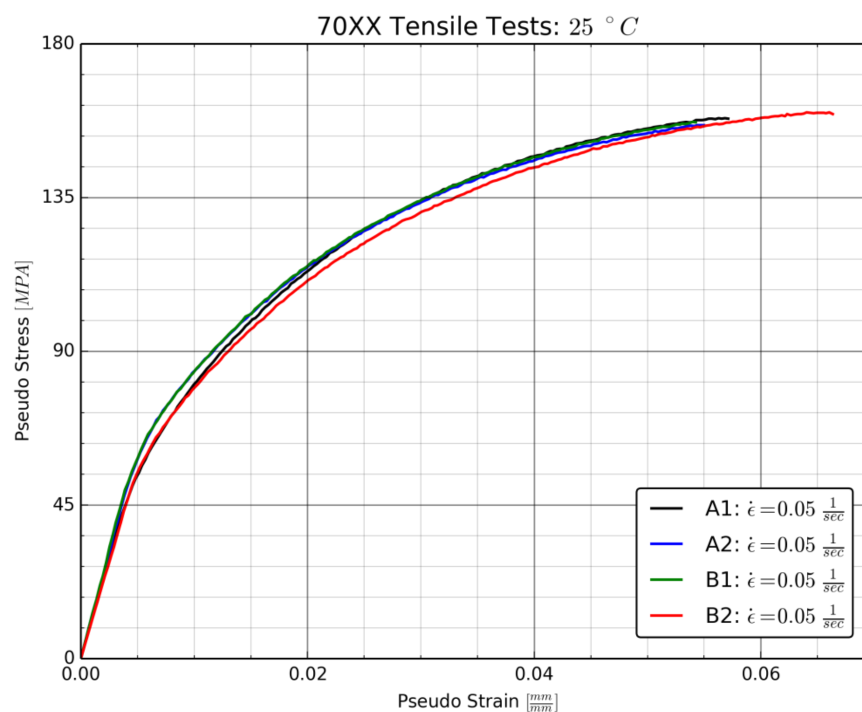


Figure 5.12: 25° C Stress Strain Curves

5.5 225°C TENSILE TESTING RESULTS

Stress strain plots for the 225°C tests are shown in Figure 5.13. From these plots it can be seen how strain rate impacts material properties at elevated temperature. Specimens tested at lower strain rates showed a significant increase in pseudo strain as well as an

increase in elongation. Specimens at 225°C and a strain rate of 0.05/sec averaged a YS of 59.4 MPa, UTS of 101.9 MPa, and an elongation of 8.5%. At a strain rate of 0.5/sec the tests averaged a YS of 60.2 MPa, UTS of 109.0 MPa, and an elongation of 5.6%. These results indicate that higher strain rates correlate to higher UTS values, while lower strain rates result in higher elongations. Looking at both Figure 5.12 and Figure 5.13 it can be seen that temperature also affects material property values. As temperature increases the YS and UTS decrease, while elongation increases.

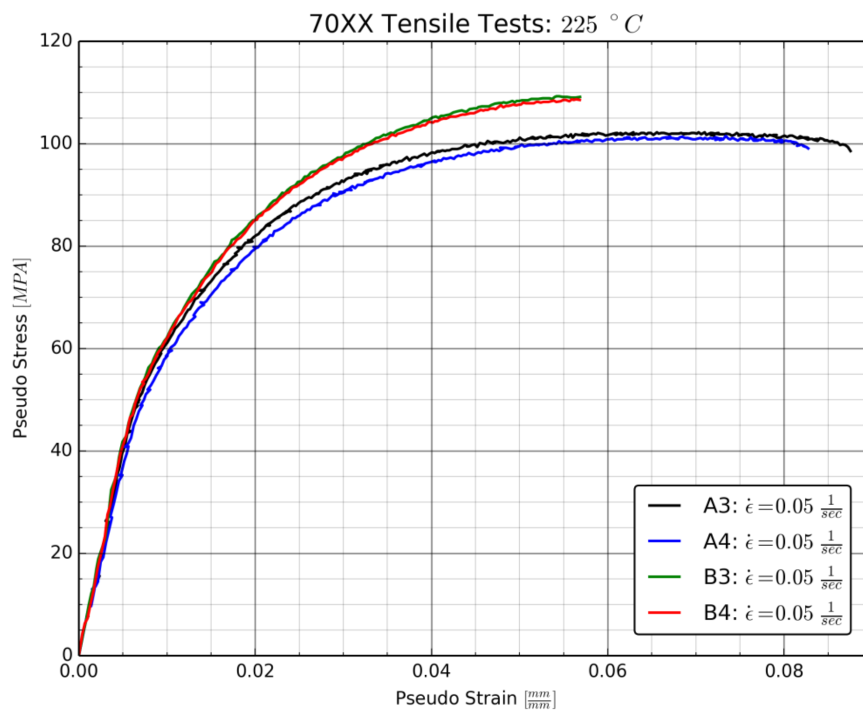


Figure 5.13: 225° C Stress Strain Curves

5.6 425°C TENSILE TESTING RESULTS

Stress strain results for the 425°C tests can be seen in Figure 5.14. Tests performed at a strain rate of 0.05/sec averaged a YS of 36.5 MPa, UTS of 39.4 MPa, and elongation of 129.0%. Testing at 0.5/sec resulted in an average YS of 57.4 MPa, UTS of 63.8 MPa, and elongation of 55.8%. Comparing these values to the 25°C and 225°C tests further demonstrates how YS and UTS values drop with higher temperatures while elongations increase. It should be noted however, that due to the selected material's elongation at high temperatures (400°C and greater), thermocouple attachment was difficult to maintain and inconsistent in reading temperatures when the thermocouples came loose as the specimen elongated and the gauge section reduced. Results are typically reliable for the type of thermocouple attachment described in this thesis, and are especially so throughout the specimen's elastic region in which there is negligible deformation, or until the thermocouples come loose. Testing for which there was minimal reduction in the gauge section, such as the 25°C and 225°C tests, provided more consistent results, as can be seen in Figure 5.12 and Figure 5.13. The results of these tests highlight how important it is to find a suitable method for thermocouple attachment or an alternative temperature measurement approach. It is also important to note that 425°C is near in temperature to the 70XX aluminum alloy semi-solid region. The semi-solid region occurs when temperatures are hot enough to melt portions of the aluminum alloy, while other sections remain solid. This formation of two phases at once in the alloy significantly weakens the material's ability to withstand applied loads and could affect testing results.

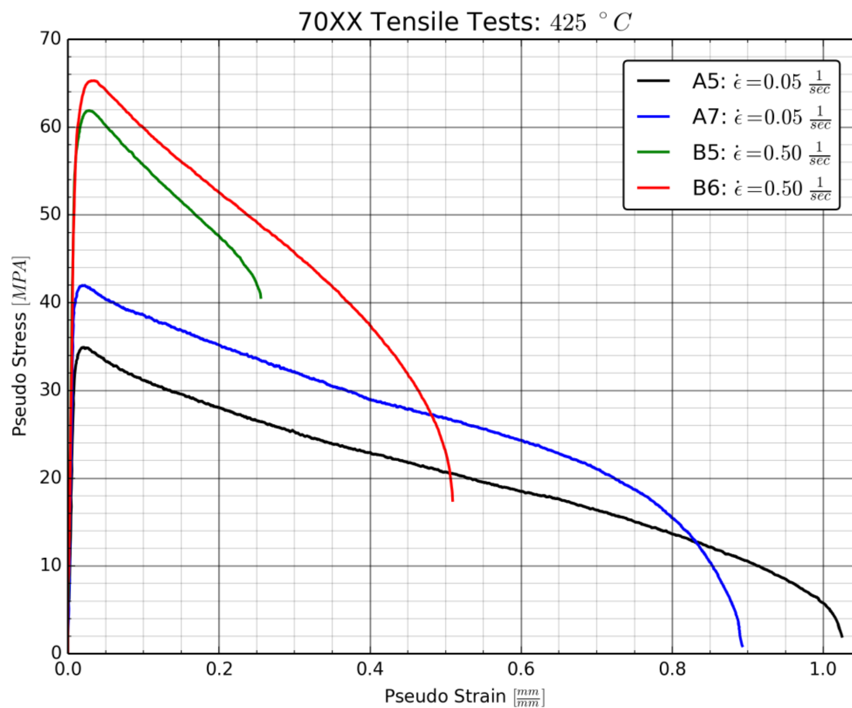


Figure 5.14: 425° C Stress Strain Curves

5.7 MECHANICAL PROPERTY ERROR BAR PLOTS

The following figures depict error bar plots for the evaluated YS, UTS, and percent elongation values. Red indicates tests ran at the 0.05/sec strain rate, while blue represents tests performed at the 0.5/sec test. From Figure 5.15 and Figure 5.16 it can be expected that for the 70XX series aluminum alloy, YS and UTS values recorded at lower temperatures will be fairly similar in value. During the 25°C and 225°C tests the range of recorded property values was quite minimal as evidenced from the plots. However, at higher temperatures, YS and UTS values see a change in behavior between the two strain rates. At 425°C greater material property values can be expected at the higher strain rate. Thus it can be concluded that a relationship exists between the material's properties and the effects of strain rate and temperature.

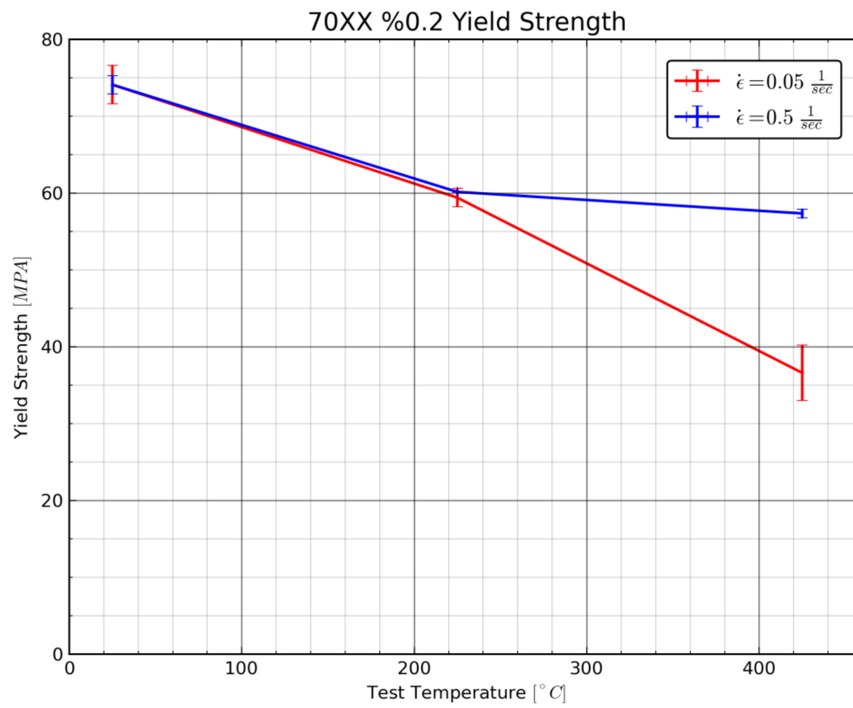


Figure 5.15: YS Error Bar Plot

Figure 5.16: UTS Error Bar Plot

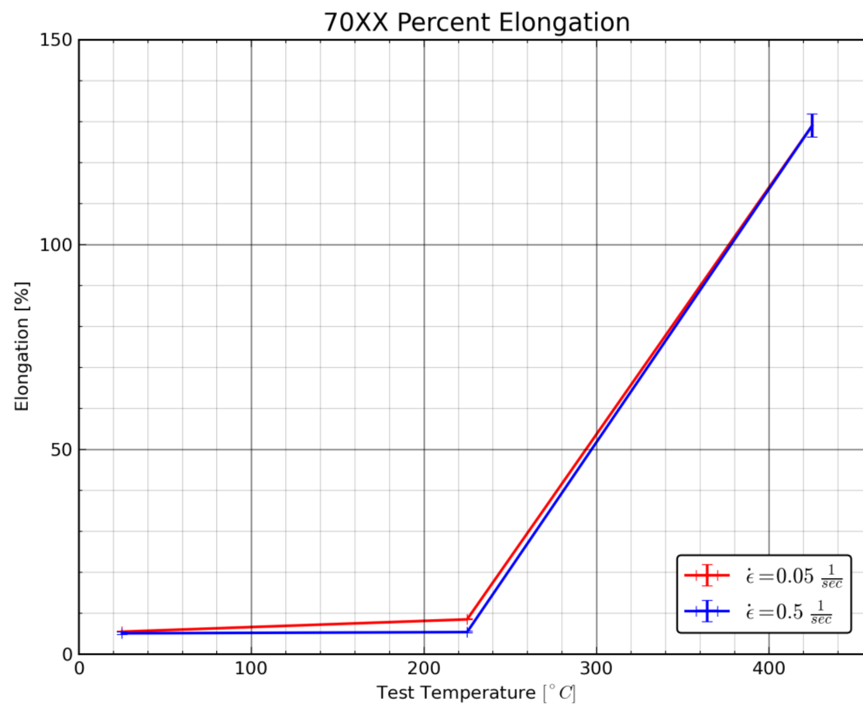


Figure 5.17: Percent Elongation Error Bar Plot

Figure 5.17 depicts a bar plot illustrating the calculated percent elongations for tests occurring at 25°C, 225°C, and 425°C and at strain rates of 0.05/sec and 0.5/sec. Again, the range of data at for each test situation is minimal, meaning there is not much variance from the mean and suggesting valid testing results. It can also be seen that the error bars overlap one another. This suggests that the material is less susceptible to variations in strain rate, especially at low temperatures.

CHAPTER 6: CONCLUSIONS AND RECOMMENDATIONS

6.1 CONCLUSIONS

The elevated temperature tensile testing apparatus described in this thesis provides a means for industry and academics to perform materials testing that conforms to standards and achieves fast heating rates, while at a significantly reduced cost to the user. The developed system consists of a servo-hydraulic test frame, capable of performing tensile tests, to which two propane torches have been affixed. These torches apply heat to both ends of the test specimen and are automated using stepper motors. The stepper motors are part of a temperature control loop and accept positional outputs from the PLC, while specimen temperature is monitored by two thermocouples and provides the PLC with inputs. Temperature control is regulated by a PID algorithm run by the PLC, which accepts system inputs such as set-point and PID settings from a GUI.

Operational verification of the apparatus and testing process was validated through testing of a 70XX series aluminum alloy at three different temperatures and two different strain rates. Twelve tensile tests were performed in total, eight of which were performed at elevated temperatures. It was found that increased temperature corresponds to increases in percent elongation measurements and decreases in YS and UTS values. All of the tests were performed according to ASTM standards, and testing occurred within +/-3°C of the temperature set-point. Heating rates also fell within the target range of 5-10°C, and averaged 6.0°C/sec for tests performed at 225°C and 5.8°C/sec for testing at 425°C. Costs for the system are a fraction of what other devices cost, totaling approximately $300.

6.2 RECOMMENDATIONS

The work presented in this thesis meets all design requirements. However, some improvements could be implemented to increase the functionality of the system. These improvements fall under one of five categories which will be discussed in this section: increasing system capacity, alternate testing platform validation, modifications to current technology, PID improvements, and improved temperature measurement.

6.2.1 INCREASING SYSTEM CAPACITY

Currently the testing apparatus' capabilities are limited to the intensity of the torch flame, and the capacity of the propane cylinders. Torch flame is directly related to the degree to which the stepper motor can open the pressure regulator without stalling. Introducing a motor with higher torque could increase the flame, resulting in higher heating rates. This was not investigated as the current motors provide sufficient flame intensities for the required testing. For prolonged testing it is recommended to develop an alternate fuel supply system. The current propane cylinders have a finite volume that required replacement. A larger fuel supply with a gauge to indicate container propane levels would be ideal.

6.2.2 VALIDATION FOR ALTERNATE TESTING PLATFORMS

In theory the developed temperature testing apparatus can be introduced to any testing platform. This was not investigated, and it would be interesting to see how changes to specimen material and geometry, and load train material and geometry affect heating

rates and system cohesion. More specifically, verification of system capabilities with respect to non-metallic specimen, and non-circular specimen should be completed.

6.2.3 MODIFICATIONS TO CURRENT TECHNOLOGY

The current testing procedure consists of starting the heat up process, and then starting the tensile test process when the temperature set-point has been reached. Testing times are recorded separately for these two actions and it is up to the user to record at what time the tensile test is started relative to the recorded temperature. Combining the two processes would be advantageous and could be accomplished by triggering a flag in the PLC program. Set-up would require connecting another sensor to the SainSmart Uno that would indicate when the tensile test begins. This could be implemented in the form of a circuit that is easily broken when the test frame actuator moves downward.

Another improvement would be to replace the current motor driver shield with one that communicates over serial and has its own PWM driver chip. This is because the current motor shield does not have its own processor, and is dependent upon the Uno to control motor position. This is extremely time intensive for the Uno, as it cannot send data to the GUI, or enter new temperatures into the PID control loop until the motors have moved to their next position. Because of this, cycle times for the PID loop are dependent upon how long it takes the motors to move from one position to the next. Implementing a shield such as the ‘Adafruit Motor/Stepper/Servo Shield for Arduino v2’ would improve cycle times, as well as free up the Uno to focus on reading temperatures, the PID loop, and updates for the next desired motor position. Another benefit is that the Adafruit motor shields are

stackable and use fewer I/O pins on the Uno so that the MAX31855 breakout boards could be directly connected.

6.2.4 PID IMPROVEMENTS

The existing PID control system is implemented in such a manner that P, I, and D values must be predetermined and entered into the control program before testing begins. In order to reduce time spent tuning parameters several adjustments to the algorithm can be explored. Developing a mathematical process model to simulate heating of the test specimen and load train would reduce time spent testing heating and cooling of the system. However, this model may require unavailable parameters, such as the thermal conductivity of the specimen. Intensive study of system parameters including flame temperature, thermal conductivity of the grip material, and system heat sinks would need to be acquired. Alternatively the temperature control algorithm could be edited to accept new PID parameters during testing. This would provide the user with the ability to manually change parameters on the fly if they noticed the system was not behaving in a satisfactory manner. A third improvement to the PID control algorithm would be to introduce a cascading PID loop. Because initiating a change to the bottom specimen temperature results in a change to the top temperature, and vice versa, a cascading PID loop would be ideal. Cascading PID loops consist of two PID loops, with one controlling the set-point of another. For implementation with respect to temperature control of the testing apparatus, an initial PID loop would control the specimen’s top temperature and a secondary PID loop would control the specimen’s bottom temperature. The top

temperature PID would operate based off a user defined set-point, while the bottom PID would operate with a set-point defined by the top temperature.

6.2.5 IMPROVED TEMPERATURE MEASUREMENTS

As noted in 5.6 accurate temperature measurement became difficult to sustain at higher temperatures. The stainless steel thermocouple clips used to ensure a secure contact between the thermocouples and test specimen could not account for the reduced diameter of the 70XX aluminum alloy when it elongated at 425°C. To improve temperature measurement at these elevated temperatures it is recommended to implement an alternative thermocouple attachment method. The new method must be able to account for the significant change in geometry that the test specimen undergoes during elongation. Several suggestions include spot welding the thermocouples to the specimen or using a high temperature adhesive compound. Both methods may prove more costly, but should provide the desired results. Alternatively, a non-contact method of temperature measurement could be implemented such as an infrared (IR) camera.

REFERENCES

[1] Smith, Brian, "The Boeing 777," *Advanced Materials & Processes,* vol. 161, no. 9, pp. 41-44, 2003.

[2] J. Martin-Bermejo, G. Van Goethem and M. Hugon, "Research Activities on High-temperature Gas-cooled Reactors (HTRs) in the 5th Euratom RTD Framework Programme," in *The Second Information Exchane Meeting on Basic Studies in the Field of High-temperature Engineering*, Paris, 2001.

[3] Z. Li, J. Xu and E. Bai, "Static and Dynamic Mechanical Properties of Concrete After High Temperature Exposure," *Materials Science and Engineering,* vol. 544, pp. 27-32, 2012.

[4] B. Santillana, R. Boom, D. Eskin, H. Mizukami, M. Hanao and M. Kawamoto, "High Temperature Mechanical Behavior and Fracture Analysis of a Low-Carbon Steel Related to Cracking," *Metallurgical and Materials Transactions A,* vol. 43A, pp. 5048-5057, 2012.

[5] W. Kasprzak, B. S. Amirkhiz and M. Niewczas, "Structure and properties of cast Al-Si based alloy," *Journal of Alloys and Compounds,* vol. 595, pp. 67-79, 2013.

[6] R. Cobden and et. all, "Aluminum: Physical Properties, Characteristics and Alloys," *Training in Aluminum Application Technologies,* vol. 1.0, pp. 1-60, 1994.

[7] D. R. Askeland and P. P. Phule, The Science and Engineering of Materials, Stamford, CT: CENGAGE Learning, 2008.

[8] K. P. Shah, "Plastic Deformation and Fracture," Practical Maintenance, 01 May 2009. [Online]. Available: http://practicalmaintenance.net/?page_id=38. [Accessed 29 December 2014].

[9] K. G. Hoge, "Influence of Strain Rate on Mechanical Properties of 6061-T6 Aluminum under Uniaxial and Biaxial States of Stress," *Experimental Mechanics,* vol. 6, no. 4, pp. 204-211, 1966.

[10] A. Dorbane, G. Ayoub, B. Mansoor, R. Hamade, G. Kridli and A. Imad, "Observations of the Mechanical Response and Evolution of Damage of AA 6061-T6 Under Different Strain Rates and Temperatures," *Materials Scence & Engineering ,* vol. 624, no. A, pp. 239-249, 2015.

[11] "Standard Test Methods for Elevated Temperature Tension Tests of Metallic Materials," in *Annual Book of ASTM Standards*, ASTM Standard E21, 1999.

[12] E. J. Hearn, Mechanics of Materials, Volume 1 : An Introduction to the Mechanics of Elastic and Plastic Deformation of Solids and Structural Materials, Oxford: Butterworth-Heinemann, August 1997.

[13] R. I. Stephens, A. Fatemi, R. R. Stephens and H. O. Fuchs, Metal Fatigue in Engineering, New York : John Wiley & Sons, Inc., 2001.

[14] R. F. Legget, "AMERICAN SOCIETY FOR TESTING AND MATERIALS," *NATURE,* vol. 203, no. 4945, pp. 565-568, 1964.

[15] G. Vigilante, S. Bartolucci, J. Izzo, M. Witherell and S. Smith, "Gleeble Testing to Assess Solid/Liquid Metal Embrittlement of Gun Steels by Copper," *Materials and Manufacturing Processes,* vol. 27, pp. 835-839, 2012.

[16] N. L. Lindeman, "Technique For Applying Direct Resistance Heating Current to a Specific Location in a Specimen Under Test While Substantially Reducing Thermal Gradients in the Specimen Gauge Length". United States Patent 7,363,822 B2, 29 April 2008.

[17] Todd Bonesteel, "Physical Simulation Speeds Product Development," *Advanced Materials & Processes,* vol. 164, no. 12, pp. 37-39, 2006.

[18] "Gleeble Systems: The Standard for Thermal-Mechanical Physical Simulation," Dynamic Systems Inc., 2014. [Online]. Available: http://www.gleeble.com/index.php/products.html. [Accessed 23 July 2013].

[19] "66--Gleeble 3500 Thermo Mechanical Simulator," 04 August 2010. [Online].

Available: https://www.neco.navy.mil/synopsis_file/N00167-10-P-0250_Redacted_JA.pdf. [Accessed 09 October 2014].

[20] T. U. o. C. Town, "New Machine to Ease Metal Processing Studies," *Monday Paper,* p. 2, May-June 2012.

[21] B. F. I. f. M. R. a. Testing, "Tensile Tests at High Temperatures up to 1900C," 4 March 2013. [Online]. Available: http://www.bam.de/en/kompetenzen/fachabteilungen/abteilung_5/fg52/fg52_medien/fg52_ht-zugversuch_englisch.pdf. [Accessed 23 March 2015].

[22] S. Zinn, I. Harry, R. Jeffress and S. Semiatin, Elements of Induction Heating: Design, Control, and Applications, Novelty: ASM International, 1998.

[23] *Proposal,* Scottsville: Ambrell Precision Induction Heating, 2014.

[24] L. B. Blackburn and J. R. Ellingsworth, "Tensile Testing Apparatus". United States of America Patent 4,535,636, 19 March 1984.

[25] Idaho National Laboratory, "High Temperature Test Laboratory," Idaho National Laboratory, [Online]. Available: https://inlportal.inl.gov/portal/server.pt/community/distinctive_signature__icis/315/httl. [Accessed 4 April 2015].

[26] E. A. Wilcox, Electric Heating, New York: McGraw-Hill Book Company, Inc. , 1928.

[27] Watlow, "Mineral Insulated (MI) Band Heaters," Watlow, 2014. [Online]. Available: https://www.watlow.com/products/heaters/mineral-insulated-band-heaters.cfm. [Accessed 12 October 2014].

[28] W. Y. Svrcek, D. P. Mahoney and B. R. Young , A Real-Time Approach to Process Control, West Sussex: John Wiley & Sons, Ltd, 2014.

[29] J. Hogenson, "PID for Dummies," Control Solutions, Inc., 2010. [Online]. Available: http://www.csimn.com/CSI_pages/PIDforDummies.html. [Accessed 9 October 2014].

[30] H. L. Wade, Basic and Advanced Regulatory Control: System Design and Application, Durham: The Instrumentation, Systems and Automation Society, 2004.

[31] P. Fry, "Specialty Gas Regulators - How They Work," CAC Gas & Instruments, 27 October 2013. [Online]. Available: http://www.cacgas.com.au/blog/bid/344734/Specialty-Gas-Regulators-How-They-Work. [Accessed 26 September 2014].

[32] "SainSmart UNO ATMEGA328P-PU ATMEGA8U2 Microcontroller For Arduino," SainSmart, 2010. [Online]. Available: http://www.sainsmart.com/sainsmart-uno-atmega328p-pu-atmega8u2-microcontroller-for-arduino.html. [Accessed 30 08 2014].

[33] Arduino, "Arduino Uno," Arduino, 2014. [Online]. Available: http://arduino.cc/en/Main/arduinoBoardUno. [Accessed 2 October 2014].

[34] "SainSmart L293D Motor Drive Shield For Arduino Duemilanove Mega UNO R3 AVR ATMEL," SainSmart, 2014. [Online]. Available: http://www.sainsmart.com/sainsmart-l293d-motor-drive-shield-for-arduino-duemilanove-mega-uno-r3-avr-atmel.html. [Accessed 10 October 2014].

[35] L. Fried, "Adafruit Motor Shield," 12 May 2014. [Online]. Available: https://learn.adafruit.com/downloads/pdf/adafruit-motor-shield.pdf. [Accessed 10 October 2014].

[36] "Thermocouple Amplifier MAX31855 Breakout Board (MAX6675 Upgrade) - v2.0," Adafruit, 2014. [Online]. Available: http://www.adafruit.com/product/269. [Accessed 10 October 2014].

[37] E. Bendersky, "A "live" data monitor with Python, PyQt and PySerial," Eli Bendersky's website, 7 August 2009. [Online]. Available: http://eli.thegreenplace.net/2009/08/07/a-live-data-monitor-with-python-pyqt-and-pyserial. [Accessed 14 October 2014].

[38] E. Bendersky, "plotting_data_monitor," GitHub, Inc., 2014. [Online]. Available: https://github.com/eliben/code-for-blog/tree/master/2009/plotting_data_monitor.

[Accessed 14 October 2014].

[39] Q. Project, "Qt Project," Qt Project, 2014. [Online]. Available: http://qt-project.org/. [Accessed 14 October 2014].

[40] R. C. Limited, "What is PyQt4," Riverbank Computing Limited, 2013. [Online]. Available: http://www.riverbankcomputing.co.uk/software/pyqt/intro. [Accessed 14 October 2014].

[41] J. Bodnar, "Events and Signals in PyQt4," ZetCode, 6 October 2013. [Online]. Available: http://zetcode.com/gui/pyqt4/eventsandsignals/. [Accessed 14 October 2014].

[42] E. Bendersky, "eblib Source," Google Project Hosting, 20 September 2010. [Online]. Available: https://code.google.com/r/arthur19891106-eli/source/browse/eblib_dir/#eblib_dir%2Feblib. [Accessed 14 October 2014].

[43] P. S. Foundation, "Queue-A Synchronized Queue Class," Python Software Foundation, 2014. [Online]. Available: https://docs.python.org/2/library/queue.html. [Accessed 14 October 2014].

[44] C. Liechti, "pySerial," pySerial, 2013. [Online]. Available: http://pyserial.sourceforge.net/pyserial.html#overview. [Accessed 14 October 2014].

[45] C. R. Simcoe, "The Discovery of Strong Aluminum," *Advanced Materials & Processes,*

vol. 169, no. 8, pp. 35-36, 2011.

[46] Bossi, Richard H., "NDE DEVELOPMENTS FOR COMPOSITE STRUCTURES," in *AIP Conference Proceedings*, Ipswich, 2006.

[47] Y. Liu and S. Kumar, "Recent Progress in Fabrication, Structure, and Properties of Carbon Fibers," Taylor & Francis Group, LLC, Atlanta, 2012.

[48] "Arduino," Arduino, 2014. [Online]. Available: http://arduino.cc/. [Accessed 30 08 2014].

[49] K. Motzfeldt, "High Temperature Experiments in Chemistry and Materials Science," John Wiley & Sons, 2013.

[50] M. McRoberts, Beginning Arduino, New York: Apress, 2013.

[51] "ARDUINO," Arduino, 2014. [Online]. Available: http://arduino.cc/en/Guide/Introduction. [Accessed 17 03 2014].

[52] D. McWhan, Sand and Silicon, Oxford: Oxford University Press, 2012.

[53] F. P. Incropera, D. P. Dewitt, T. L. Bergman and A. S. Lavine, Fundamentals of Heat and Mass Trainsfer, Danvers: John Wile & Sons, 2007.

[54] M. Ashby, H. Shercliff and D. Cebon, Materials-Engineering, Science, Processing and

Design, Oxford: Elsevier, 2007.

[55] K. Motzfeldt, High Temperature Experiments in Chemistry and Materials Science, Chichester: John Wiley & Sons, Ltd., 2013.

[56] D. Polka, Motors and Drives - A Practical Technology Guide, Durham: The Instrumentation, Systems, and Automation Society, 2003.

[57] B. Zhang, "On Typical Materials Acting as the Dividing Standard of the Development Stages of Human Substance Civilization," *Interdisciplinary Description of Complex Systems,* vol. 10, no. 2, pp. 114-126, 2012.

[58] S. Li, *Hot Tearing in Cast Aluminum Alloys: Measures and Effects of Process Variables,* Worcester: Worcester Polytechnic Institute, 2010.

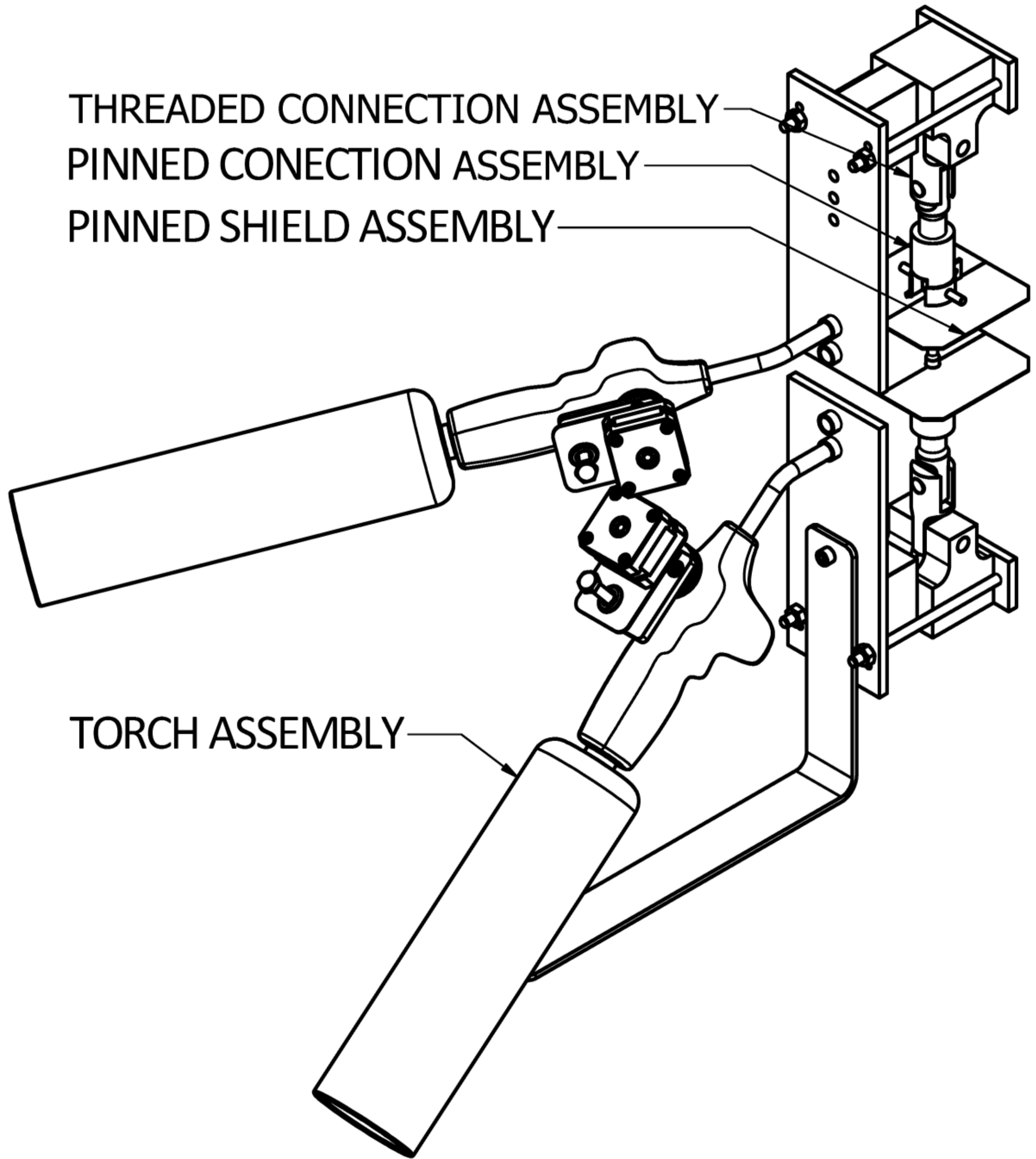
[59] B. S. Mitchell, Materials Engineering and Science for Chemical and Materials Engineers, Hoboken: John Wiley & Sons, Inc., 2004.

[60] "Stepper Motor with Cable," Cana Kit Corporation, 2014. [Online]. Available: http://www.canakit.com/stepper-motor-with-cable-rob-09238.html. [Accessed 11 October 2014].

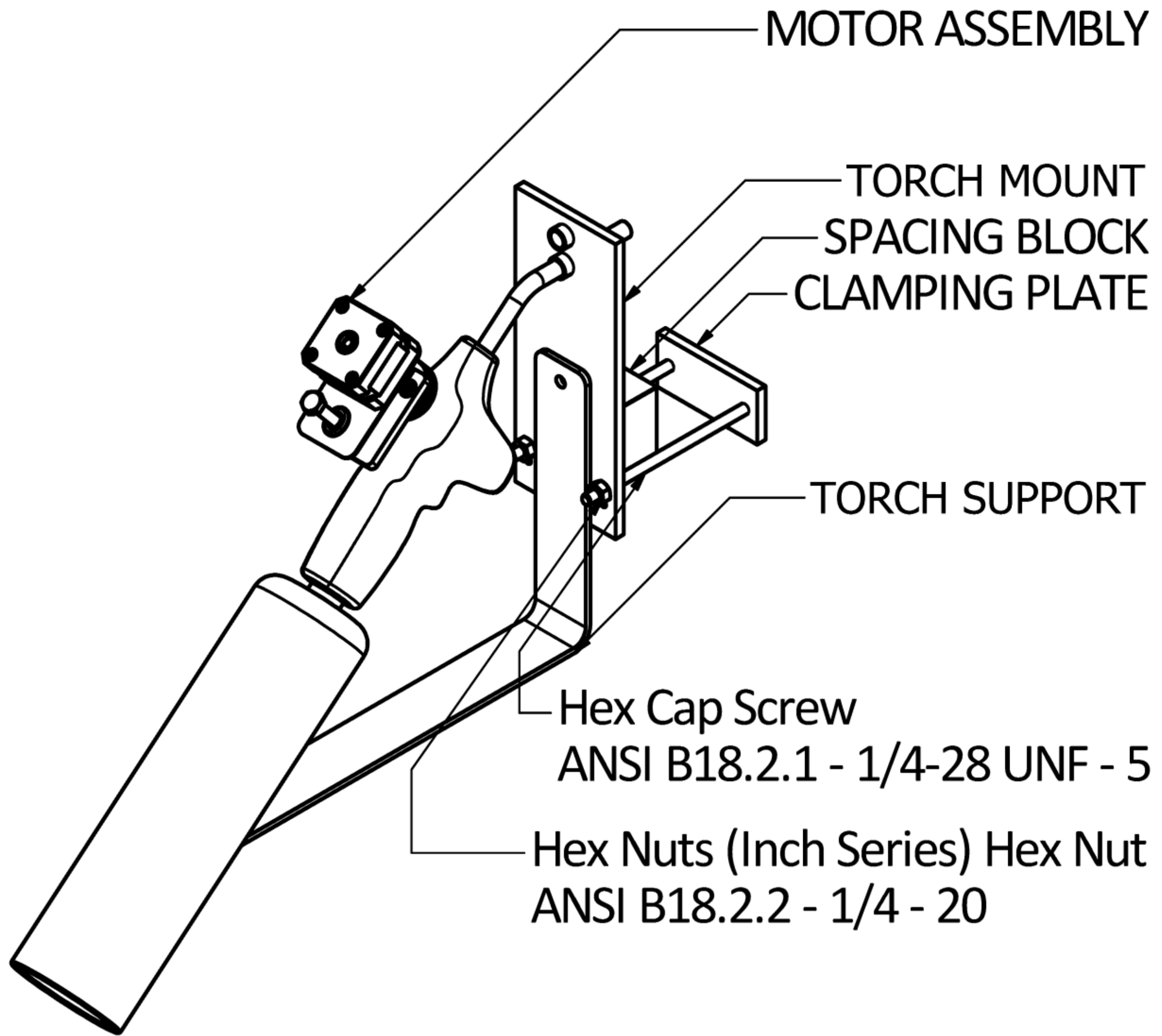
[61] B. Beauregard, "Arduino PID Library," Arduino, 2014. [Online]. Available: http://playground.arduino.cc/Code/PIDLibrary. [Accessed 13 October 2014].

[62] P. Raybaut, "Ecosystem Pverview," Python(x,y), March 209. [Online]. Available: http://pythonxy.googlecode.com/files/python_2722.png. [Accessed 2014].

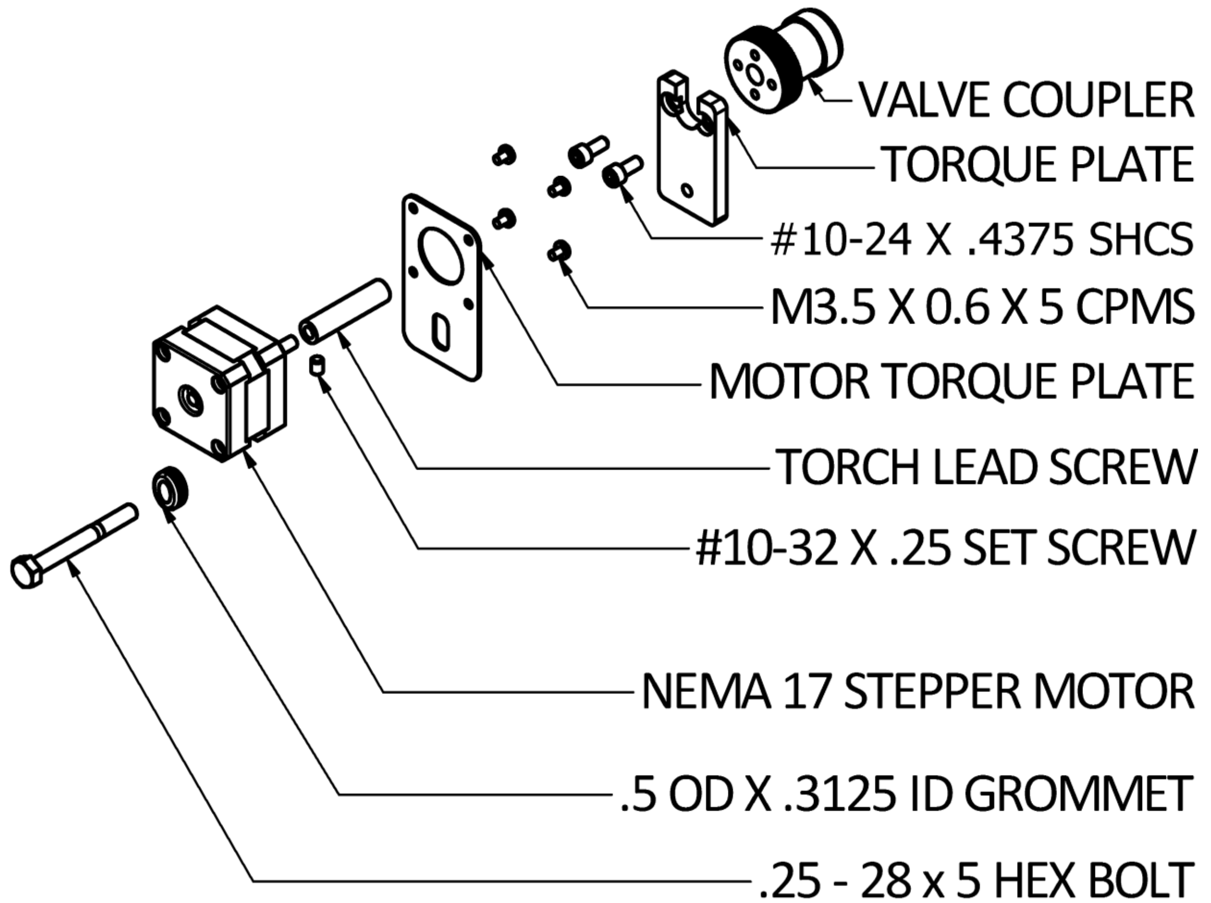
APPENDIX A: DRAWING PACKAGE



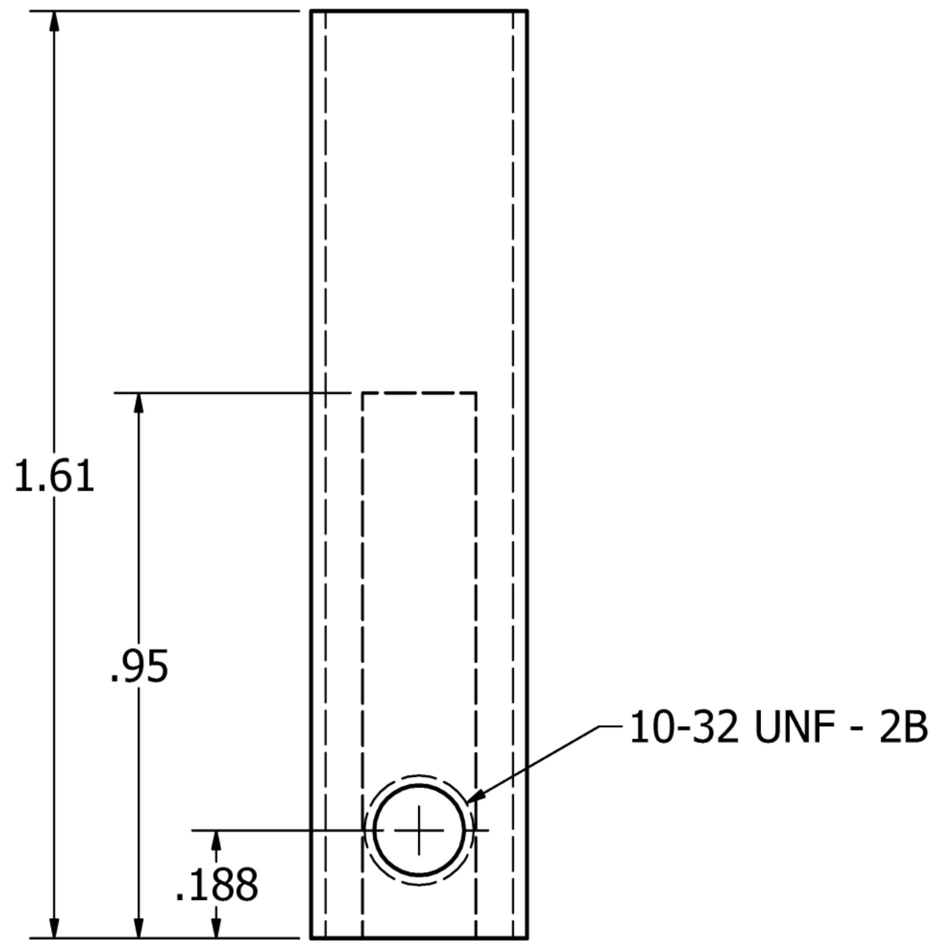
DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: MULTIPLE
SCALE: 4:1	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 1 of 24	PART NAME: PINNED SYSTEM ASSEMBLY	

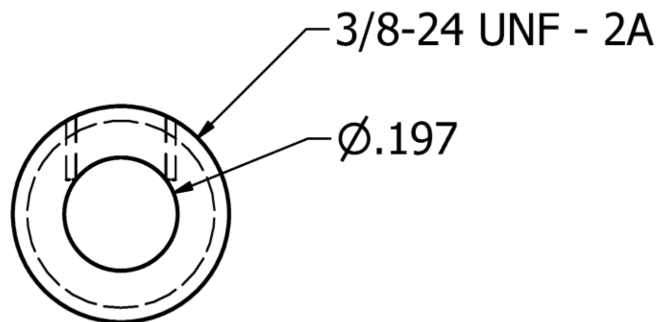


DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: MULTIPLE
SCALE: 4:1	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 2 of 24	PART NAME: TORCH ASSEMBLY	

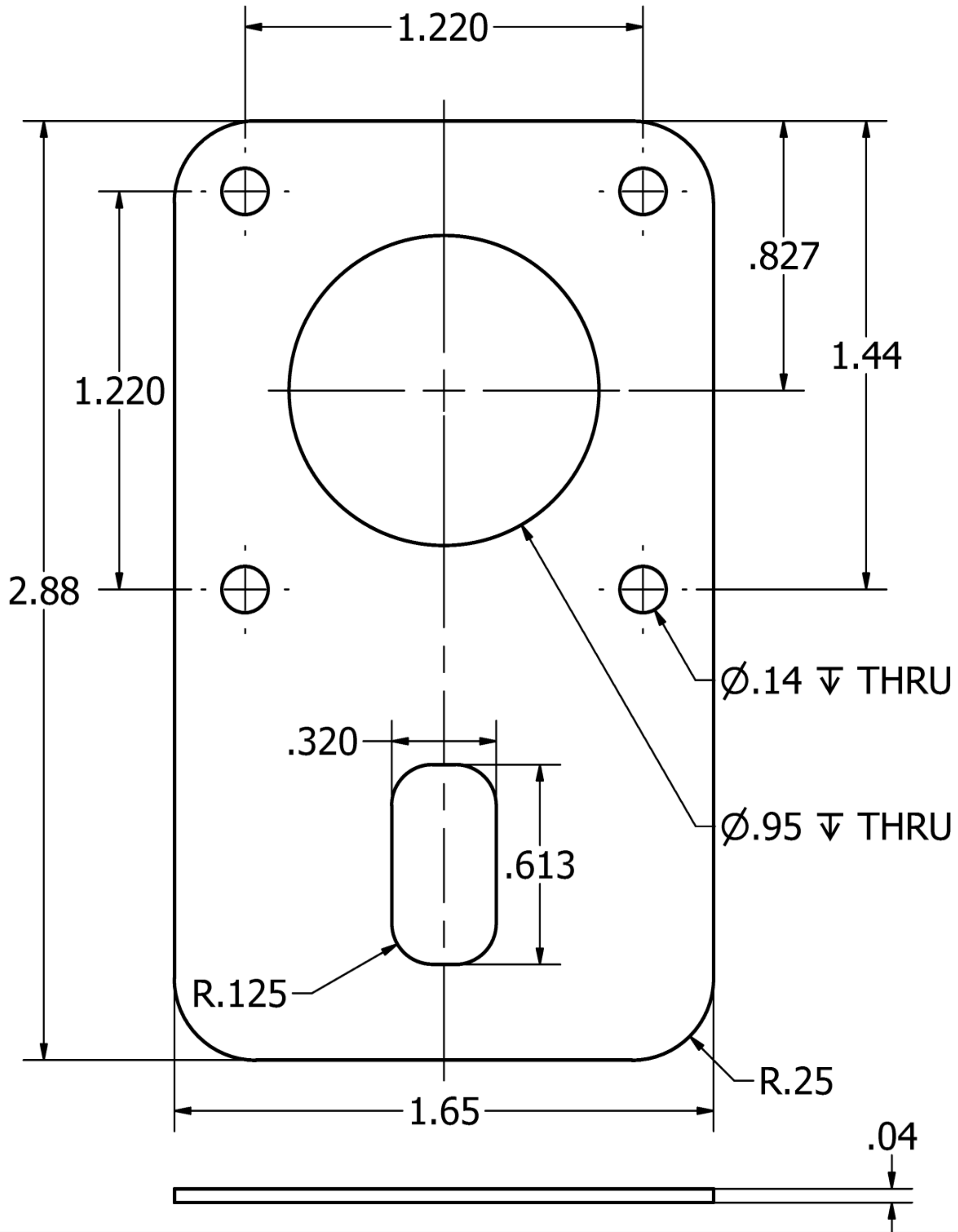


DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: MULTIPLE
SCALE: 1:1	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 3 of 24	PART NAME: MOTOR ASSEMBLY	

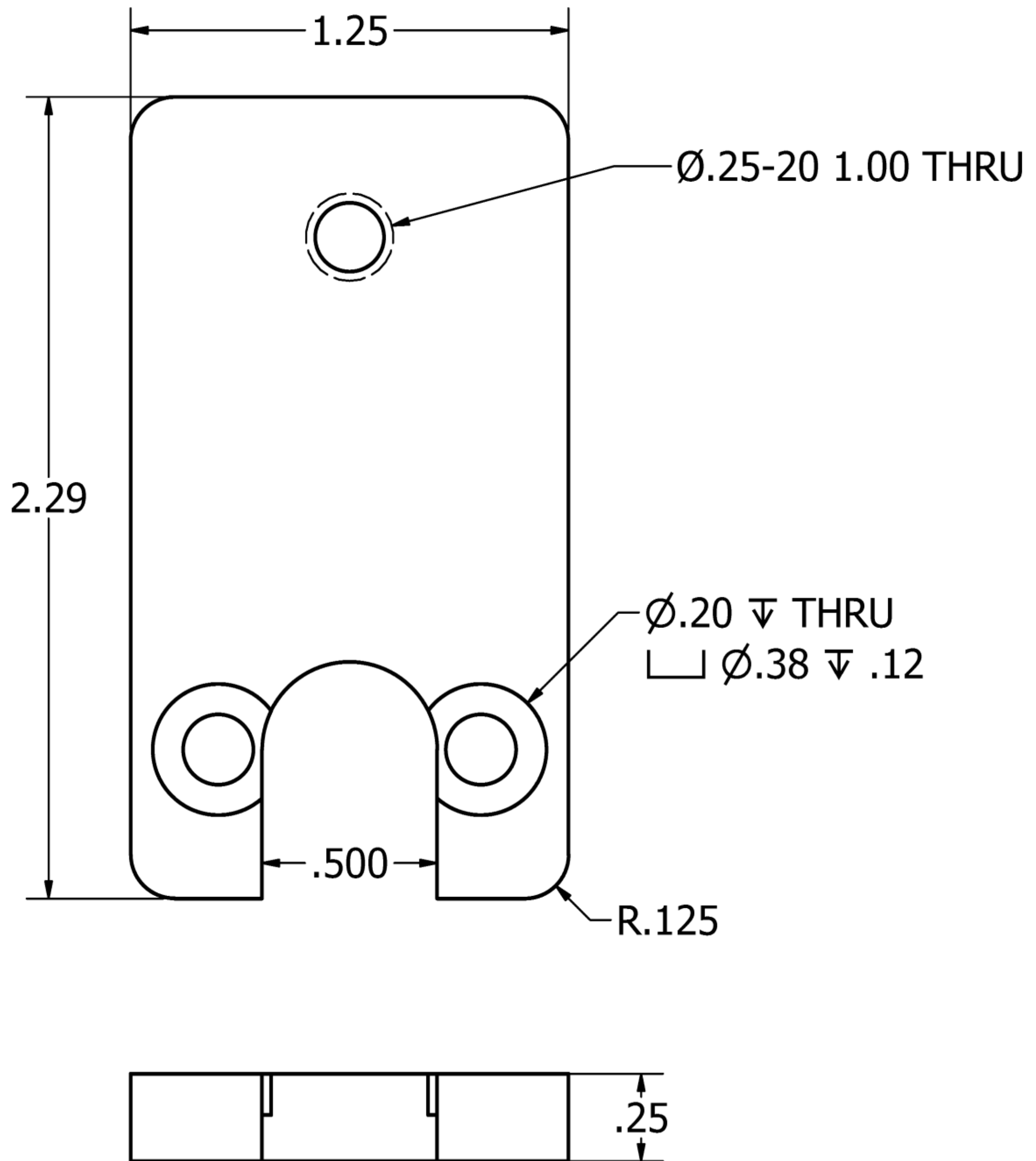




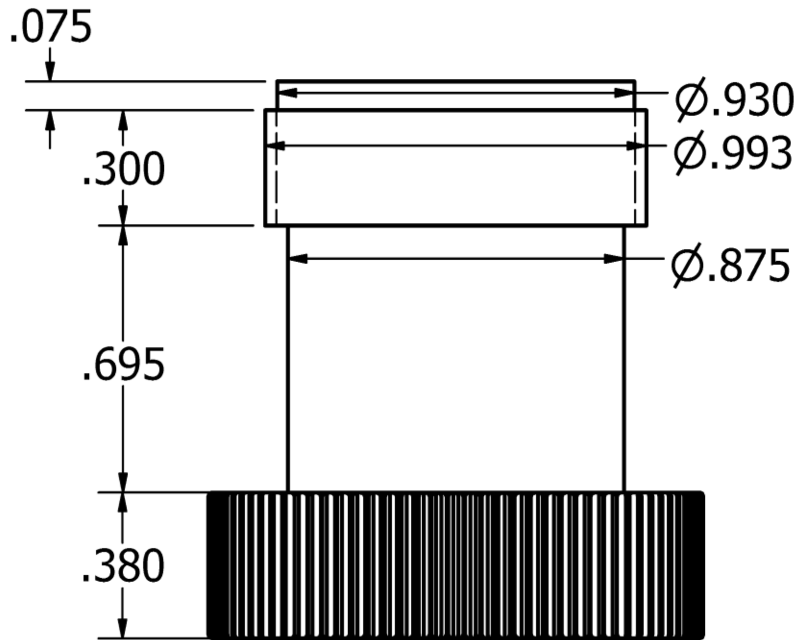
DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: Stainless Steel
SCALE: 3:1	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 4 of 24	PART NAME: TORCH LEAD SCREW	

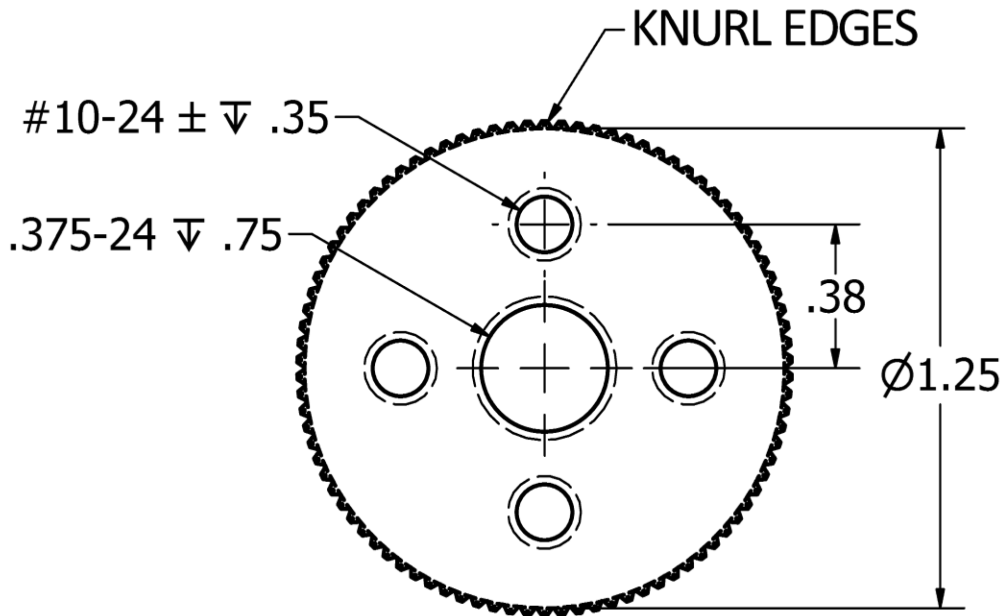


DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: Aluminum 6061
SCALE: 2:1	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 5 of 24	PART NAME: MOTOR TORQUE PLATE	

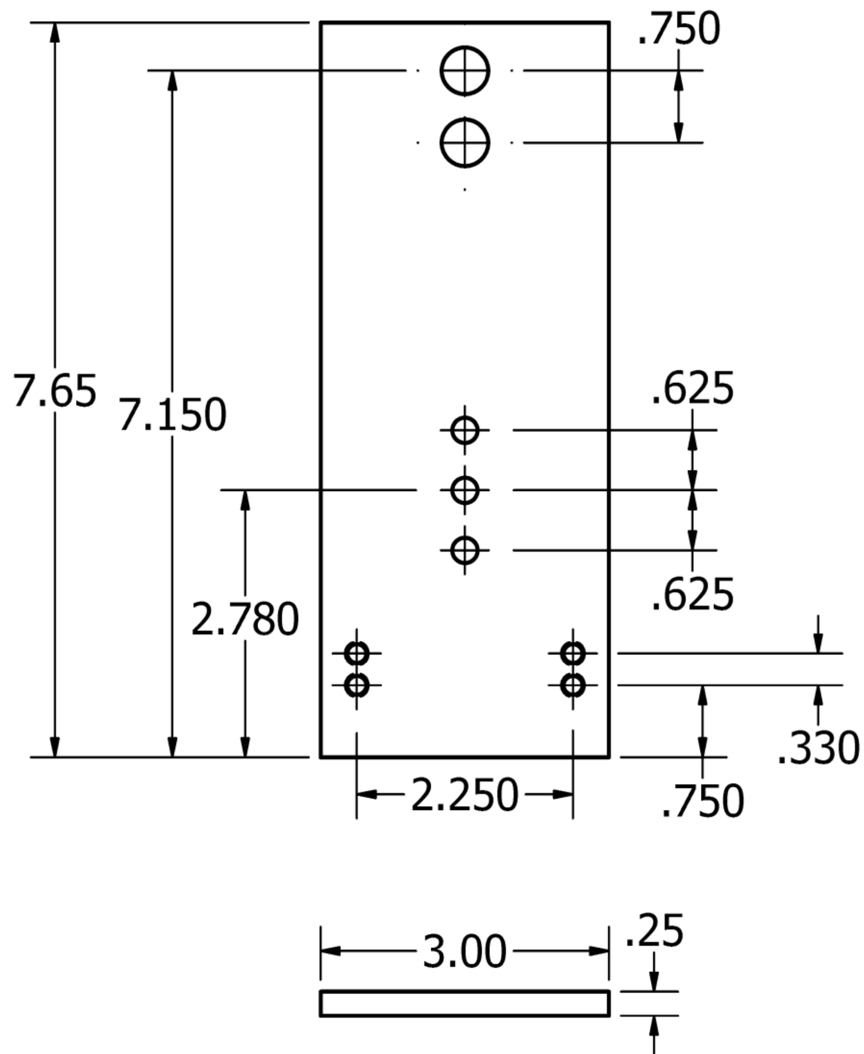


DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: Aluminum 6061
SCALE: 2:1	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 6 of 24	PART NAME: TORQUE PLATE	

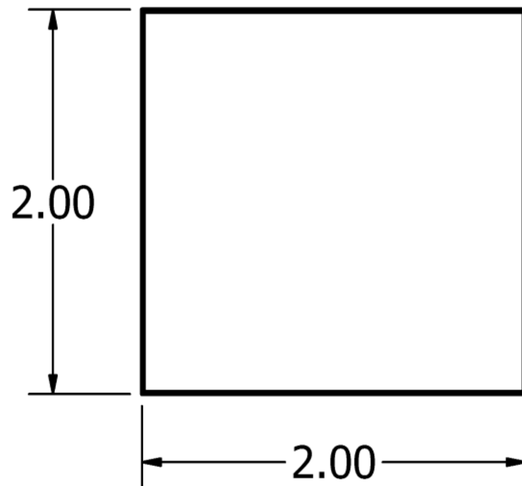


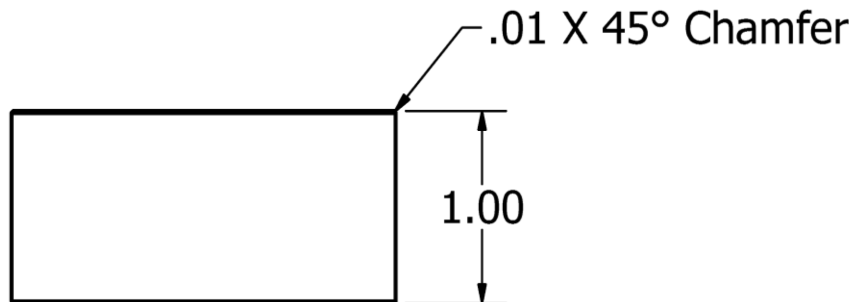


DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: Aluminum 6061
SCALE: 2:1	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 7 of 24	PART NAME: VALVE COUPLER	

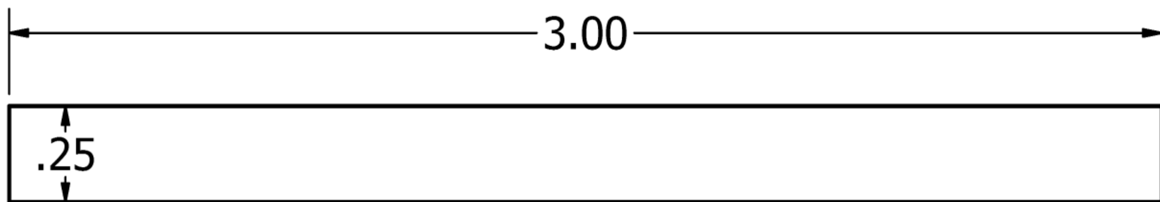


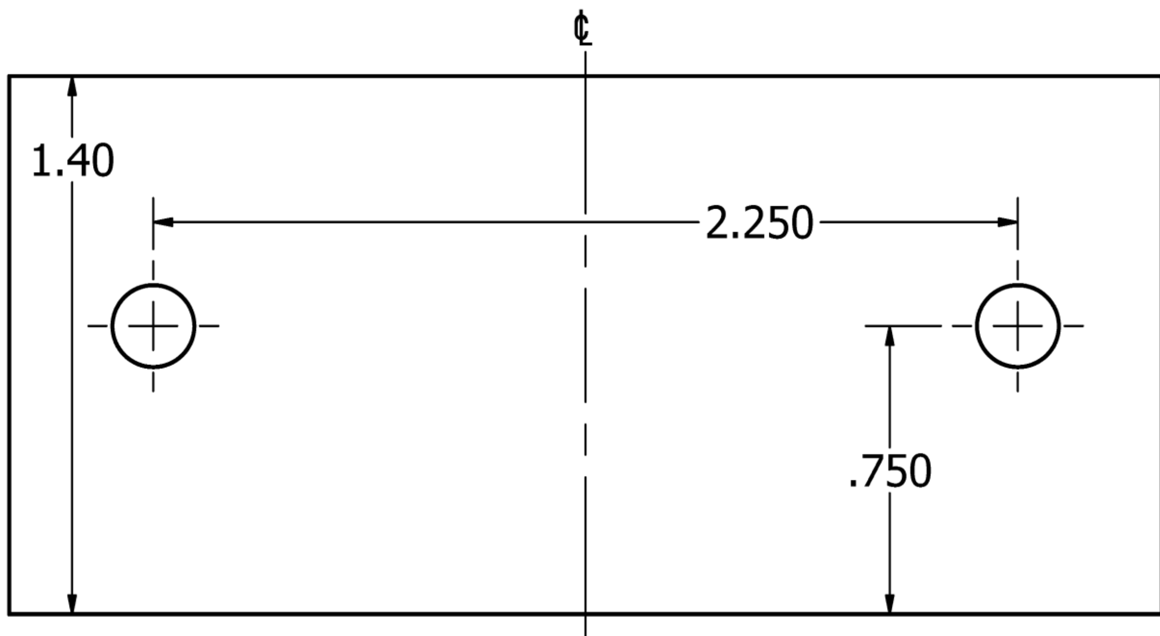
DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: Steel
SCALE: 2:1	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 8 of 24	PART NAME: TORCH MOUNT	



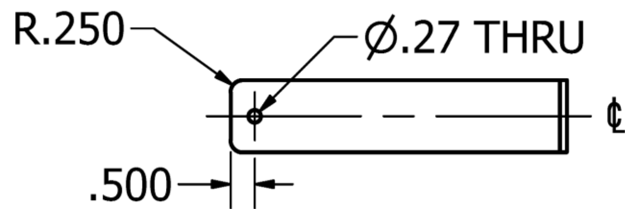


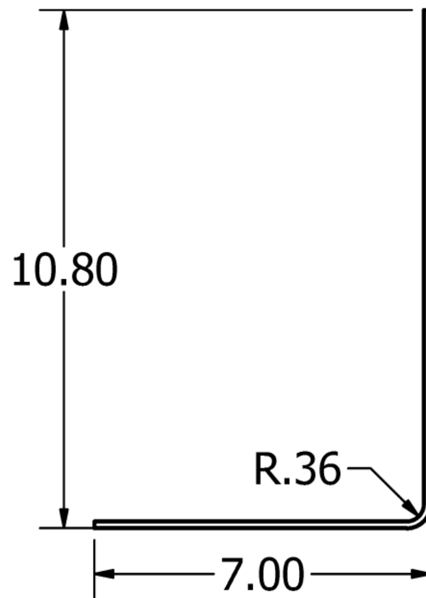
DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: Steel
SCALE: 1:1	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 9 of 24	PART NAME: SPACING BLOCK	



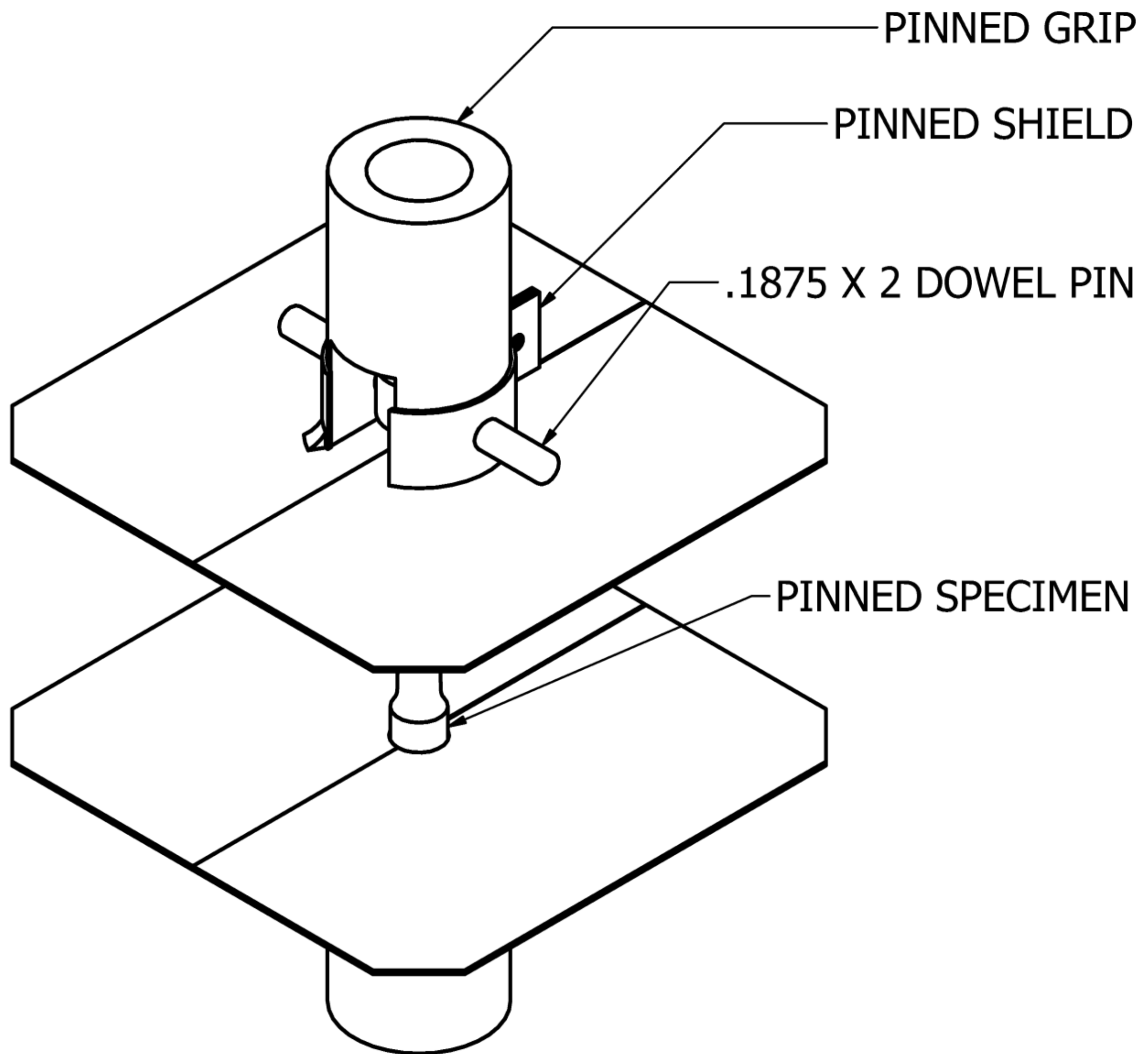


DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: Steel
SCALE: 2:1	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 10 of 24	PART NAME: CLAMPING PLATE	

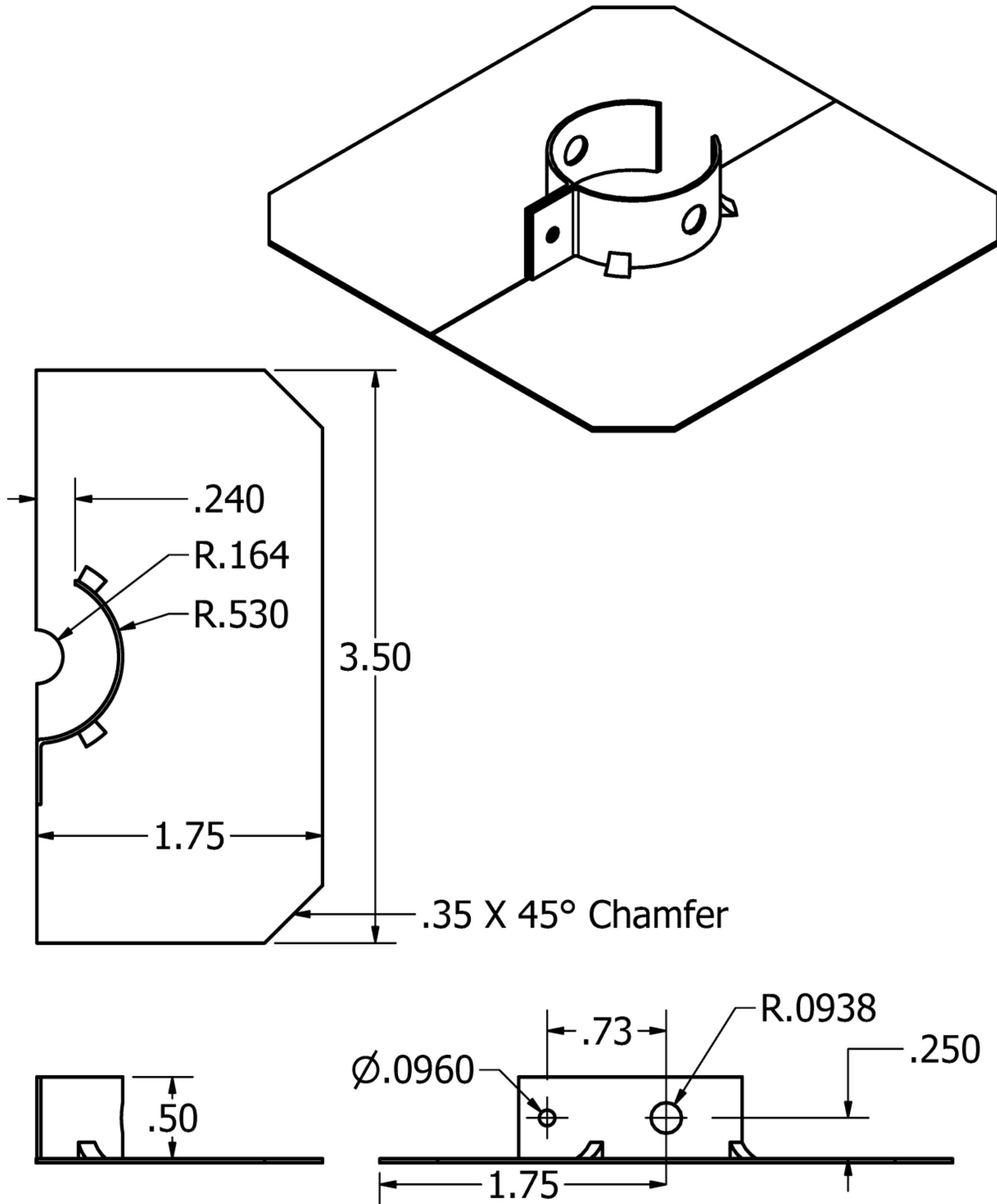




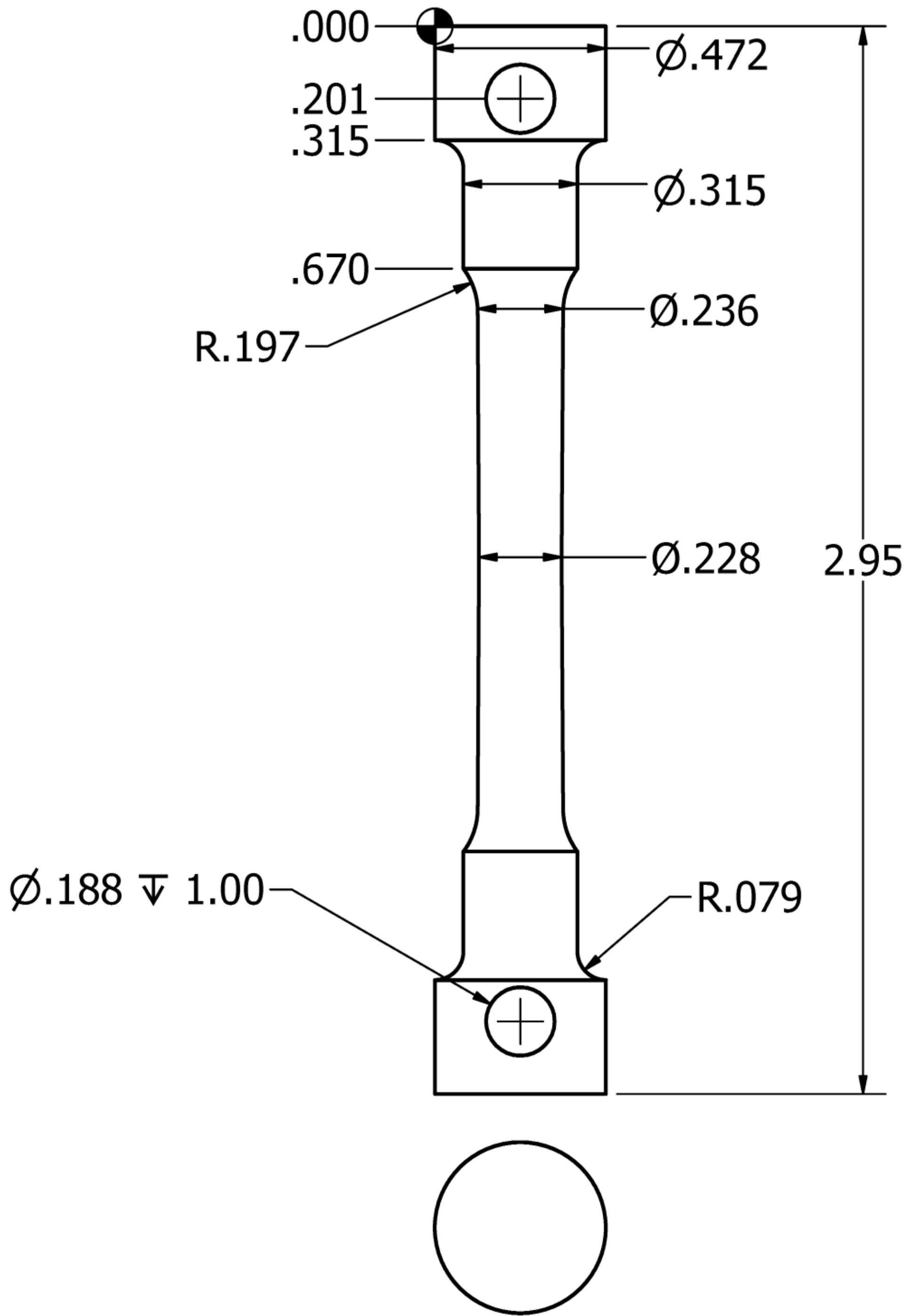
DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: Steel
SCALE: 1/4	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 11 of 24	PART NAME: TORCH SUPPORT	



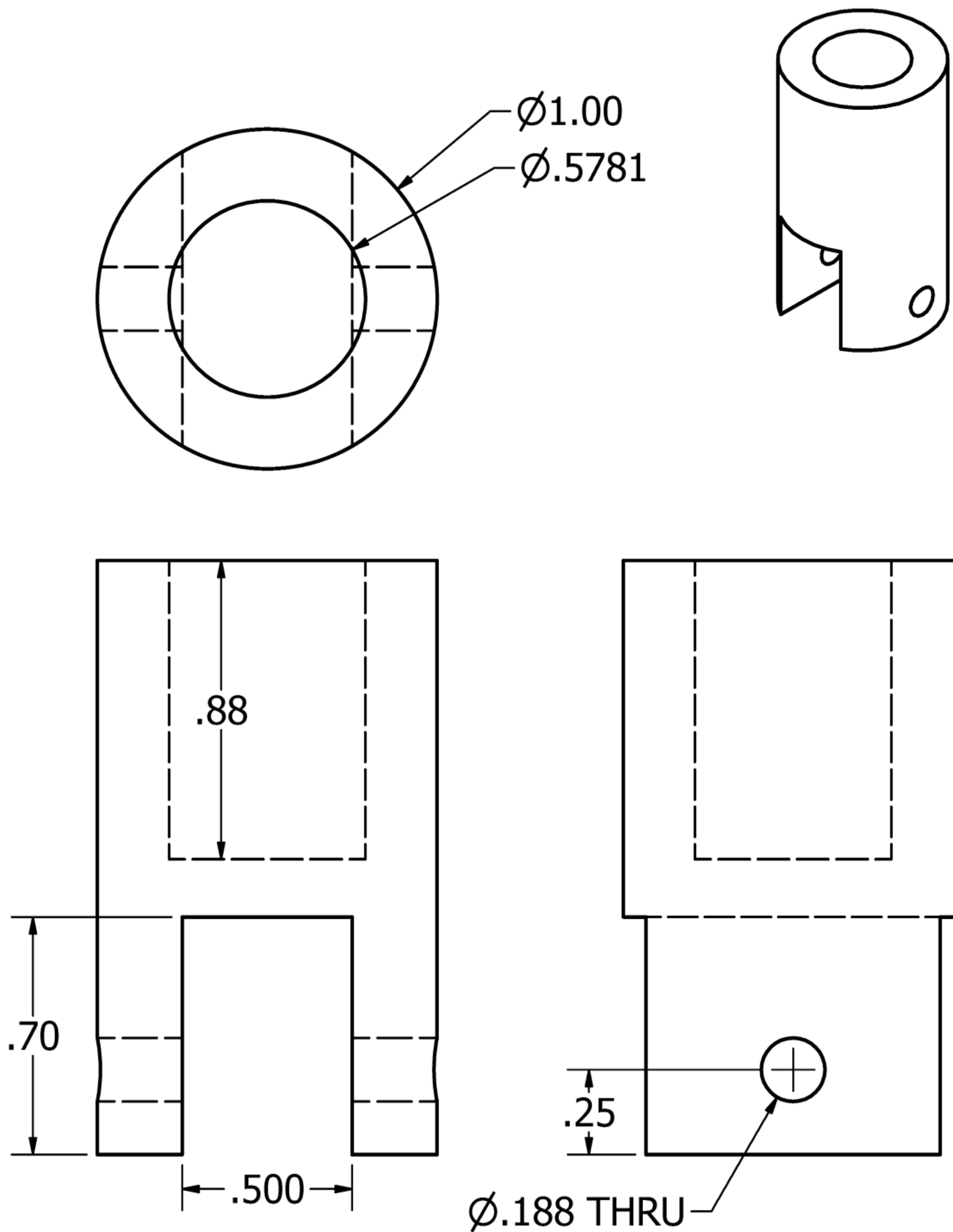
DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: MULTIPLE
SCALE: 1:1	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 12 of 24	PART NAME: PINNED CONNECTION ASSEMBLY	



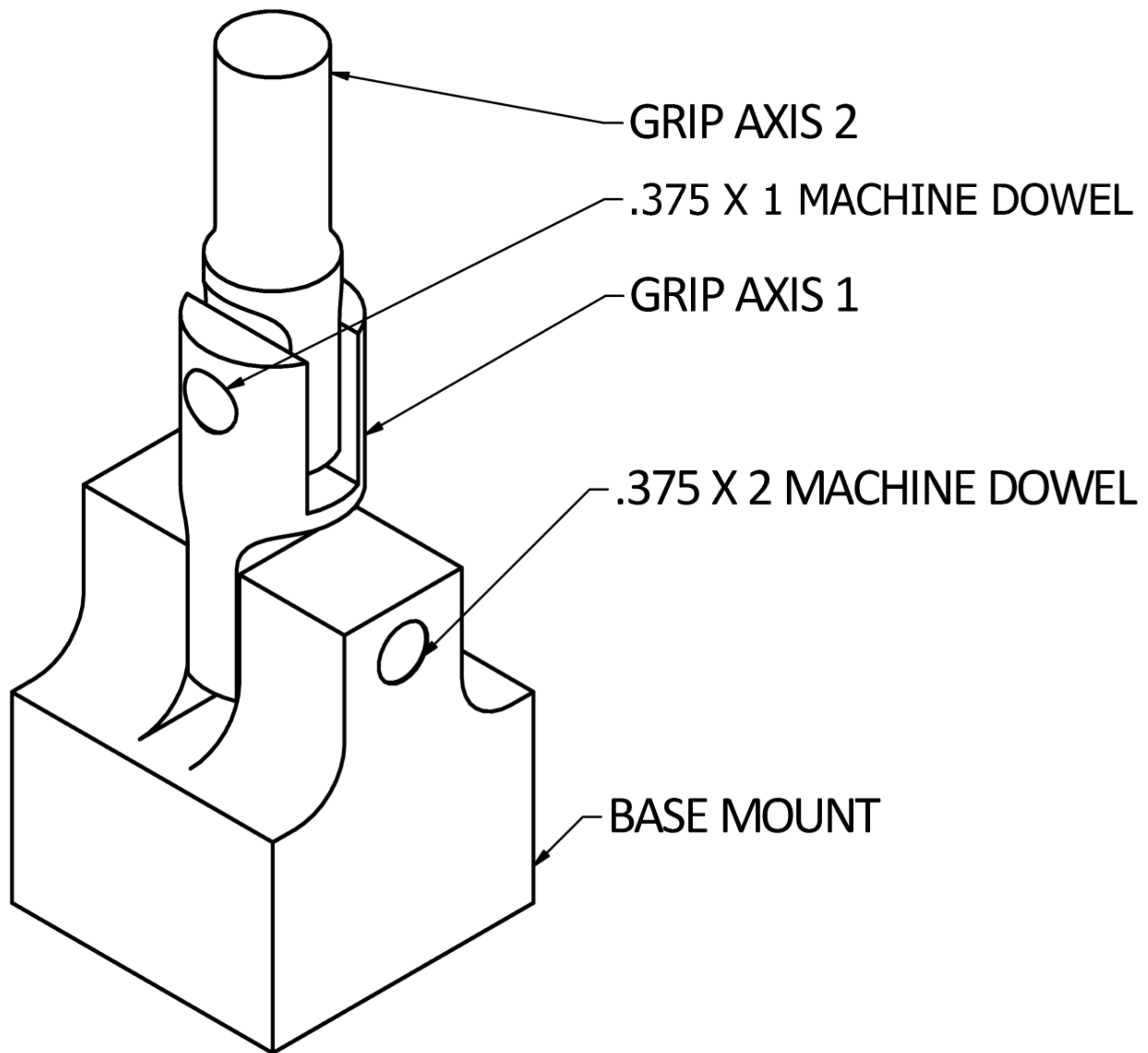
DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: Stainless Steel
SCALE: 1:1	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 13 of 24	PART NAME: PINNED SHIELD	



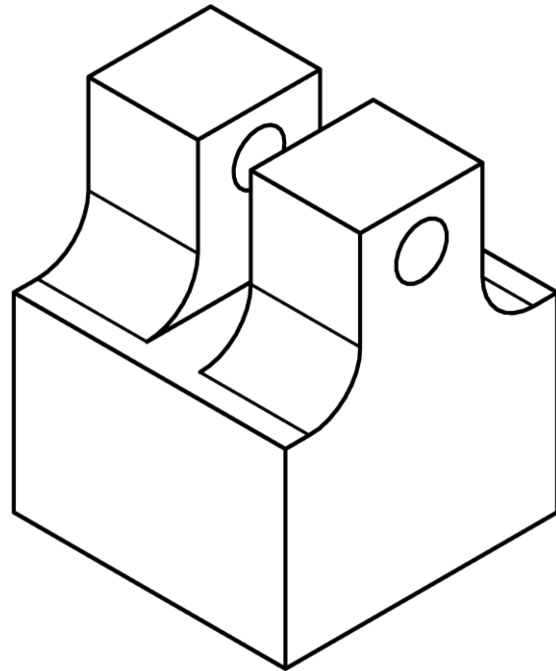
DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: NA
SCALE: 2:1	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 14 of 24	PART NAME: PINNED SPECIMEN	

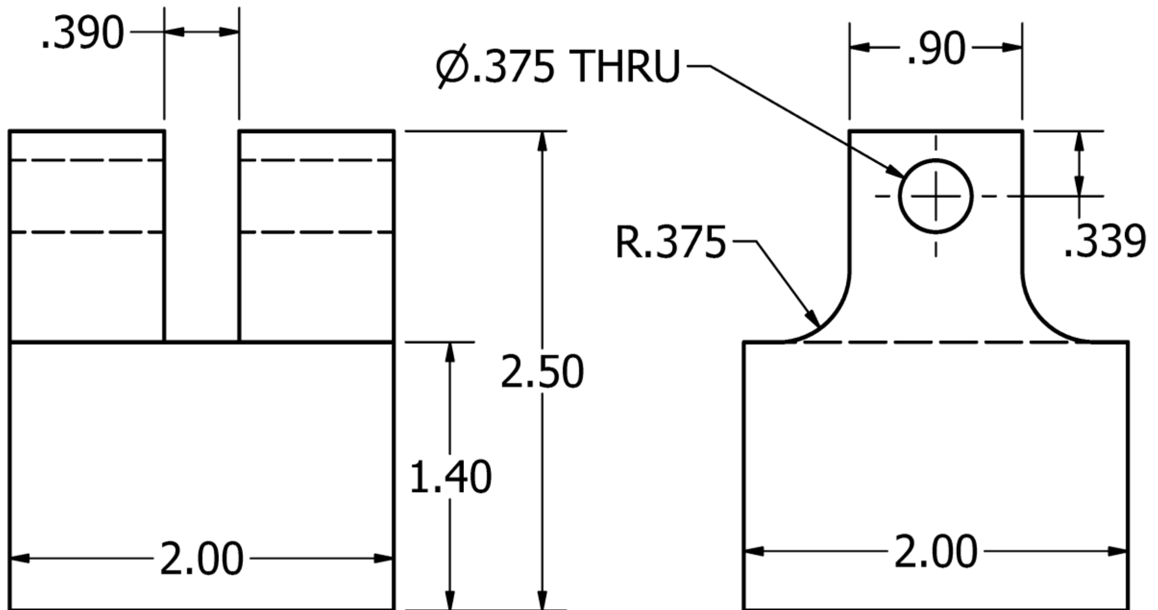


DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: Steel
SCALE: 2:1	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 15 of 24	PART NAME: PINNED GRIP	

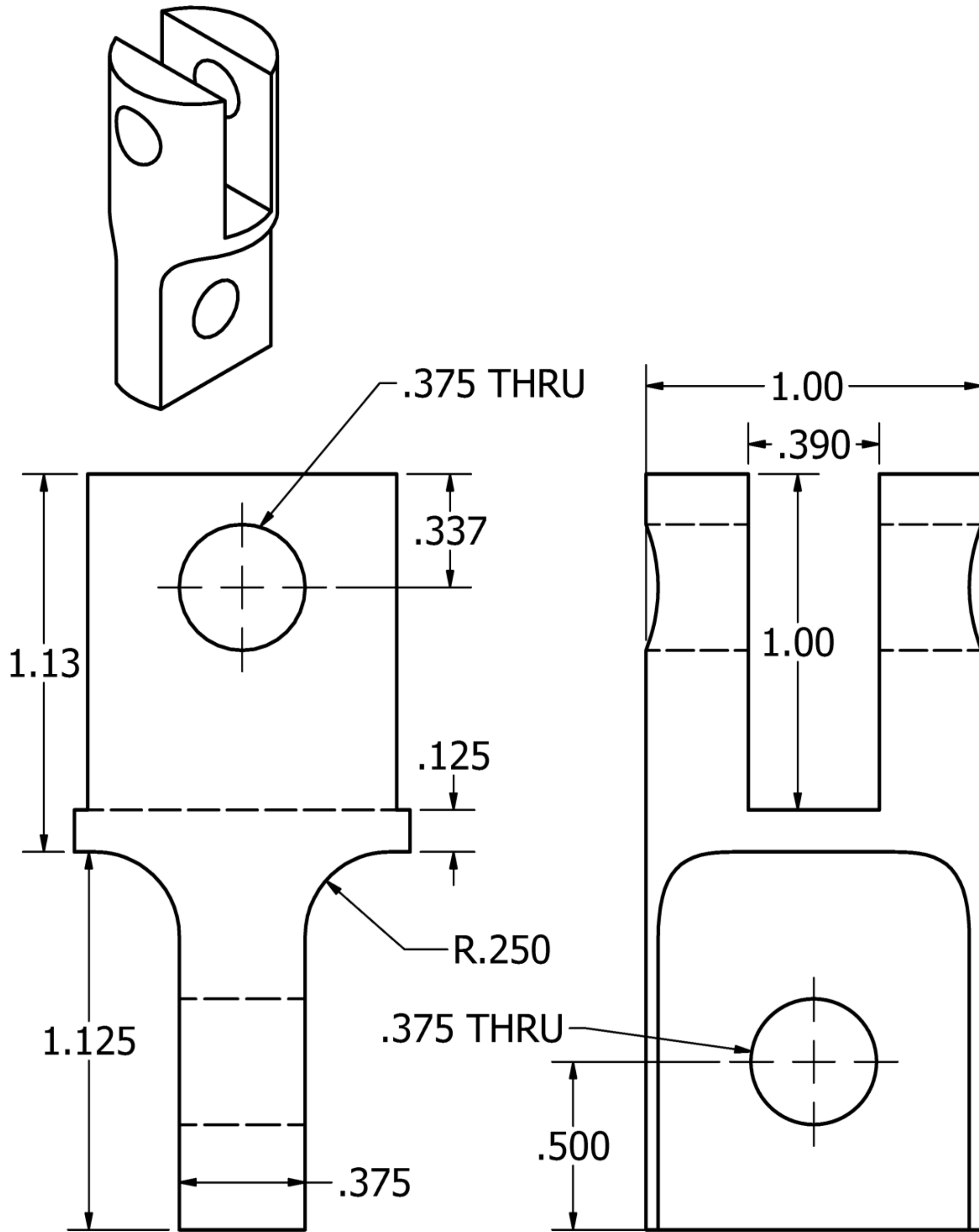


DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: MULTIPLE
SCALE: 1:1	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 16 of 24	PART NAME: THREADED CONECTION ASSEMBLY	

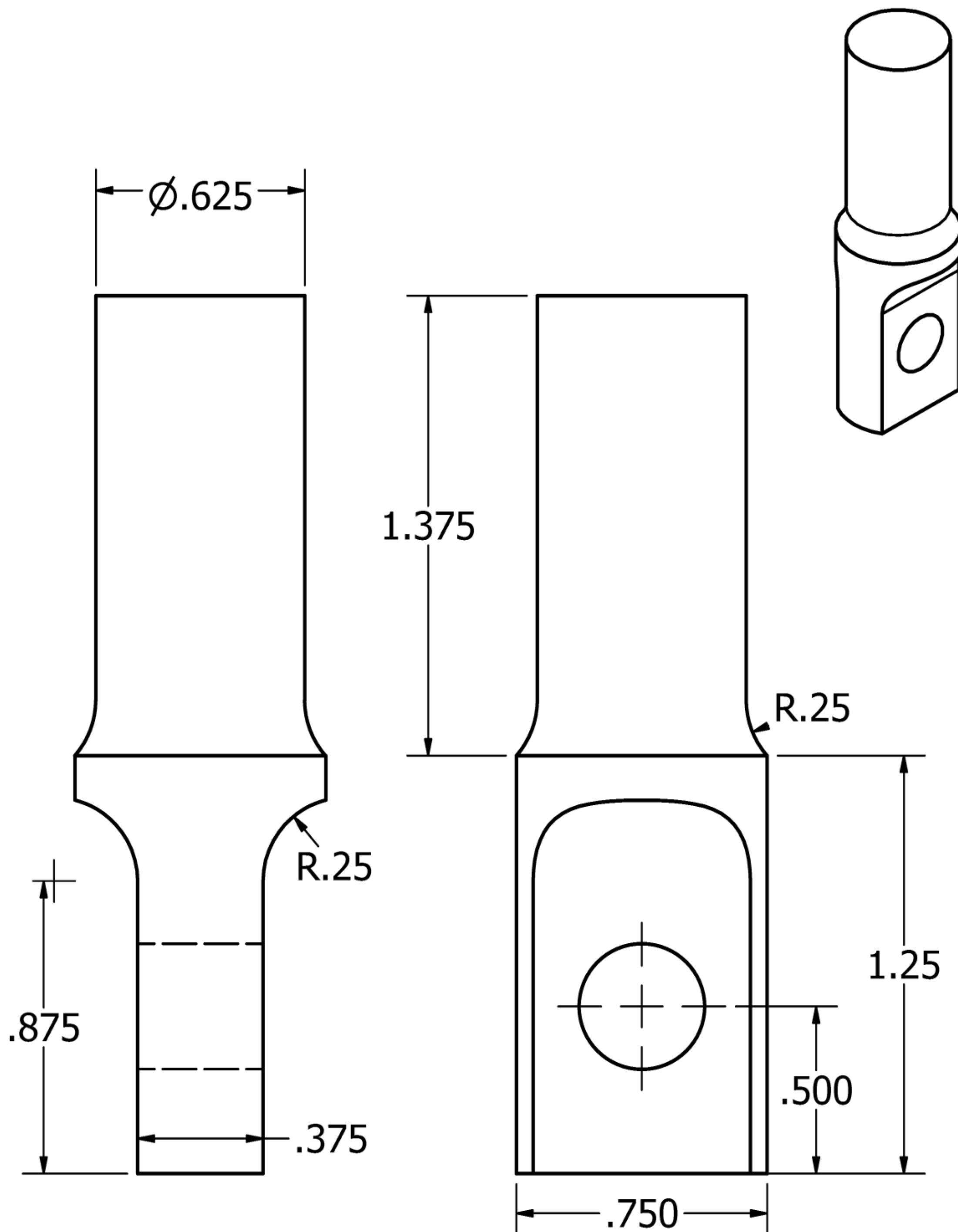




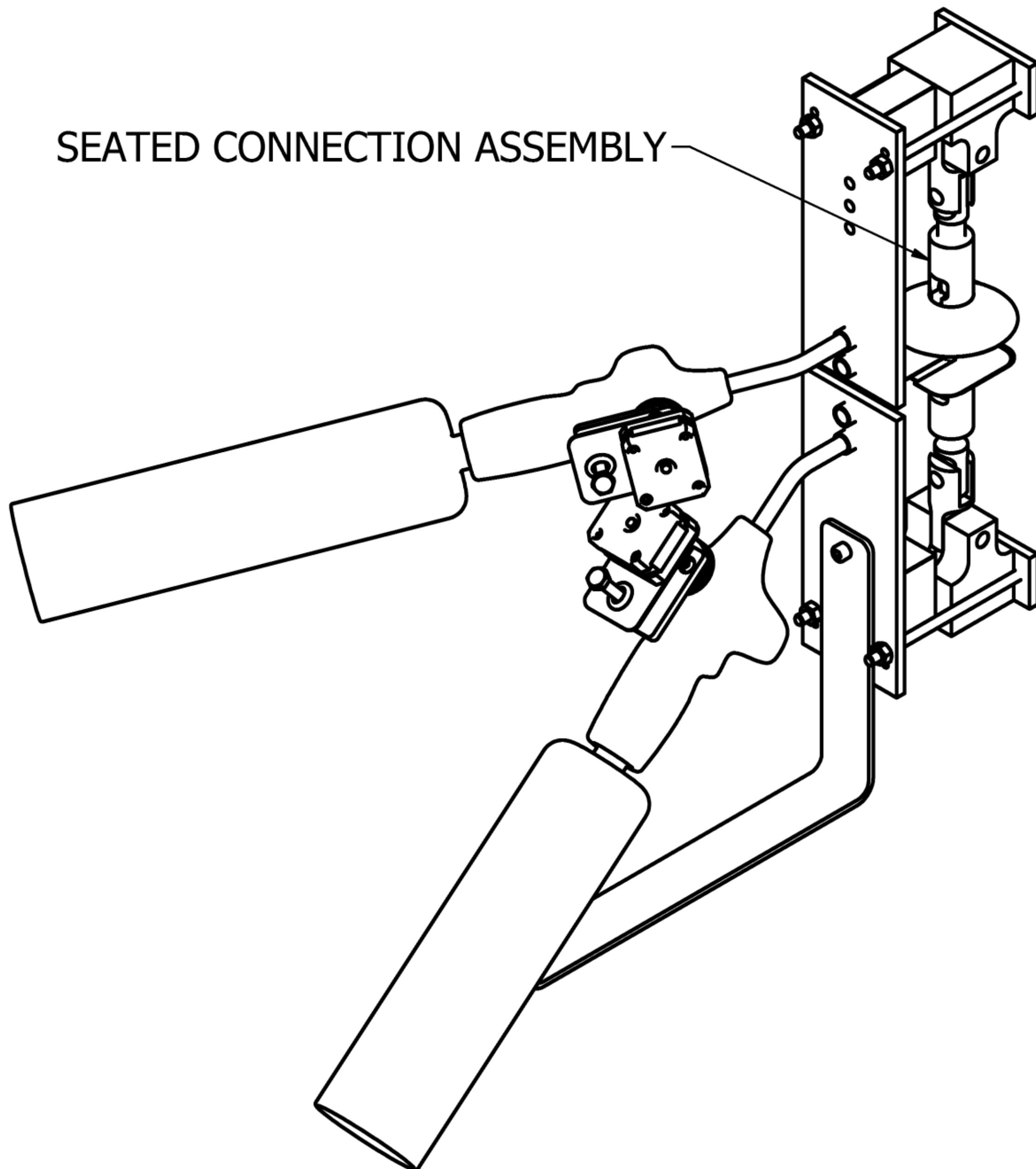
DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: Steel, Mild
SCALE: 1:1	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 17 of 24	PART NAME: BASE MOUNT	



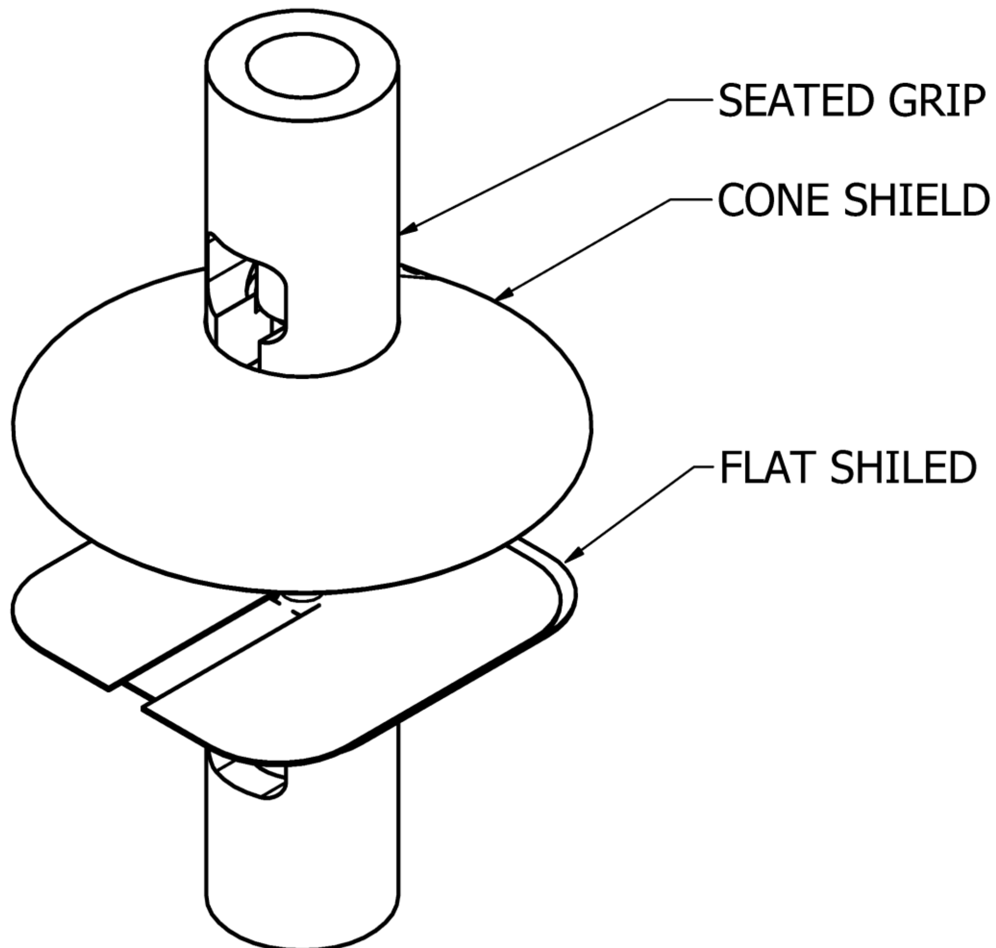
DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: Steel, Mild
SCALE: 2:1	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 18 of 24	PART NAME: GRIP AXIS 1	



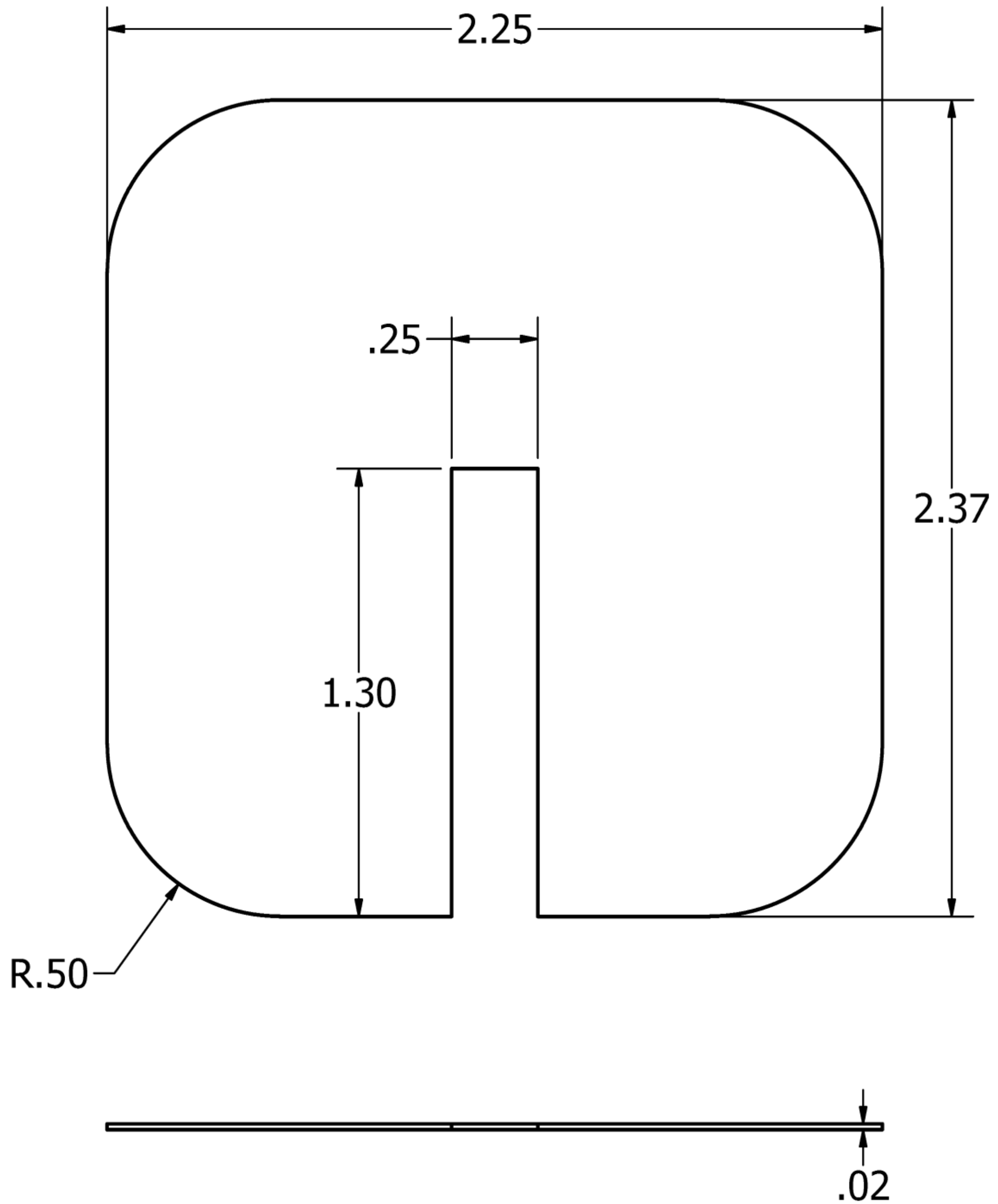
DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: Steel, Mild
SCALE: 2:1	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 19 of 24	PART NAME: GRIP AXIS 2	



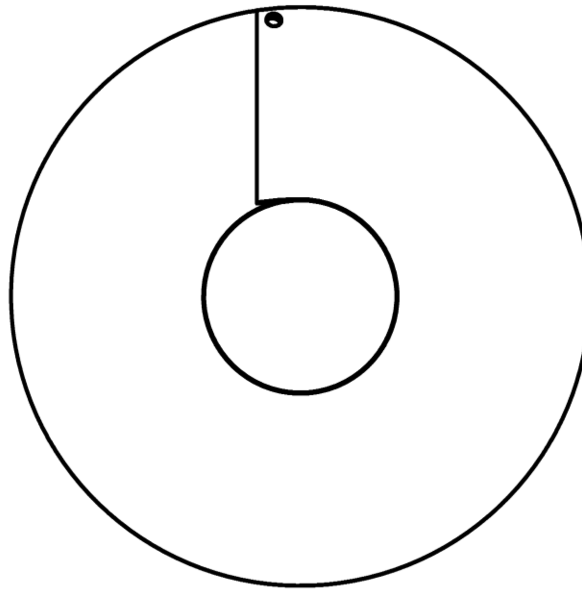
DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: MULTIPLE
SCALE: 1:4	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 20 of 24	PART NAME: SEATED PINNED ASSEMBLY	

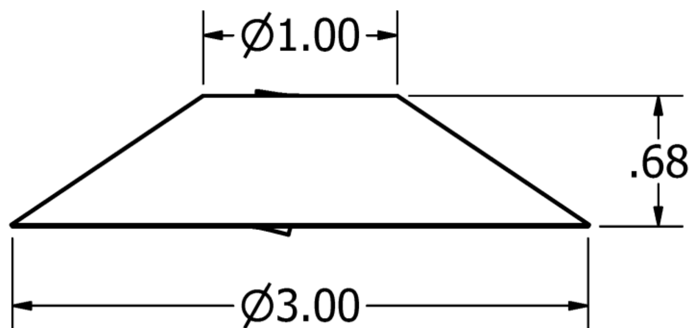


DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: MULTIPLE
SCALE: 1:1	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 21 of 24	PART NAME: SEATED CONNECTION ASSEMBLY	

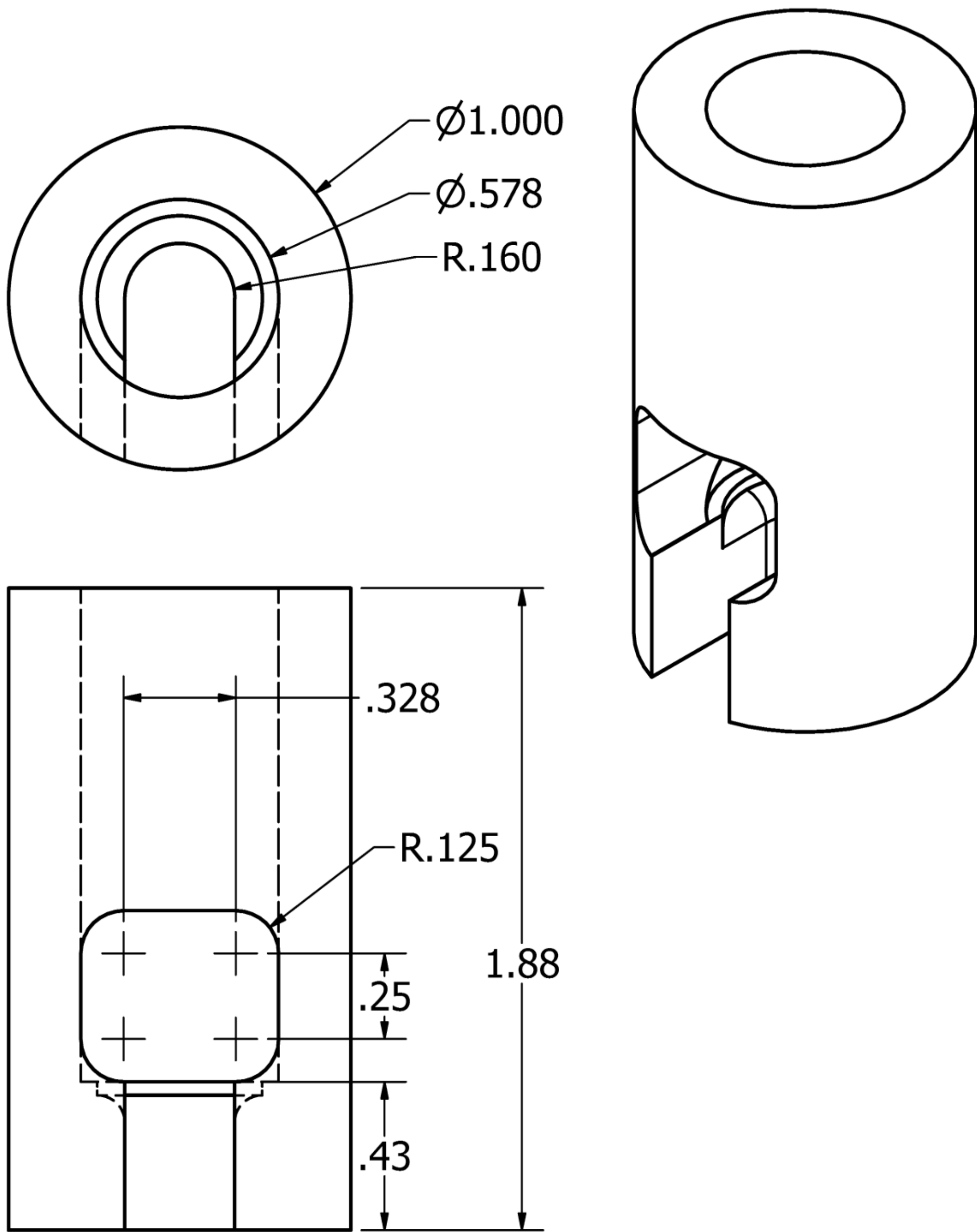


DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: Stainless Steel
SCALE: 2:1	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 22 of 24	PART NAME: FLAT SHIELD	





DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: Stainless Steel
SCALE: 1:1	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 23 of 24	PART NAME: CONE SHIELD	



DRAWN BY: V. KAMPFER	ORGANIZATION: UNIVERSITY OF IDAHO	MATERIAL: Steel, Mild
SCALE: 2:1	PROJECT: HIGH TEMPERATURE TESTING APPARATUS	
SHEET: 24 of 24	PART NAME: SEATED GRIP	

APPENDIX B: HMI SOURCE CODE

GUI_PID.py:

```
"""
Victoria Kampfer
08-31-2014

This GUI was developed to display the progress of temperatures and
motor positions for a high temperature testing apparatus. The apparatus
consists of two propane torches operated by stepper motors to heat a
metal specimen. The temperature control system is managed by a
SainSmart Uno microcontroller coupled with a motor driver shield and
two thermocouple breakout boards. The Uno runs a PID control algorithm
and communicates with this GUI to accept inputs and display outputs.
Key features of the GUI include two numerical temperature displays; two
numerical motor position displays; a real-time plot visually displaying
time, motor position, and temperature; input fields for specimen name,
temperature set-point, maximum motor positions, and PID settings; and a
timer that displays a three second countdown monitoring if the
temperature is within the +/-3 degC range of the set-point.
"""

from __future__ import division
from __future__ import print_function

import sys
import Queue
from collections import deque
import this
import time

import PyQt4.QtCore as QtCore
import PyQt4.QtGui as QtGui
import PyQt4.Qwt5 as Qwt

from com_monitor import ComMonitorThread
from eblib.serialutils import full_port_name, enumerate_serial_ports
from eblib.utils import get_all_from_queue, get_item_from_queue
from live_data_feed import LiveDataFeed


class PlottingDataMonitor(QtGui.QMainWindow):
    """
    The main class of the GUI_PID. GUI dequeues and lists are created
    to hold data, GUI components are created and set up, and the
    com_monitor/data feed are initialized. Program flags are also set.
    This class is responsible for creating and managing graphical
    entities
    of the GUI.

    :param parent: Always a top-level widget, never used
    :type parent: QWidget

    The following sub functions are part of __init__ and help
```

```
organize the
class

 - create_status_bar()
    Creates a status bar below the central widget to display text
    messages.

 - create_top_info()
    Creates and populates the top h-box, which is sub-divided into
    four group-boxes, one each for displaying information about the
    top motors and temperatures, bottom motors and temperatures,
    communication of the program, and test information such as time
    and status.

 - create_mid_plot()
    Creates the real-time plot and plot controls which are grouped
    together in the middle of the main h-box. It contains five
    curves:
    - bottom_motor_position
    - top_motor_position
    - bottom_temp,
    - top_temp,
    - top_temps_delta
    It also has a widget for turning the display of the
    top_temps_delta curve on and off.

 - create_bottom_inputs()
    Creates and populates two, stacked h-boxes at the bottom of the
    GUI. The upper h-box is the input_h_box houses entry fields for
    system input parameters. The lower h-box is the control_h_box
    and contains three push buttons:

    - start
    - stop
    - connect

"""
signalUpdateInfo = QtCore.pyqtSignal(str)

def __init__(self, parent=None):
    super(PlottingDataMonitor, self).__init__(parent)

    def create_status_bar():
        self.status_text = QtGui.QLabel('Monitor idle')
        self.statusBar().addWidget(self.status_text, 1)

    def create_top_info():
        # INFO H BOX ############################################
        # top group box
        top_info_v_box = QtGui.QVBoxLayout()
        top_info_group_box = QtGui.QGroupBox('Top Heater')
        top_info_group_box.setLayout(top_info_v_box)

        self.top_temp = CoupledBox(top_info_v_box, 'Temperature: ')

        self.top_motor_position = CoupledBox(top_info_v_box,
                                             'Motor Position: ')
```

```
self.top_heat_rate = CoupledBox(top_info_v_box,
                                'Heat Rate: ')

# bottom group box
bottom_info_v_box = QtGui.QVBoxLayout()
bottom_info_group_box = QtGui.QGroupBox('Bottom Heater')
bottom_info_group_box.setLayout(bottom_info_v_box)

self.bottom_temp = CoupledBox(bottom_info_v_box,
                              'Temperature: ')

self.bottom_motor_position = CoupledBox(bottom_info_v_box,
                                        'Motor Position: ')

self.bottom_heat_rate = CoupledBox(bottom_info_v_box,
                                   'Heat Rate: ')

# com port group box
com_info_v_box = QtGui.QVBoxLayout()
com_info_group_box = QtGui.QGroupBox('Com Port Info')
com_info_group_box.setLayout(com_info_v_box)

self.com_select = CoupledBox(com_info_v_box,
                             'Selected Port: ',
                             field_type=QtGui.QComboBox)

self.com_select.field.addItems(self.on_update_ports())
self.port_name = str(self.com_select.field.currentText())

self.com_baud = CoupledBox(com_info_v_box, 'Baud Rate: ')
self.com_cycle = CoupledBox(com_info_v_box, 'Cycle Rate: ')

# test info group box
test_info_v_box = QtGui.QVBoxLayout()
test_info_group_box = QtGui.QGroupBox('Test Info')
test_info_group_box.setLayout(test_info_v_box)

self.lcd_count_down = MyLCDCounter(3, 1000)

self.test_time_coupled_box = CoupledBox(test_info_v_box,
                                        'Test Duration: ')

self.btn_ready = QtGui.QPushButton()
self.btn_ready.setStyleSheet("background-color:#333333")

test_info_v_box.addWidget(self.lcd_count_down)
test_info_v_box.addWidget(self.btn_ready)

# info box layout
info_h_box = QtGui.QHBoxLayout()
info_h_box.setAlignment(QtCore.Qt.AlignLeft)

info_h_box.addWidget(top_info_group_box)
info_h_box.addWidget(bottom_info_group_box)
info_h_box.addWidget(com_info_group_box)
info_h_box.addWidget(test_info_group_box)
```

```
        info_h_box.setAlignment(QtCore.Qt.AlignLeft)

        self.main_v_layout.addLayout(info_h_box)

    def create_mid_plot():
        # PLOT HELPERS
        # ###############################################
        self.startMarker = Qwt.QwtPlotMarker()

        # PLOT H BOX
        # #################################################
        self.plot = RealTimePlot()
        self.plot.add_curve('bottom_motor_position',
                            self.time_display,
                            self.bottom_motor_display,
                            penStyle=QtCore.Qt.DashDotDotLine)

        self.plot.add_curve('top_motor_position',
                            self.time_display,
                            self.top_motor_display,
                            color='yellow',
                            penStyle=QtCore.Qt.DashDotDotLine)

        self.plot.add_curve('bottom_temp',
                            self.time_display,
                            self.bottom_temps_display,
                            yAxisRight=True)
        self.plot.add_curve('top_temp',
                            self.time_display,
                            self.top_temps_display,
                            color='yellow',
                            yAxisRight=True)

        self.plot.add_curve('top_temps_delta',
                            self.time_display,
                            self.top_temps_delta,
                            color='red',
                            yAxisRight=True)

        # PLOT CONTROLS
        btn_top_dt_curve = QtGui.QPushButton('Top DT Curve')

        self.connect(btn_top_dt_curve,
                     QtCore.SIGNAL('clicked()'),
                     self.plot.curves[
                         'top_temps_delta'].toggle_curves)

        # Create VBox and add controls
        plot_controls_v_box = QtGui.QVBoxLayout()
        plot_controls_v_box.addWidget(btn_top_dt_curve)

        # Add controls to groupBox
        plot_controls_group_box = QtGui.QGroupBox('Plot Controls')
        plot_controls_group_box.setLayout(plot_controls_v_box)

        plot_h_box = QtGui.QHBoxLayout()
        plot_h_box.addWidget(plot_controls_group_box)
```

```
        plot_h_box.addWidget(self.plot)

        self.main_v_layout.addLayout(plot_h_box)

    def create_bottom_inputs():
        # INPUTS ##########################################################
        input_sub_h_box = QtGui.QHBoxLayout()

        self.set_point = CoupledBox(input_sub_h_box,
                                    'Set Temp: ',
                                    field_type=QtGui.QLineEdit,
                                    default_str=str(200))

        self.top_max_position = \
            CoupledBox(input_sub_h_box,
                       'Top Max Motor '
                       'Position: ',
                       field_type=QtGui.QLineEdit,
                       default_str=str(600))

        self.bottom_max_position = \
            CoupledBox(input_sub_h_box,
                       'Bottom Max Motor Position: ',
                       field_type=QtGui.QLineEdit,
                       default_str=str(600))

        self.proportional = CoupledBox(input_sub_h_box, 'PID p: ',
                                       field_type=QtGui.QLineEdit,
                                       default_str=str(10))

        self.integral = CoupledBox(input_sub_h_box, 'PID i: ',
                                   field_type=QtGui.QLineEdit,
                                   default_str=str(2))

        self.derivative = CoupledBox(input_sub_h_box, 'PID d: ',
                                     field_type=QtGui.QLineEdit,
                                     default_str=str(0.06))

        self.btn_update_settings = \
            QtGui.QPushButton('Update\nSettings')

        self.btn_update_settings.setDisabled(True)

        self.connect(self.btn_update_settings,
                     QtCore.SIGNAL('clicked()'),
                     self.on_update_settings)

        input_group_box = QtGui.QGroupBox('Arduino Settings')
        input_group_box.setLayout(input_sub_h_box)
        input_h_box = QtGui.QHBoxLayout()

        self.specimen_name = CoupledBox(input_h_box,
                                        'Specimen Name: ',
                                        field_type=QtGui.QLineEdit,
                                        default_str='')

        self.specimen_name.field.setMinimumWidth(100)
```

```
        input_h_box.addWidget(input_group_box)
        input_sub_h_box.addWidget(self.btn_update_settings)

        # CONTROL ##################################################
        control_h_box = QtGui.QHBoxLayout()

        self.btn_start = QtGui.QPushButton('Start')
        self.btn_start.setDisabled(True)

        self.connect(self.btn_start, QtCore.SIGNAL('clicked()'),
                     self.on_start)

        self.btn_stop = QtGui.QPushButton('Stop')

        self.connect(self.btn_stop, QtCore.SIGNAL('clicked()'),
                     self.on_stop)

        self.btn_connect = QtGui.QPushButton('Connect')

        self.connect(self.btn_connect, QtCore.SIGNAL('clicked()'),
                     self.on_connect)

        control_h_box.addWidget(self.btn_start)
        control_h_box.addWidget(self.btn_stop)
        control_h_box.addWidget(self.btn_connect)

        self.main_v_layout.addLayout(input_h_box)
        self.main_v_layout.addLayout(control_h_box)

    # Create data ques / lists
    self.time_display = deque(maxlen=100)
    self.bottom_temps_display = deque(maxlen=100)
    self.bottom_motor_display = deque(maxlen=100)
    self.top_temps_display = deque(maxlen=100)
    self.top_motor_display = deque(maxlen=100)
    self.top_temps_delta = deque(maxlen=100)

    self.time = deque()
    self.bottom_temperatures = deque()
    self.bottom_motor = deque()
    self.top_temperatures = deque()
    self.top_motor = deque()

    self.arduino_inputs = list()

    # Creates GUI components
    self.main_v_layout = QtGui.QVBoxLayout()
    create_top_info()
    create_mid_plot()
    create_bottom_inputs()
    create_status_bar()

    # Sets up GUI components
    self.main_frame = QtGui.QWidget()
    self.main_frame.setLayout(self.main_v_layout)
    self.setCentralWidget(self.main_frame)
```

```
        # Set up com monitor / data feed
        self.data_from_arduino = Queue.Queue()
        self.error_que = Queue.Queue()
        self.msg_to_send = deque(maxlen=1)
        self.com_monitor_active = False

        self.com_monitor = ComMonitorThread(
            self.data_from_arduino,
            self.error_que,
            self.msg_to_send,
            full_port_name(self.port_name),
            9600)

        self.live_data_feed = LiveDataFeed()

        print(self.port_name)

        # Set program flags
        self.flag_ready = False
        self.test_started = False
        self.go_time = 0.0
        self.start_time = 0.0

    def on_update_settings(self):
        """
        Is called when the 'clicked' signal from btn_update is emitted.
        Sends program parameters to the PLC. Builds a string from the
        input_h_box entries and posts it to the deque, msg_to_send. It
        also updates plot settings and stores inputs in a list for
        future
        reference.
        """

        # Inputs stored as deque: msg_to_send
        inputs = [self.set_point.val(),
                  self.top_max_position.val(),
                  self.bottom_max_position.val(),
                  self.proportional.val(),
                  self.integral.val(),
                  self.derivative.val()]

        str_msg = str(inputs[0]) + '\t'
        str_msg += str(inputs[1]) + '\t'
        str_msg += str(inputs[2]) + '\t'
        str_msg += str(inputs[3]) + '\t'
        str_msg += str(inputs[4]) + '\t'
        str_msg += str(inputs[5])

        # Inputs added to deque
        self.msg_to_send.append(str_msg)

        # Plot title from inputs
        self.plot.plot.setTitle(str_msg)

        # Setting motor axis based on max motor position
        self.plot.set_motor_axis(
            max([self.top_max_position.val(),
```

```
                self.bottom_max_position.val()]))

        # Adding inputs to arduino list
        self.arduino_inputs = inputs

    @staticmethod
    def on_update_ports():
        """
        Uses pySerial's enumerate_serial_ports() to determine available
        serial ports. Is called when the serial port's
        combo-box is created.

        :return: List of serial port names.
        :rtype: list
        """
        ports = list(enumerate_serial_ports())
        return ports

    def on_stop(self):
        """
        Is called when the 'clicked' signal from btn_stop is emitted. A
        message to move motors to the zero positions at a speed of 75 is
        posted to msg_to_send. A file is saved reporting time,
        temperature, and motor position data for both the top and bottom
        locations with a name containing the specimen name, date, and
        time. A .png image of the GUI real-time curves plot is saved
        with the same file name as the text document.
        """
        self.msg_to_send.append('0 0 75')

        file_name = str(
            self.specimen_name.lbl_2.text() + '_' + time.strftime(
                "%d-%m-%Y") +
            '_' + time.strftime("%I-%M-%S") + '.txt')

        with open(file_name, 'w') as f:
            line = 'TIME' + \
                   '\t' + 'BOTTOM TEMP' + \
                   '\t' + 'BOTTOM MOTOR' + \
                   '\t' + 'TOP TEMP' + \
                   '\t' + 'TOP MOTOR' + '\n '

            f.write(line)
            for i1, t in enumerate(self.time):
                str_1 = self.bottom_temperatures[i1]
                str_2 = self.bottom_motor[i1]
                str_3 = self.top_temperatures[i1]
                str_4 = self.top_motor[i1]

                line = '{0}\t{1}\t{2}\t{3}\t{4}\n'.format(str(t),
                                                          str_1,
                                                          str_2,
                                                          str_3,
                                                          str_4)

                f.write(line)
```

```
        f.close()

        QtGui.QPixmap.grabWidget(self.plot.plot).save(
            file_name.rstrip('.txt') + '.png', 'PNG')

    def on_connect(self):
        """
        Is called when the 'clicked' signal from btn_connect is emitted.
        Clears previous test data, establishes communication over the
        com_monitor, and sets up the com_timer
        """

        # Clear previous test data
        self.time_display.clear()
        self.bottom_temps_display.clear()
        self.bottom_motor_display.clear()
        self.top_temps_display.clear()
        self.top_motor_display.clear()
        self.top_temps_delta.clear()
        self.bottom_temperatures.clear()
        self.bottom_motor.clear()
        self.top_temperatures.clear()
        self.top_motor.clear()
        self.time.clear()

        # Ensures com_timer is disconnected from serial to begin with
        self.com_timer = None

        # If a valid port is unavailable, exit function
        if self.port_name == '':
            return

        # Create and start the com_monitor object
        self.com_monitor = ComMonitorThread(
            self.data_from_arduino,
            self.error_que,
            self.msg_to_send,
            full_port_name(self.port_name),
            9600)
        self.com_monitor.start()

        # Checks for com monitor error
        com_error = get_item_from_queue(self.error_que)
        if com_error is not None:

            QtGui.QMessageBox.critical(self,
                                       'ComMonitorThread error',
                                       com_error)

            self.com_monitor = None

        # Creates, connects, and starts com timer. Separate from
        # com_monitor's thread.
        self.portTimer = time.clock()
        self.com_timer = QtCore.QTimer()

        self.connect(self.com_timer, QtCore.SIGNAL('timeout()'),
```

```
                self.on_timer)

        self.com_timer.start(1000.0 / 100)

    def on_start(self):
        """
        Is called when the 'clicked' signal from btn_start is emitted.
        Posts start trigger 's' to msg_to_send. If the specimen name
        entry field is empty the user is notified and the start trigger
        is not posted.
        """

        if len(self.specimen_name.lbl_2.text()) > 0:
            self.msg_to_send.append('s')
        else:
            QtGui.QMessageBox.critical(self, 'Invalid Specimen Name',
                                       'Please Enter Specimen Name and '
                                       'Press Start')

    def read_serial_data(self):
        """
        Function called periodically by the update timer to read data
        from the serial port.

        Check to see if data is a message or command. Messages will be
        displayed by the status bar.

        If data is a command, determine type and action to be taken.
         - *Parameters* indicates the PLC is ready to accept input
           parameters, disables the connect button and enables the
           update settings button in the GUI.

         - *Ready* indicates the PLC is ready to begin test, disables
           the update settings button and enables the start button.

         - *Ignited* indicates the PLC has witnessed a 3degC change in
           temperature, and that the bottom heating element is working.
           test_started is set to True to trigger updating the plot and
           info. Plotting parameters are started.

         - *Stopped* initiates the closing of the com monitor, and the
           start button is disabled while the connect button is enabled.
           test_started is set to False.

         If neither MSG or CMD is issued, continue plotting time and
         data over the live_data_feed.
        """
        q_data = list(get_all_from_queue(self.data_from_arduino))
        if len(q_data) > 0:
            _time = q_data[-1][1]
            _data = q_data[-1][0].split()
            if _data[0].startswith("MSG:"):
                msg = q_data[-1][0]
                self.status_text.setText(msg)

            elif _data[0].startswith("CMD:"):
                cmd = _data[1]
```

```
            if cmd == 'Parameters':
                self.btn_connect.setDisabled(True)
                self.btn_update_settings.setDisabled(False)

            elif cmd == 'Ready':
                self.btn_update_settings.setDisabled(True)
                self.btn_start.setDisabled(False)

            elif cmd == 'Started':
                self.btn_start.setDisabled(True)
                self.btn_connect.setDisabled(True)

            elif cmd == 'Ignited':
                self.test_started = True
                self.start_time = _time
                self.startMarker.setXValue(_time)
                self.startMarker.setYValue(0)
                self.startMarker.setLineStyle(
                    Qwt.QwtPlotMarker.HLine)

                self.startMarker.attach(self.plot.plot)

            elif cmd == 'Stopped':
                self.com_monitor.close()
                self.com_monitor = None
                self.btn_start.setDisabled(True)
                self.btn_connect.setDisabled(False)
                self.test_started = False

        else:
            self.live_data_feed.add_data((_time, _data))

    def on_timer(self):
        """
        Is called when the timeout signal from the com_timer is emitted.
        Serial data is read and the plot is updated.
        """
        self.read_serial_data()
        self.update_info_and_plot()

    def update_info_and_plot(self):
        """
        It is repeatedly called from on_timer and updates test
        information in the info_h_box, and plot.
        """

        if self.live_data_feed.has_new_data:
            _time, _data = self.live_data_feed.read_data()
            self.time_display.append(_time)
            self.bottom_motor_display.append(float(_data[0]))
            self.bottom_temps_display.append(float(_data[1]))
            self.top_motor_display.append(float(_data[2]))
            self.top_temps_display.append(float(_data[3]))

            # Determines if parameters have been sent to the PLC
            if len(self.arduino_inputs) > 0:
```

```
            self.top_temps_delta.append(
                self.arduino_inputs[0] - self.top_temps_display[-1])

        # On each test_start trigger, append new data to dequeues
        if self.test_started:
            _testDuration = _time - self.start_time

            self.test_time_coupled_box.lbl_2.setText(
                "%.1f" % _testDuration)

            self.time.append(_time - self.start_time)
            self.bottom_motor.append(float(_data[0]))
            self.bottom_temperatures.append(float(_data[1]))
            self.top_motor.append(float(_data[2]))
            self.top_temperatures.append(float(_data[3]))

            # Starts countdown timer to determine 3 sec wait period
            # if temperature is within +/-3degC of set-point
            if abs(self.bottom_temps_display[
                -1] - self.set_point.val()) < 3 and abs(
                        self.top_temps_display[
                            -1] - self.set_point.val()) < 3:

                if not self.flag_ready:
                    self.lcd_count_down.start()

                    self.btn_ready.setStyleSheet(
                        "background-color:yellow")

                    self.btn_ready.setText('GO!')
                    self.flag_ready = True
                    self.go_time = _time

                elif self.flag_ready and _time > self.go_time + 3:
                    self.btn_ready.setText('SUCCESS')

                    self.btn_ready.setStyleSheet(
                        "background-color:green")

            else:
                self.btn_ready.setStyleSheet("background-color:red")
                self.flag_ready = False
                self.go_time = 0.0
                self.btn_ready.setText('')

        self.plot.update_plot()

        # Updates info_h_box. Starts after 9 data points to
        # provide a

        # more accurate avg for the heat up rate
        if len(self.time_display) > 9:
            cycle_rate = \
                (self.time_display[-1] - self.time_display[-2])

            self.bottom_motor_position.lbl_2.setText(_data[0])
            self.bottom_temp.lbl_2.setText(_data[1])
```

```
                dt1_dt = \
                    (self.bottom_temps_display[-1] -
                     self.bottom_temps_display[-10]) / \
                    (self.time_display[-1] - self.time_display[-10])

                self.bottom_heat_rate.lbl_2.setText("%.4f" % dt1_dt)

                self.top_motor_position.lbl_2.setText(_data[2])
                self.top_temp.lbl_2.setText(_data[3])

                dt2_dt = (self.top_temps_display[-1] -
                          self.top_temps_display[-10]) / \
                         (self.time_display[-1] -
                          self.time_display[-10])

                self.top_heat_rate.lbl_2.setText("%.4f" % dt2_dt)

                self.com_cycle.lbl_2.setText("%.4f" % cycle_rate)


class CoupledBox(QtGui.QWidget):
    """
    Class to combine a label and a second object known as field. field
    may be a QLabel, QLineEdit, or QComboBox

    :param parent: The QWidget that will house the new coupled box
    :type parent: QWidget
    :param label: A string to be displayed as a label
    :type label: str
    :param default_str: The default string to display in the coupled box
    :type default_str: str
    :param field_type: Specifies the type of the combo box
    :type field_type: QLabel, QComboBox, QLineEdit
    :return: Nothing
    """

    def __init__(self, parent, label, default_str='NA',
                 field_type=QtGui.QLabel):

        super(CoupledBox, self).__init__()

        assert isinstance(default_str,
                          str), "Make sure CoupledBox default_str is " \
                                "a sting and not an int!!!"

        _label = QtGui.QLabel(label)
        _label.setAlignment(QtCore.Qt.AlignLeft)
        if field_type == QtGui.QLabel:
            self.field = QtGui.QLabel(default_str)
            self.field.setFixedWidth(40)
            self.field.setAlignment(QtCore.Qt.AlignRight)
        elif field_type == QtGui.QLineEdit:
            self.field = QtGui.QLineEdit()
            self.field.setText(default_str)
        elif field_type == QtGui.QComboBox:
            self.field = QtGui.QComboBox()
```

```
        h_box = QtGui.QHBoxLayout()
        h_box.addWidget(_label)
        h_box.addWidget(self.field)
        self.setLayout(h_box)

        parent.addWidget(self)

    def val(self):
        """
        Returns value of field as a float.

        :return: Value of field
        :rtype: float
        """
        return float(self.field.text())

    def set_val(self, value):
        """
        Sets the value of field as a str.

        :param value: Value to be set
        :type value: str, float, int
        """
        self.field.setText(str(value))


class RealTimePlot(QtGui.QWidget):
    """
    Class to plot real time data.

    :var plot: a Qwt.QwtPlot
    :var curves: A dictionary that contains the curves to be plotted
        onto plot

    """

    def __init__(self):
        super(RealTimePlot, self).__init__()
        self.plot = self.create_plot()

        self.curves = {}

        layout = QtGui.QHBoxLayout()
        layout.addWidget(self.plot)
        self.setLayout(layout)

    def create_plot(self):
        """
        Creates plot titles, axes, labels, etc...

        :return: plot
        :rtype plot: Qwt.QwtPlot
        """
        plot = Qwt.QwtPlot(Qwt.QwtText('title'), self)
        plot.setCanvasBackground(QtCore.Qt.black)
        plot.setAxisTitle(Qwt.QwtPlot.xBottom, 'Time')
```

```
        plot.setAxisScale(Qwt.QwtPlot.xBottom, 0, 10, 1)
        plot.setAxisTitle(Qwt.QwtPlot.yLeft, 'Motor')
        plot.setAxisScale(Qwt.QwtPlot.yLeft, 0, 700)
        plot.enableAxis(Qwt.QwtPlot.yRight)
        plot.setAxisTitle(Qwt.QwtPlot.yRight, 'Temperature')
        plot.setAxisScale(Qwt.QwtPlot.yRight, 0, 500)
        legend = Qwt.QwtLegend(plot)
        plot.insertLegend(legend, Qwt.QwtPlot.TopLegend)
        plot.replot()

        return plot

    def set_motor_axis(self, max_position):
        """
        Sets motor position axis limits with respect to user inputs.

        :param max_position: Max motor position as entered in the GUI
            inputs
        :type max_position: float
        """
        self.plot.setAxisScale(Qwt.QwtPlot.yLeft, 0, max_position)
        print("max position:", max_position)

    def add_curve(self, name, x_data, y_data, color='limegreen',
                  yAxisRight=False, penStyle=QtCore.Qt.SolidLine):
        """
        Adds curves to plot.

        :param name: name of curve
        :type name: str
        :param x_data: x values for curve plot (time)
        :type x_data: deque
        :param y_data: y values for curve plot (motor position or
            temperature)
        :type y_data: deque
        :param color: color changes with motor and temperature position.
        :type color: str
        :param yAxisRight: sets y-axis to the right of the plot
        :type yAxisRight: bool
        :param penStyle: sets curve line styles
        :type penStyle: QtCore.Qt.SolidLine
        """
        curve = RealTimeCurve(name, x_data, y_data, color, yAxisRight)
        curve.attach(self.plot)
        self.curves[name] = curve

    def update_plot(self):
        """
        Updates plot's curves and reset x-axis limits.
        """
        for name, curve in self.curves.iteritems():
            curve.update_curve()

        x_data = list(self.curves.values()[0].x_data)

        self.plot.setAxisScale(Qwt.QwtPlot.xBottom, x_data[0],
                               max(20, x_data[-1]))
```

```
        self.plot.replot()


class RealTimeCurve(Qwt.QwtPlotCurve):
    """
    Class to define curve as used by the RealTimePlot.

    :param name: name of curve
    :type name: str
    :param x_data: x values for curve plot (time)
    :type x_data: deque
    :param y_data: y values for curve plot (motor position or
        temperature)
    :type y_data: deque
    :param color: color changes with motor and temperature position.
    :type color: str
    :param yAxisRight: sets y-axis to the right of the plot
    :type yAxisRight: bool
    :param penStyle: sets curve line styles
    :type penStyle: QtCore.Qt.SolidLine
    """

    def __init__(self, name, x_data, y_data, color='limegreen',
                 yAxisRight=False, penStyle=QtCore.Qt.SolidLine):

        Qwt.QwtPlotCurve.__init__(self)
        self.x_data = x_data
        self.y_data = y_data

        self.setRenderHint(Qwt.QwtPlotItem.RenderAntialiased)
        pen = QtGui.QPen(QtGui.QColor(color), penStyle)
        pen.setWidth(2)
        self.setPen(pen)
        self.setData(list(x_data), list(y_data))

        self.setTitle(name)

        if yAxisRight:
            self.setAxis(Qwt.QwtPlot.yRight, True)

    def update_curve(self):
        """
        Updates x and y curve data. Call this function when new data is
        available and plot should be updated.
        """
        self.setData(list(self.x_data), list(self.y_data))

    def toggle_curves(self):
        """
        Hides or un-hides selected plot curves.
        """
        if self.isVisible():
            self.hide()
            print('Hidden')
        else:
            self.show()
            print('Shown')
```

```
class MyLCDCounter(QtGui.QLCDNumber):
    """
    Class to create a countdown timer based off of QtGui.QLCDNumber. It
    incorporates the QtCore.QTimer to automatically
    update the QtGui.QLCDNumber.

    :param start_time: Starting time of LCD countdown
    :type start_time: int, float
    :param interval: Time decremented from start_time value
    :type interval: int, float
    """

    def __init__(self, start_time, interval):
        QtGui.QLCDNumber.__init__(self)
        self.interval = interval
        self.value = start_time
        self.timer = QtCore.QTimer()

        self.connect(self.timer, QtCore.SIGNAL("timeout()"), self,
                     QtCore.SLOT("update()"))

    @QtCore.pyqtSlot()
    def update(self):
        """
        Updates the countdown timer display values. This function is
        automatically called by the QtCore.QtTimer.
        """
        if self.value == 0:
            self.display(self.value)
            self.stop()
        else:
            self.display(self.value)
            self.value -= self.interval / 1000

    def start(self):
        """
        Initiates countdown. Call this function to start.
        """
        self.timer.start(self.interval)

    def stop(self):
        """
        Stops countdown timer. Automatically called when countdown
        reaches
        zero.
        """
        self.timer.stop()


def main():
    """
    Main Function of GUI  script
    """
    app = QtGui.QApplication(sys.argv)
    form = PlottingDataMonitor()
```

```
    form.setFixedSize(1335, 890)
    form.show()
    # form.showMaximized()
    app.exec_()

# Main script:
if __name__ == "__main__":
    main()
```

live_data_feed.py:

```
class LiveDataFeed(object):
    """
    Class to house the latest data. It allows the user to post or read
    data. It keeps track of if the current data has
    ever been read.

    :var cur_data: The newest data
    :type cur_data: tuple
    """

    def __init__(self):
        self.cur_data = None
        self.has_new_data = False

    def add_data(self, data):
        """
        Adds new data to the object.

        :param data: The data
        :type data: tuple
        """
        self.cur_data = data
        self.has_new_data = True

    def read_data(self):
        """
        Returns data.

        :return cur_data: Returns most recent data
        ":rtype cur_data: tuple
        """
        self.has_new_data = False
        return self.cur_data


if __name__ == "__main__":
    pass
```

com_monitor.py:

```
from __future__ import print_function
import threading
import time
```

```
import serial


class ComMonitorThread(threading.Thread):
    """ A thread for monitoring a COM port. The COM port is
        opened when the thread is started.

        data_q:
            Queue for received data. Items in the queue are
            (data, timestamp) pairs, where data is a binary
            string representing the received data, and timestamp
            is the time_display elapsed from the thread's start (in
            seconds).

        error_q:
            Queue for error messages. In particular, if the
            serial port fails to open for some reason, an error
            is placed into this queue.

        port_num:
            The COM port to open. Must be recognized by the
            system.

        port_baud/stopbits/parity:
            Serial communication parameters

        port_timeout:
            The timeout used for reading the COM port. If this
            value is low, the thread will return data in finer
            grained chunks, with more accurate timestamps, but
            it will also consume more CPU.
    """
    def __init__(   self,
                    data_q, error_q, msg2Send,
                    port_num,
                    port_baud,
                    port_stopbits=serial.STOPBITS_ONE,
                    port_parity=serial.PARITY_NONE,
                    port_timeout=0.01):

        threading.Thread.__init__(self)

        self.serial_port = None
        self.serial_arg = dict( port=port_num,
                                baudrate=port_baud,
                                stopbits=port_stopbits,
                                parity=port_parity,
                                timeout=port_timeout)

        self.data_q = data_q
        self.error_q = error_q
        self.msg2Send = msg2Send

        self.alive = threading.Event()
        self.alive.set()

    def run(self):
```

```
        try:
            if self.serial_port:
                self.serial_port.close()
            self.serial_port = serial.Serial(**self.serial_arg)
        except serial.SerialException, e:
            self.error_q.put(e.message)
            return

        # Restart the clock
        time.clock()

        while self.alive.isSet():
            # Reading 1 byte, followed by whatever is left in the
            # read buffer, as suggested by the developer of
            # PySerial.
            #
            #print(self.msg_to_send)
            if len(self.msg2Send) > 0:
                self.serial_port.write(self.msg2Send[-1])
                self.msg2Send.pop()

            data = self.serial_port.readline()

            if len(data) > 0:
                print(data)
                timestamp = time.clock()
                self.data_q.put((data, timestamp))

        # clean up
        if self.serial_port:
            self.serial_port.close()

    def join(self, timeout=None):
        self.alive.clear()
        threading.Thread.join(self, timeout)

    def close(self):
        self.alive.clear()
```

serialutils.py:

```
"""
Some serial port utilities for Windows and PySerial

Eli Bendersky (eliben@gmail.com)
License: this code is in the public domain
"""
import re, itertools
import _winreg as winreg


def full_port_name(portname):
    """
    Given a port-name (of the form COM7, COM12, CNCA0, etc.) returns a
    full name suitable for opening with the Serial class.
    """
```

```
    m = re.match('^COM(\d+)$', portname)
    if m and int(m.group(1)) < 10:
        return portname
    return '\\\\.\\' + portname


def enumerate_serial_ports():
    """
    Uses the Win32 registry to return an iterator of serial (COM) ports
    existing on this computer.
    """
    path = 'HARDWARE\\DEVICEMAP\\SERIALCOMM'
    try:
        key = winreg.OpenKey(winreg.HKEY_LOCAL_MACHINE, path)
    except WindowsError:
        raise StopIteration

    for i in itertools.count():
        try:
            val = winreg.EnumValue(key, i)
            yield str(val[1])
        except EnvironmentError:
            break


if __name__ == "__main__":
    import serial
    for p in enumerate_serial_ports():
        print p, full_port_name(p)
```

utils.py:

```
import random, time
import Queue


class Timer(object):
    def __init__(self, name=None):
        self.name = name

    def __enter__(self):
        self.tstart = time.time()

    def __exit__(self, type, value, traceback):
        if self.name:
            print '[%s]' % self.name,
        print 'Elapsed: %s' % (time.time() - self.tstart)


def get_all_from_queue(Q):
    """
    Generator to yield one after the others all items currently in the
    queue Q, without any waiting.
    """
    try:
        while True:
```

```
            yield Q.get_nowait()
    except Queue.Empty:
        raise StopIteration


def get_item_from_queue(Q, timeout=0.01):
    """
    Attempts to retrieve an item from the queue Q. If Q is empty, None
    is returned. Blocks for 'timeout' seconds in case the queue is
    empty, so don't use this method for speedy retrieval of multiple
    items (use get_all_from_queue for that).
    """
    try:
        item = Q.get(True, 0.01)
    except Queue.Empty:
        return None

    return item


def flatten(iterables):
    """
    Flatten an iterable of iterables. Returns a generator.
    list(flatten([[2, 3], [5, 6]])) => [2, 3, 5, 6]
    """
    return (elem for iterable in iterables for elem in iterable)


def argmin_list(seq, func):
    """
    Return a list of elements of seq[i] with the lowest func(seq[i])
    scores.

    argmin_list(['one', 'to', 'three', 'or'], len)
    ['to', 'or']
    """
    best_score, best = func(seq[0]), []
    for x in seq:
        x_score = func(x)
        if x_score < best_score:
            best, best_score = [x], x_score
        elif x_score == best_score:
            best.append(x)
    return best


def argmin_random_tie(seq, func):
    """
    Return an element with lowest func(seq[i]) score; break ties at
    random.
    """
    return random.choice(argmin_list(seq, func))


def argmin(seq, func):
    """
    Return an element with lowest func(seq[i]) score; tie goes to first
```

```
    one.
    argmin(['one', 'to', 'three'], len)
    'to'
    """
    return min(seq, key=func)


def argmax_list(seq, func):
    """ Return a list of elements of seq[i] with the highest
        func(seq[i]) scores.
        >>> argmax_list(['one', 'three', 'seven'], len)
        ['three', 'seven']
    """
    return argmin_list(seq, lambda x: -func(x))


def argmax_random_tie(seq, func):
    """ Return an element with highest func(seq[i]) score; break
        ties at random.
    """
    return random.choice(argmax_list(seq, func))


def argmax(seq, func):
    """ Return an element with highest func(seq[i]) score; tie
        goes to first one.
        >>> argmax(['one', 'to', 'three'], len)
        'three'
    """
    return max(seq, key=func)


# -----------------------------------------------------------------
if __name__ == "__main__":
    #~ print list(flatten([[1, 2], (4, 5), [5], [6, 6, 8]]))
    #~ print argmin_random_tie(['one', 'to', 'three', 'or'], len)

    print min(['one', 'to', 'three', 'or'], key=len)
    print argmin(['one', 'to', 'three', 'or'], len)
```

APPENDIX C: PLC SOURCE CODE

PID_Motor_control.ino:

```
/* Victoria Kampfer
03-18-2014

PID responsive motor positioning program:

The goal of this program is to run a temperature control system with a
heat source from two propane torches. These torches are operated by
stepper motors whose position is kept track of on a global coordinate
system with lower and upper bounds defined by the mechanical limits of the
propane torch dials, and stall position of the motors. A PID algorithm
controls movement of the motors based off of inputs from two thermocouples
and user inputs such as temperature set point and PID values.
*/

//INCLUDED LIBRARIES:
#include <AFMotor.h>
#include <EEPROM.h>
#include "Adafruit_MAX31855.h"
#include <PID_v1.h>

//PIN ASSIGNMENTS:
#define thermoDO A0
#define thermoCLK A1
#define thermo1CS A2
#define thermo2CS A3
#define thermo1VIN A4
#define thermo2VIN A5

//EEPROM VARIABLES:
//Top motor position, part a
int TMP_a = EEPROM.read(0);
//Top motor position, part b
int TMP_b = EEPROM.read(1);
//Bottom motor position, part a
int BMP_a = EEPROM.read(2);
//Bottom motor position, part a
int BMP_b = EEPROM.read(3);
//Power error flag
int flag_PWR_ERROR = EEPROM.read(4);

//VARIABLES:
int topMotorPosition = TMP_a*256 + TMP_b;
int bottomMotorPosition = BMP_a*256 + BMP_b;
int bottomSteps = 0;
int topSteps = 0;
int bottomDesired = 0;
int topDesired = 0;
int speedVal = 50;
int newSpeedVal = speedVal;
int oldBottomDesired = bottomMotorPosition;
int oldTopDesired = topMotorPosition;
```

```
unsigned long thermoReadTime;

uint8_t bottomDir;
uint8_t topDir;
uint32_t usperstep;

double temps[2];
double desiredTemp;
double bottomOutput;
double topOutput;

float bottomMaxPosition;
float topMaxPosition;

//Starting PID values acting as place holders, will be reset by GUI
//inputs
float pid_P = 100.1;
float pid_I = 10.1;
float pid_D = 10.1;

//CONSTANT VARIABLES:
//Start position of motor
const int startPosition = 0;
//Steps per revolution
const int stepsPerRev = 200;
//Upper bound for torch valve (mechanical limit)
const float maxTurns = 6;
//Pre-defined upper limits of motor position
const int endPosition = maxTurns*stepsPerRev;
//Pre-defined lower limits of bottom motor position
const int bottomMinPosition = 203;
//Pre-defined lower limits of top motor position
const int topMinPosition = 204;

//OBJECTS:
AF_Stepper bottomMotor(stepsPerRev, 1);
AF_Stepper topMotor(stepsPerRev, 2);

Adafruit_MAX31855 thermo1(thermoCLK, thermo1CS, thermoDO);
Adafruit_MAX31855 thermo2(thermoCLK, thermo2CS, thermoDO);

PID bottomPID(&temps[0], &bottomOutput, &desiredTemp, pid_P, \
pid_I, pid_D, DIRECT);

PID topPID(&temps[1], &topOutput, &desiredTemp, pid_P, pid_I, \
pid_D, DIRECT);

void setup() {
  //Opens serial port, sets data rate to 9600
  Serial.begin(9600);
  //Sets bottom thermocouple pin to low (powered off)
  pinMode(thermo1VIN, OUTPUT); digitalWrite(thermo1VIN, LOW);
  //Sets top thermocouple pin to low (powered off)
  pinMode(thermo2VIN, OUTPUT); digitalWrite(thermo2VIN, LOW);
  //Message for HMI to display in status bar
  Serial.println("MSG: Serial Connected, Enter Settings");
```

```
  Serial.println("CMD: Parameters");
  //Check power error flag for unclean shutdown indication
  if (flag_PWR_ERROR == 1){
    Serial.print("MSG: ERROR: Unclean Shutdown..."
   " please reset motor positions and EEPROM");

    while(1){
    }
  }

  //Initiate motors at starting positions [0,0]
  moveMotors(0);
  //Reads input parameters from HMI
  receiveParameters();

  //Update PID loops...
  //PID parameters
  bottomPID.SetTunings(pid_P, pid_I, pid_D);
  //PID mode
  bottomPID.SetMode(AUTOMATIC);
  //Motor limits
  bottomPID.SetOutputLimits(bottomMinPosition, (int)bottomMaxPosition);
  //1 millisec sample time
  bottomPID.SetSampleTime(1);

  topPID.SetTunings(pid_P, pid_I, pid_D);
  topPID.SetMode(AUTOMATIC);
  topPID.SetOutputLimits(topMinPosition, (int)topMaxPosition);
  topPID.SetSampleTime(1);

  //Delay between each motor movement
  usperstep = bottomMotor.setSpeed(speedVal);
  usperstep = topMotor.setSpeed(speedVal);

  //Release motors to avoid overheating
  bottomMotor.release();
  topMotor.release();

  //Begin start process for lighting propane torches
  lightFires();

}

//Monitor start time
unsigned long controlStart = millis();

void loop() {
  printStatus();

  //Prints motor positions and temps to serial port
  //Compute PID loop for bottom heater system
  bottomPID.Compute();
  //Compute PID loop for top heater system
  topPID.Compute();

  //Store bottom system PID outputs
  bottomDesired = (int)bottomOutput;
```

```
  //Store top system PID outputs
  topDesired = (int)topOutput;

  //Move motors to new positions
  moveMotors(0);
  //Check serial port for data from HMI
  parseSerial();

}




//Get starting test parameters from HMI
void receiveParameters() {
  //If serial is not available check for motor positions
  //and temperatures
  while(!Serial.available()){
    printStatus();
  }
  //Else read in testing parameters from the HMI
  desiredTemp = Serial.parseFloat();
  topMaxPosition = Serial.parseFloat();
  bottomMaxPosition = Serial.parseFloat();
  pid_P = float(Serial.parseFloat());
  pid_I = float(Serial.parseFloat());
  pid_D = float(Serial.parseFloat());
}


//Begin start process for lighting propane torches
void lightFires(){
  //Message sent to HMI to be displayed to user
  Serial.println("MSG: Press Start to Begin Test");
  //Indicates arduino is ready for test start
  Serial.println("CMD: Ready");
  //If serial is not availabe keep checking for data
  while(!Serial.available()){
    printStatus();
  }
  //Set exit flag condition
  bool exitFlag = false;
  //Enter while loop to get serial data
  while(exitFlag == false){
    //Print motor positions and temperatures to serial port
    printStatus();
    //If available serial is greater than zero check for start "trigger"
from HMI
    if(Serial.available()>0){
      //Start trigger from HMI
      if(Serial.read() == 's'){
        //Trigger exit flag condition
        exitFlag = true;
      }
    }
  }
  //Indicates testing has started
```

```
  Serial.println("CMD: Started");
  //Ignition point bottom motor position
  bottomDesired = 600;
  //Ignition point top motor position
  topDesired = 600;
  //Move motors to locations for iginition
  moveMotors(0);
  //Msg for HMI status bar
  Serial.println("MSG: Ignite Propane Torches");

  //Save bottom starting temperature
  double startingBotTemp = temps[0];
  //Save top starting temperature
  double startingTopTemp = temps[1];
  //Enter while loop when bottom temperature is 3 degC greater than
  //starting temp
  while(temps[0] < 3 + startingBotTemp){
    //Print motor positions and temperatures to HMI
    printStatus();
  }
  //Indicates torches have been ignited
  Serial.println("CMD: Ignited");
  //Msg for status bar
  Serial.println("MSG: Fires lit, waiting for SS target");
}


//Moves motors to next positions
void moveMotors(int mode){

  //test to see if a move is desired
  if((bottomMotorPosition != bottomDesired) || (topMotorPosition !=
topDesired)){

    //constrain and calculate steps to move
    bottomSteps = constrain(bottomDesired, startPosition, endPosition) -
bottomMotorPosition;
    topSteps = constrain(topDesired, startPosition, endPosition) -
topMotorPosition;

    //set speed if if it has changed
    if(speedVal != constrain(newSpeedVal, 10, 600)){
      Serial.print("MSG: new speed ");
      Serial.print(constrain(newSpeedVal, 10, 600));
      Serial.println("");
      speedVal = constrain(newSpeedVal, 10, 600);
      usperstep = bottomMotor.setSpeed(speedVal);
      usperstep = topMotor.setSpeed((speedVal));
    }

    //set motor movement direction
    if(bottomSteps > 0){
      bottomDir = BACKWARD;
    }
    else if (bottomSteps < 0){
      bottomDir =  FORWARD;
      bottomSteps = -1*bottomSteps;
```

```
    }
    if(topSteps > 0){
      topDir = BACKWARD;
    }
    else if (topSteps < 0){
      topDir =  FORWARD;
      topSteps = -1*topSteps;
    }

    //Move motors!
    uint8_t retBottom = 0;
    uint8_t retTop = 0;
    int count = 0;
    while((bottomSteps > 0) || (topSteps > 0)){
      count ++;
      EEPROM.write(4,1);
      if(bottomSteps > 0){
        retBottom = bottomMotor.onestep(bottomDir, DOUBLE);
        bottomSteps --;
      }
      if(topSteps > 0){
        retTop = topMotor.onestep(topDir, DOUBLE);
        topSteps --;
      }

      //Modes; mode(0) is currently the only mode
      //called mode(1) available if desired
      if(mode == 0){
        //do nothing
      }
      else if(mode == 1){
        //check for serial
        parseSerial();
      }
      //Set timing of motor movement using a delay
      delay(usperstep/1000);
    }
    //Save motor positions
    saveMotorPositions(bottomDesired, topDesired);
    //Release motors to prevent overheating
    bottomMotor.release();
    topMotor.release();
  }
}


//Save motor positions for reference
void saveMotorPositions(int bottomPosition, int topPosition){

  //Update motor positions:
    bottomMotorPosition = bottomPosition;
    topMotorPosition = topPosition;

    //save bottom motor position to EEPROM
    BMP_a = bottomMotorPosition/256;
    BMP_b = bottomMotorPosition;
    EEPROM.write(2,BMP_a);
```

```
    EEPROM.write(3,BMP_b);

    //save top motor position to EEPROM
    TMP_a = topMotorPosition/256;
    TMP_b = topMotorPosition;
    EEPROM.write(0,TMP_a);
    EEPROM.write(1,TMP_b);
    EEPROM.write(4,0);
}


//Checks for data from HMI after the test has started
void parseSerial(){
  if(Serial.available()){
    while(1){
      while(Serial.available() > 0) {

        //correct motor positions
        int topActual;
        int bottomActual;
        //Checking for and setting correct motor positions
        if(topDesired != topMotorPosition){
          topActual = topDesired + (topMotorPosition -
topDesired)/abs(topMotorPosition - topDesired)*topSteps;
        }
        else{
          topActual = topMotorPosition;
        }
        if(bottomDesired != bottomMotorPosition){
          bottomActual = bottomDesired + (bottomMotorPosition -
bottomDesired)/abs(bottomMotorPosition - bottomDesired)*bottomSteps;
        }
        else{
          bottomActual = bottomMotorPosition;
        }
        //Save motor positions
        saveMotorPositions(bottomActual, topActual);

        //Record new desired motor positions
        bottomDesired = Serial.parseInt();
        topDesired = Serial.parseInt();
        newSpeedVal = Serial.parseInt();
        moveMotors(0);      //Move motors
        Serial.println("CMD: Stopped");
      }
    }
  }
}


//read in temperatures from MAX31855 breakout boards
void readTemps(double *ptemps)
{
  //read values until acceptable
  double val_1;
//Bottom temp temperature
```

```
  double val_2;
//Top temp temperature
  //Power MAX31855 boards on and off to avoid ground looping effects
  thermoReadTime = millis();
  digitalWrite(thermo1VIN, HIGH);
//Power bottom thermocouple board on
  while(1){
    val_1 = thermo1.readCelsius();
//Store bottom temperature
    if((!isnan(val_1))&(!(val_1==0))){
      digitalWrite(thermo1VIN, LOW);
//Power bottom thermocouple board off
      digitalWrite(thermo2VIN, HIGH);
//Power top thermocouple board on
      while(1){
        val_2 = thermo2.readCelsius();
//Store top temperature
        if((!isnan(val_2))&(!(val_2==0))){
          digitalWrite(thermo2VIN, LOW);
//Power top thermocouple pin off
          break;
        }
      }
      break;
    }
  }
  thermoReadTime = millis() - thermoReadTime;

  //Cold bath temperature offsets
  const float b_o = -1.25;
//Bottom thermocouple offset (board specific)
  const float t_o = -0.5;
//Top thermocouple offset (board specific)

  //Apply cold offset to temps and save
  ptemps[0] = val_1 + b_o;
//bottom temperature
  ptemps[1] = val_2 + t_o;
//top temperature
}


//Print data to serial port
void printStatus(){
  readTemps(&temps[0]);
//Access temperature array

  //Bottom Motor Position:
  Serial.print(bottomMotorPosition);
  Serial.print("\t");

  //Bottom Temperature:
  Serial.print(temps[0]);
  Serial.print("\t");

  //Top Motor Position:
  Serial.print(topMotorPosition);
```

```
  Serial.print("\t");

  //Top Temperature:
  Serial.println(temps[1]);
}
```

AFMotor.h:

```
// Adafruit Motor shield library
// copyright Adafruit Industries LLC, 2009
// this code is public domain, enjoy!

/*
 * Usage Notes:
 * For PIC32, all features work properly with the following two
 * exceptions:
 *
 * 1) Because the PIC32 only has 5 PWM outputs, and the AFMotor shield
 *    needs 6 to completely operate (four for motor outputs and two for RC
 *    servos), the M1 motor output will not have PWM ability when used
 *    with a PIC32 board. However, there is a very simple workaround. If
 *    you need to drive a stepper or DC motor with PWM on motor output M1,
 *    you can use the PWM output on pin 9 or pin 10 (normally use for RC
 *    servo outputs on Arduino, not needed for RC servo outputs on PIC32)
 *    to drive the PWM input for M1 by simply putting a jumper from pin 9
 *    to pin 11 or pin 10 to pin 11. Then uncomment one of the two
 *    #defines below to activate the PWM on either pin 9 or pin 10. You
 *    will then have a fully functional micro-stepping for 2 stepper
 *    motors, or four DC motor outputs with PWM.
 *
 * 2) There is a conflict between RC Servo outputs on pins 9 and pins 10
 *    and the operation of DC motors and stepper motors as of 9/2012. This
 *    issue will get fixed in future MPIDE releases, but at the present
 *    time it means that the Motor Party example will NOT work properly.
 *    Any time you attach an RC servo to pins 9 or pins 10, ALL PWM
 *    outputs on the whole board will stop working. Thus no steppers or DC
 *    motors.
 *
 */
// <BPS> 09/15/2012 Modified for use with chipKIT boards


#ifndef _AFMotor_h_
#define _AFMotor_h_

#include <inttypes.h>
#if defined(__AVR__)
    #include <avr/io.h>

    //#define MOTORDEBUG 1

    #define MICROSTEPS 16                       // 8 or 16

    #define MOTOR12_64KHZ _BV(CS20)             // no prescale
    #define MOTOR12_8KHZ _BV(CS21)              // divide by 8
    #define MOTOR12_2KHZ _BV(CS21) | _BV(CS20)  // divide by 32
```

```
    #define MOTOR12_1KHZ _BV(CS22)              // divide by 64

    #define MOTOR34_64KHZ _BV(CS00)             // no prescale
    #define MOTOR34_8KHZ _BV(CS01)              // divide by 8
    #define MOTOR34_1KHZ _BV(CS01) | _BV(CS00)  // divide by 64

    #define DC_MOTOR_PWM_RATE   MOTOR34_8KHZ    // PWM rate for DC motors
    #define STEPPER1_PWM_RATE   MOTOR12_64KHZ   // PWM rate for stepper 1
    #define STEPPER2_PWM_RATE   MOTOR34_64KHZ   // PWM rate for stepper 2

#elif defined(__PIC32MX__)
    //#define MOTORDEBUG 1

    // Uncomment the one of following lines if you have put a jumper from
    // either pin 9 to pin 11 or pin 10 to pin 11 on your Motor Shield.
    // Either will enable PWM for M1
    //#define PIC32_USE_PIN9_FOR_M1_PWM
    //#define PIC32_USE_PIN10_FOR_M1_PWM

    #define MICROSTEPS 16       // 8 or 16

    // For PIC32 Timers, define prescale settings by PWM frequency
    #define MOTOR12_312KHZ  0   // 1:1, actual frequency 312KHz
    #define MOTOR12_156KHZ  1   // 1:2, actual frequency 156KHz
    #define MOTOR12_64KHZ   2   // 1:4, actual frequency 78KHz
    #define MOTOR12_39KHZ   3   // 1:8, acutal frequency 39KHz
    #define MOTOR12_19KHZ   4   // 1:16, actual frequency 19KHz
    #define MOTOR12_8KHZ    5   // 1:32, actual frequency 9.7KHz
    #define MOTOR12_4_8KHZ  6   // 1:64, actual frequency 4.8KHz
    #define MOTOR12_2KHZ    7   // 1:256, actual frequency 1.2KHz
    #define MOTOR12_1KHZ    7   // 1:256, actual frequency 1.2KHz

    #define MOTOR34_312KHZ  0   // 1:1, actual frequency 312KHz
    #define MOTOR34_156KHZ  1   // 1:2, actual frequency 156KHz
    #define MOTOR34_64KHZ   2   // 1:4, actual frequency 78KHz
    #define MOTOR34_39KHZ   3   // 1:8, acutal frequency 39KHz
    #define MOTOR34_19KHZ   4   // 1:16, actual frequency 19KHz
    #define MOTOR34_8KHZ    5   // 1:32, actual frequency 9.7KHz
    #define MOTOR34_4_8KHZ  6   // 1:64, actual frequency 4.8KHz
    #define MOTOR34_2KHZ    7   // 1:256, actual frequency 1.2KHz
    #define MOTOR34_1KHZ    7   // 1:256, actual frequency 1.2KHz

    // PWM rate for DC motors.
    #define DC_MOTOR_PWM_RATE   MOTOR34_39KHZ
    // Note: for PIC32, both of these must be set to the same value
    // since there's only one timebase for all 4 PWM outputs
    #define STEPPER1_PWM_RATE   MOTOR12_39KHZ
    #define STEPPER2_PWM_RATE   MOTOR34_39KHZ

#endif

// Bit positions in the 74HCT595 shift register output
#define MOTOR1_A 2
#define MOTOR1_B 3
#define MOTOR2_A 1
#define MOTOR2_B 4
#define MOTOR4_A 0
```

```
#define MOTOR4_B 6
#define MOTOR3_A 5
#define MOTOR3_B 7

// Constants that the user passes in to the motor calls
#define FORWARD 1
#define BACKWARD 2
#define BRAKE 3
#define RELEASE 4

// Constants that the user passes in to the stepper calls
#define SINGLE 1
#define DOUBLE 2
#define INTERLEAVE 3
#define MICROSTEP 4

/*
#define LATCH 4
#define LATCH_DDR DDRB
#define LATCH_PORT PORTB

#define CLK_PORT PORTD
#define CLK_DDR DDRD
#define CLK 4

#define ENABLE_PORT PORTD
#define ENABLE_DDR DDRD
#define ENABLE 7

#define SER 0
#define SER_DDR DDRB
#define SER_PORT PORTB
*/

// Arduino pin names for interface to 74HCT595 latch
#define MOTORLATCH 12
#define MOTORCLK 4
#define MOTORENABLE 7
#define MOTORDATA 8

class AFMotorController
{
  public:
    AFMotorController(void);
    void enable(void);
    friend class AF_DCMotor;
    void latch_tx(void);
    uint8_t TimerInitalized;
};

class AF_DCMotor
{
 public:
  AF_DCMotor(uint8_t motornum, uint8_t freq = DC_MOTOR_PWM_RATE);
  void run(uint8_t);
  void setSpeed(uint8_t);
```

```
 private:
  uint8_t motornum, pwmfreq;
};

class AF_Stepper {
 public:
  AF_Stepper(uint16_t, uint8_t);
  void step(uint16_t steps, uint8_t dir,  uint8_t style = SINGLE);
  uint32_t setSpeed(uint16_t);
  uint8_t onestep(uint8_t dir, uint8_t style);
  void release(void);
  uint16_t revsteps; // # steps per revolution
  uint8_t steppernum;
  uint32_t usperstep, steppingcounter;
 private:
  uint8_t currentstep;

};

uint8_t getlatchstate(void);

#endif
```

AFMotor.cpp:

```
// Adafruit Motor shield library
// copyright Adafruit Industries LLC, 2009
// this code is public domain, enjoy!


#if (ARDUINO >= 100)
  #include "Arduino.h"
#else
  #if defined(__AVR__)
    #include <avr/io.h>
  #endif
  #include "WProgram.h"
#endif

#include "AFMotor.h"



static uint8_t latch_state;

#if (MICROSTEPS == 8)
uint8_t microstepcurve[] = {0, 50, 98, 142, 180, 212, 236, 250, 255};
#elif (MICROSTEPS == 16)
uint8_t microstepcurve[] = {0, 25, 50, 74, 98, 120, 141, 162, 180, 197,
212, 225, 236, 244, 250, 253, 255};
#endif

AFMotorController::AFMotorController(void) {
    TimerInitalized = false;
}
```

```
void AFMotorController::enable(void) {
  // setup the latch
  /*
  LATCH_DDR |= _BV(LATCH);
  ENABLE_DDR |= _BV(ENABLE);
  CLK_DDR |= _BV(CLK);
  SER_DDR |= _BV(SER);
  */
  pinMode(MOTORLATCH, OUTPUT);
  pinMode(MOTORENABLE, OUTPUT);
  pinMode(MOTORDATA, OUTPUT);
  pinMode(MOTORCLK, OUTPUT);

  latch_state = 0;

  latch_tx();  // "reset"

  //ENABLE_PORT &= ~_BV(ENABLE); // enable the chip outputs!
  digitalWrite(MOTORENABLE, LOW);
}


void AFMotorController::latch_tx(void) {
  uint8_t i;

  //LATCH_PORT &= ~_BV(LATCH);
  digitalWrite(MOTORLATCH, LOW);

  //SER_PORT &= ~_BV(SER);
  digitalWrite(MOTORDATA, LOW);

  for (i=0; i<8; i++) {
    //CLK_PORT &= ~_BV(CLK);
    digitalWrite(MOTORCLK, LOW);

    if (latch_state & _BV(7-i)) {
      //SER_PORT |= _BV(SER);
      digitalWrite(MOTORDATA, HIGH);
    } else {
      //SER_PORT &= ~_BV(SER);
      digitalWrite(MOTORDATA, LOW);
    }
    //CLK_PORT |= _BV(CLK);
    digitalWrite(MOTORCLK, HIGH);
  }
  //LATCH_PORT |= _BV(LATCH);
  digitalWrite(MOTORLATCH, HIGH);
}

static AFMotorController MC;

/******************************************
               MOTORS
******************************************/
inline void initPWM1(uint8_t freq) {
#if defined(__AVR_ATmega8__) || \
    defined(__AVR_ATmega48__) || \
```

```
    defined(__AVR_ATmega88__) || \
    defined(__AVR_ATmega168__) || \
    defined(__AVR_ATmega328P__)
    // use PWM from timer2A on PB3 (Arduino pin #11)
    TCCR2A |= _BV(COM2A1) | _BV(WGM20) | _BV(WGM21);
    TCCR2B = freq & 0x7;
    // fast PWM, turn on oc2a
    OCR2A = 0;
#elif defined(__AVR_ATmega1280__) || defined(__AVR_ATmega2560__)
    // on arduino mega, pin 11 is now PB5 (OC1A)
    TCCR1A |= _BV(COM1A1) | _BV(WGM10); // fast PWM, turn on oc1a
    TCCR1B = (freq & 0x7) | _BV(WGM12);
    OCR1A = 0;
#elif defined(__PIC32MX__)
    #if defined(PIC32_USE_PIN9_FOR_M1_PWM)
        // Make sure that pin 11 is an input, since we have tied together
        // 9 and 11
        pinMode(9, OUTPUT);
        pinMode(11, INPUT);
        if (!MC.TimerInitalized)
        {   // Set up Timer2 for 80MHz counting from 0 to 256
            // ON=1, FRZ=0, SIDL=0, TGATE=0, TCKPS=<freq>, T32=0, TCS=0;
            // ON=1, FRZ=0, SIDL=0, TGATE=0, TCKPS=0, T32=0, TCS=0
            T2CON = 0x8000 | ((freq & 0x07) << 4);
            TMR2 = 0x0000;
            PR2 = 0x0100;
            MC.TimerInitalized = true;
        }
         // Setup OC4 (pin 9) in PWM mode, with Timer2 as timebase
        OC4CON = 0x8006;    // OC32 = 0, OCTSEL=0, OCM=6
        OC4RS = 0x0000;
        OC4R = 0x0000;
    #elif defined(PIC32_USE_PIN10_FOR_M1_PWM)
        // Make sure that pin 11 is an input, since we have tied together
        // 9 and 11
        pinMode(10, OUTPUT);
        pinMode(11, INPUT);
        if (!MC.TimerInitalized)
        {   // Set up Timer2 for 80MHz counting from 0 to 256’
            // ON=1, FRZ=0, SIDL=0, TGATE=0, TCKPS=<freq>, T32=0, TCS=0;
            // ON=1, FRZ=0, SIDL=0, TGATE=0, TCKPS=0, T32=0, TCS=0
            T2CON = 0x8000 | ((freq & 0x07) << 4);
            TMR2 = 0x0000;
            PR2 = 0x0100;
            MC.TimerInitalized = true;
        }
         // Setup OC5 (pin 10) in PWM mode, with Timer2 as timebase
        OC5CON = 0x8006;    // OC32 = 0, OCTSEL=0, OCM=6
        OC5RS = 0x0000;
        OC5R = 0x0000;
    #else
        // If we are not using PWM for pin 11, then just do digital
        digitalWrite(11, LOW);
    #endif
#else
   #error "This chip is not supported!"
#endif
```

```
    #if !defined(PIC32_USE_PIN9_FOR_M1_PWM) &&
!defined(PIC32_USE_PIN10_FOR_M1_PWM)
        pinMode(11, OUTPUT);
    #endif
}

inline void setPWM1(uint8_t s) {
#if defined(__AVR_ATmega8__) || \
    defined(__AVR_ATmega48__) || \
    defined(__AVR_ATmega88__) || \
    defined(__AVR_ATmega168__) || \
    defined(__AVR_ATmega328P__)
    // use PWM from timer2A on PB3 (Arduino pin #11)
    OCR2A = s;
#elif defined(__AVR_ATmega1280__) || defined(__AVR_ATmega2560__)
    // on arduino mega, pin 11 is now PB5 (OC1A)
    OCR1A = s;
#elif defined(__PIC32MX__)
    #if defined(PIC32_USE_PIN9_FOR_M1_PWM)
        // Set the OC4 (pin 9) PMW duty cycle from 0 to 255
        OC4RS = s;
    #elif defined(PIC32_USE_PIN10_FOR_M1_PWM)
        // Set the OC5 (pin 10) PMW duty cycle from 0 to 255
        OC5RS = s;
    #else
        // If we are not doing PWM output for M1, then just use on/off
        if (s > 127)
        {
            digitalWrite(11, HIGH);
        }
        else
        {
            digitalWrite(11, LOW);
        }
    #endif
#else
   #error "This chip is not supported!"
#endif
}

inline void initPWM2(uint8_t freq) {
#if defined(__AVR_ATmega8__) || \
    defined(__AVR_ATmega48__) || \
    defined(__AVR_ATmega88__) || \
    defined(__AVR_ATmega168__) || \
    defined(__AVR_ATmega328P__)
    // use PWM from timer2B (pin 3)
    TCCR2A |= _BV(COM2B1) | _BV(WGM20) | _BV(WGM21);
    TCCR2B = freq & 0x7;
    // fast PWM, turn on oc2b
    OCR2B = 0;
#elif defined(__AVR_ATmega1280__) || defined(__AVR_ATmega2560__)
    // on arduino mega, pin 3 is now PE5 (OC3C)
    TCCR3A |= _BV(COM1C1) | _BV(WGM10); // fast PWM, turn on oc3c
    TCCR3B = (freq & 0x7) | _BV(WGM12);
    OCR3C = 0;
#elif defined(__PIC32MX__)
```

```
    if (!MC.TimerInitalized)
    {   // Set up Timer2 for 80MHz counting from 0 to 256
        // ON=1, FRZ=0, SIDL=0, TGATE=0, TCKPS=<freq>, T32=0, TCS=0;
        // ON=1, FRZ=0, SIDL=0, TGATE=0, TCKPS=0, T32=0, TCS=0
        T2CON = 0x8000 | ((freq & 0x07) << 4);
        TMR2 = 0x0000;
        PR2 = 0x0100;
        MC.TimerInitalized = true;
    }
    // Setup OC1 (pin3) in PWM mode, with Timer2 as timebase
    OC1CON = 0x8006;    // OC32 = 0, OCTSEL=0, OCM=6
    OC1RS = 0x0000;
    OC1R = 0x0000;
#else
   #error "This chip is not supported!"
#endif

    pinMode(3, OUTPUT);
}

inline void setPWM2(uint8_t s) {
#if defined(__AVR_ATmega8__) || \
    defined(__AVR_ATmega48__) || \
    defined(__AVR_ATmega88__) || \
    defined(__AVR_ATmega168__) || \
    defined(__AVR_ATmega328P__)
    // use PWM from timer2A on PB3 (Arduino pin #11)
    OCR2B = s;
#elif defined(__AVR_ATmega1280__) || defined(__AVR_ATmega2560__)
    // on arduino mega, pin 11 is now PB5 (OC1A)
    OCR3C = s;
#elif defined(__PIC32MX__)
    // Set the OC1 (pin3) PMW duty cycle from 0 to 255
    OC1RS = s;
#else
   #error "This chip is not supported!"
#endif
}

inline void initPWM3(uint8_t freq) {
#if defined(__AVR_ATmega8__) || \
    defined(__AVR_ATmega48__) || \
    defined(__AVR_ATmega88__) || \
    defined(__AVR_ATmega168__) || \
    defined(__AVR_ATmega328P__)
    // use PWM from timer0A / PD6 (pin 6)
    TCCR0A |= _BV(COM0A1) | _BV(WGM00) | _BV(WGM01); // fast PWM, turn on
OC0A
    //TCCR0B = freq & 0x7;
    OCR0A = 0;
#elif defined(__AVR_ATmega1280__) || defined(__AVR_ATmega2560__)
    // on arduino mega, pin 6 is now PH3 (OC4A)
    TCCR4A |= _BV(COM1A1) | _BV(WGM10); // fast PWM, turn on oc4a
    TCCR4B = (freq & 0x7) | _BV(WGM12);
    //TCCR4B = 1 | _BV(WGM12);
    OCR4A = 0;
#elif defined(__PIC32MX__)
```

```
    if (!MC.TimerInitalized)
    {   // Set up Timer2 for 80MHz counting from 0 to 256
        // ON=1, FRZ=0, SIDL=0, TGATE=0, TCKPS=<freq>, T32=0, TCS=0;
        // ON=1, FRZ=0, SIDL=0, TGATE=0, TCKPS=0, T32=0, TCS=0
        T2CON = 0x8000 | ((freq & 0x07) << 4);
        TMR2 = 0x0000;
        PR2 = 0x0100;
        MC.TimerInitalized = true;
    }
    // Setup OC3 (pin 6) in PWM mode, with Timer2 as timebase
    OC3CON = 0x8006;    // OC32 = 0, OCTSEL=0, OCM=6
    OC3RS = 0x0000;
    OC3R = 0x0000;
#else
   #error "This chip is not supported!"
#endif
    pinMode(6, OUTPUT);
}

inline void setPWM3(uint8_t s) {
#if defined(__AVR_ATmega8__) || \
    defined(__AVR_ATmega48__) || \
    defined(__AVR_ATmega88__) || \
    defined(__AVR_ATmega168__) || \
    defined(__AVR_ATmega328P__)
    // use PWM from timer0A on PB3 (Arduino pin #6)
    OCR0A = s;
#elif defined(__AVR_ATmega1280__) || defined(__AVR_ATmega2560__)
    // on arduino mega, pin 6 is now PH3 (OC4A)
    OCR4A = s;
#elif defined(__PIC32MX__)
    // Set the OC3 (pin 6) PMW duty cycle from 0 to 255
    OC3RS = s;
#else
   #error "This chip is not supported!"
#endif
}



inline void initPWM4(uint8_t freq) {
#if defined(__AVR_ATmega8__) || \
    defined(__AVR_ATmega48__) || \
    defined(__AVR_ATmega88__) || \
    defined(__AVR_ATmega168__) || \
    defined(__AVR_ATmega328P__)
    // use PWM from timer0B / PD5 (pin 5)
    TCCR0A |= _BV(COM0B1) | _BV(WGM00) | _BV(WGM01);
    // fast PWM, turn on oc0a
    //TCCR0B = freq & 0x7;
    OCR0B = 0;
#elif defined(__AVR_ATmega1280__) || defined(__AVR_ATmega2560__)
    // on arduino mega, pin 5 is now PE3 (OC3A)
    TCCR3A |= _BV(COM1A1) | _BV(WGM10); // fast PWM, turn on oc3a
    TCCR3B = (freq & 0x7) | _BV(WGM12);
    //TCCR4B = 1 | _BV(WGM12);
    OCR3A = 0;
```

```
#elif defined(__PIC32MX__)
    if (!MC.TimerInitalized)
    {   // Set up Timer2 for 80MHz counting from 0 to 256
        // ON=1, FRZ=0, SIDL=0, TGATE=0, TCKPS=<freq>, T32=0, TCS=0;
        // ON=1, FRZ=0, SIDL=0, TGATE=0, TCKPS=0, T32=0, TCS=0
        T2CON = 0x8000 | ((freq & 0x07) << 4);
        TMR2 = 0x0000;
        PR2 = 0x0100;
        MC.TimerInitalized = true;
    }
    // Setup OC2 (pin 5) in PWM mode, with Timer2 as timebase
    OC2CON = 0x8006;    // OC32 = 0, OCTSEL=0, OCM=6
    OC2RS = 0x0000;
    OC2R = 0x0000;
#else
   #error "This chip is not supported!"
#endif
    pinMode(5, OUTPUT);
}

inline void setPWM4(uint8_t s) {
#if defined(__AVR_ATmega8__) || \
    defined(__AVR_ATmega48__) || \
    defined(__AVR_ATmega88__) || \
    defined(__AVR_ATmega168__) || \
    defined(__AVR_ATmega328P__)
    // use PWM from timer0A on PB3 (Arduino pin #6)
    OCR0B = s;
#elif defined(__AVR_ATmega1280__) || defined(__AVR_ATmega2560__)
    // on arduino mega, pin 6 is now PH3 (OC4A)
    OCR3A = s;
#elif defined(__PIC32MX__)
    // Set the OC2 (pin 5) PMW duty cycle from 0 to 255
    OC2RS = s;
#else
   #error "This chip is not supported!"
#endif
}

AF_DCMotor::AF_DCMotor(uint8_t num, uint8_t freq) {
  motornum = num;
  pwmfreq = freq;

  MC.enable();

  switch (num) {
  case 1:
    latch_state &= ~_BV(MOTOR1_A) & ~_BV(MOTOR1_B);
    // set both motor pins to 0
    MC.latch_tx();
    initPWM1(freq);
    break;
  case 2:
    latch_state &= ~_BV(MOTOR2_A) & ~_BV(MOTOR2_B);
    // set both motor pins to 0
    MC.latch_tx();
    initPWM2(freq);
```

```
    break;
  case 3:
    latch_state &= ~_BV(MOTOR3_A) & ~_BV(MOTOR3_B);
    // set both motor pins to 0
    MC.latch_tx();
    initPWM3(freq);
    break;
  case 4:
    latch_state &= ~_BV(MOTOR4_A) & ~_BV(MOTOR4_B);
    // set both motor pins to 0
    MC.latch_tx();
    initPWM4(freq);
    break;
  }
}

void AF_DCMotor::run(uint8_t cmd) {
  uint8_t a, b;
  switch (motornum) {
  case 1:
    a = MOTOR1_A; b = MOTOR1_B; break;
  case 2:
    a = MOTOR2_A; b = MOTOR2_B; break;
  case 3:
    a = MOTOR3_A; b = MOTOR3_B; break;
  case 4:
    a = MOTOR4_A; b = MOTOR4_B; break;
  default:
    return;
  }

  switch (cmd) {
  case FORWARD:
    latch_state |= _BV(a);
    latch_state &= ~_BV(b);
    MC.latch_tx();
    break;
  case BACKWARD:
    latch_state &= ~_BV(a);
    latch_state |= _BV(b);
    MC.latch_tx();
    break;
  case RELEASE:
    latch_state &= ~_BV(a);     // A and B both low
    latch_state &= ~_BV(b);
    MC.latch_tx();
    break;
  }
}

void AF_DCMotor::setSpeed(uint8_t speed) {
  switch (motornum) {
  case 1:
    setPWM1(speed); break;
  case 2:
    setPWM2(speed); break;
  case 3:
```

```
    setPWM3(speed); break;
  case 4:
    setPWM4(speed); break;
  }
}

/******************************************
               STEPPERS
******************************************/

AF_Stepper::AF_Stepper(uint16_t steps, uint8_t num) {
  MC.enable();

  revsteps = steps;
  steppernum = num;
  currentstep = 0;

  if (steppernum == 1) {
    latch_state &= ~_BV(MOTOR1_A) & ~_BV(MOTOR1_B) &
      ~_BV(MOTOR2_A) & ~_BV(MOTOR2_B); // all motor pins to 0
    MC.latch_tx();

    // enable both H bridges
    pinMode(11, OUTPUT);
    pinMode(3, OUTPUT);
    digitalWrite(11, HIGH);
    digitalWrite(3, HIGH);

    // use PWM for micro-stepping support
    initPWM1(STEPPER1_PWM_RATE);
    initPWM2(STEPPER1_PWM_RATE);
    setPWM1(255);
    setPWM2(255);

  } else if (steppernum == 2) {
    latch_state &= ~_BV(MOTOR3_A) & ~_BV(MOTOR3_B) &
      ~_BV(MOTOR4_A) & ~_BV(MOTOR4_B); // all motor pins to 0
    MC.latch_tx();

    // enable both H bridges
    pinMode(5, OUTPUT);
    pinMode(6, OUTPUT);
    digitalWrite(5, HIGH);
    digitalWrite(6, HIGH);

    // use PWM for micro-stepping support
    // use PWM for micro-stepping support
    initPWM3(STEPPER2_PWM_RATE);
    initPWM4(STEPPER2_PWM_RATE);
    setPWM3(255);
    setPWM4(255);
  }
}

uint32_t AF_Stepper::setSpeed(uint16_t rpm) {
  usperstep = 60000000 / ((uint32_t)revsteps * (uint32_t)rpm);
  steppingcounter = 0;
```

```
  return usperstep;
}

void AF_Stepper::release(void) {
  if (steppernum == 1) {
    latch_state &= ~_BV(MOTOR1_A) & ~_BV(MOTOR1_B) &
      ~_BV(MOTOR2_A) & ~_BV(MOTOR2_B); // all motor pins to 0
    MC.latch_tx();
  } else if (steppernum == 2) {
    latch_state &= ~_BV(MOTOR3_A) & ~_BV(MOTOR3_B) &
      ~_BV(MOTOR4_A) & ~_BV(MOTOR4_B); // all motor pins to 0
    MC.latch_tx();
  }
}

void AF_Stepper::step(uint16_t steps, uint8_t dir,  uint8_t style) {
  uint32_t uspers = usperstep;
  uint8_t ret = 0;

  if (style == INTERLEAVE) {
    uspers /= 2;
  }
 else if (style == MICROSTEP) {
    uspers /= MICROSTEPS;
    steps *= MICROSTEPS;
#ifdef MOTORDEBUG
    Serial.print("steps = "); Serial.println(steps, DEC);
#endif
  }

  while (steps--) {
    ret = onestep(dir, style);
    delay(uspers/1000); // in ms
    steppingcounter += (uspers % 1000);
    if (steppingcounter >= 1000) {
      delay(1);
      steppingcounter -= 1000;
    }
  }
  if (style == MICROSTEP) {
    while ((ret != 0) && (ret != MICROSTEPS)) {
      ret = onestep(dir, style);
      delay(uspers/1000); // in ms
      steppingcounter += (uspers % 1000);
      if (steppingcounter >= 1000) {
    delay(1);
    steppingcounter -= 1000;
      }
    }
  }
}

uint8_t AF_Stepper::onestep(uint8_t dir, uint8_t style) {
  uint8_t a, b, c, d;
  uint8_t ocrb, ocra;
```

```
ocra = ocrb = 255;

if (steppernum == 1) {
  a = _BV(MOTOR1_A);
  b = _BV(MOTOR2_A);
  c = _BV(MOTOR1_B);
  d = _BV(MOTOR2_B);
} else if (steppernum == 2) {
  a = _BV(MOTOR3_A);
  b = _BV(MOTOR4_A);
  c = _BV(MOTOR3_B);
  d = _BV(MOTOR4_B);
} else {
  return 0;
}

// next determine what sort of stepping procedure we're up to
if (style == SINGLE) {
  if ((currentstep/(MICROSTEPS/2)) % 2) { // we're at an odd step, weird
    if (dir == FORWARD) {
  currentstep += MICROSTEPS/2;
    }
    else {
  currentstep -= MICROSTEPS/2;
    }
  } else {           // go to the next even step
    if (dir == FORWARD) {
  currentstep += MICROSTEPS;
    }
    else {
  currentstep -= MICROSTEPS;
    }
  }
} else if (style == DOUBLE) {
  if (! (currentstep/(MICROSTEPS/2) % 2)) {
    // we're at an even step, weird
    if (dir == FORWARD) {
  currentstep += MICROSTEPS/2;
    } else {
  currentstep -= MICROSTEPS/2;
    }
  } else {           // go to the next odd step
    if (dir == FORWARD) {
  currentstep += MICROSTEPS;
    } else {
  currentstep -= MICROSTEPS;
    }
  }
} else if (style == INTERLEAVE) {
  if (dir == FORWARD) {
     currentstep += MICROSTEPS/2;
  } else {
     currentstep -= MICROSTEPS/2;
  }
}

if (style == MICROSTEP) {
```

```
    if (dir == FORWARD) {
      currentstep++;
    } else {
      // BACKWARDS
      currentstep--;
    }

    currentstep += MICROSTEPS*4;
    currentstep %= MICROSTEPS*4;

    ocra = ocrb = 0;
    if ( (currentstep >= 0) && (currentstep < MICROSTEPS)) {
      ocra = microstepcurve[MICROSTEPS - currentstep];
      ocrb = microstepcurve[currentstep];
    } else if  ( (currentstep >= MICROSTEPS) && (currentstep <
MICROSTEPS*2)) {
      ocra = microstepcurve[currentstep - MICROSTEPS];
      ocrb = microstepcurve[MICROSTEPS*2 - currentstep];
    } else if  ( (currentstep >= MICROSTEPS*2) && (currentstep <
MICROSTEPS*3)) {
      ocra = microstepcurve[MICROSTEPS*3 - currentstep];
      ocrb = microstepcurve[currentstep - MICROSTEPS*2];
    } else if  ( (currentstep >= MICROSTEPS*3) && (currentstep <
MICROSTEPS*4)) {
      ocra = microstepcurve[currentstep - MICROSTEPS*3];
      ocrb = microstepcurve[MICROSTEPS*4 - currentstep];
    }
  }

  currentstep += MICROSTEPS*4;
  currentstep %= MICROSTEPS*4;

#ifdef MOTORDEBUG
  Serial.print("current step: "); Serial.println(currentstep, DEC);
  Serial.print(" pwmA = "); Serial.print(ocra, DEC);
  Serial.print(" pwmB = "); Serial.println(ocrb, DEC);
#endif

  if (steppernum == 1) {
    setPWM1(ocra);
    setPWM2(ocrb);
  } else if (steppernum == 2) {
    setPWM3(ocra);
    setPWM4(ocrb);
  }


  // release all
  latch_state &= ~a & ~b & ~c & ~d; // all motor pins to 0

  //Serial.println(step, DEC);
  if (style == MICROSTEP) {
    if ((currentstep >= 0) && (currentstep < MICROSTEPS))
      latch_state |= a | b;
    if ((currentstep >= MICROSTEPS) && (currentstep < MICROSTEPS*2))
      latch_state |= b | c;
    if ((currentstep >= MICROSTEPS*2) && (currentstep < MICROSTEPS*3))
```

```
      latch_state |= c | d;
    if ((currentstep >= MICROSTEPS*3) && (currentstep < MICROSTEPS*4))
      latch_state |= d | a;
  } else {
    switch (currentstep/(MICROSTEPS/2)) {
    case 0:
      latch_state |= a; // energize coil 1 only
      break;
    case 1:
      latch_state |= a | b; // energize coil 1+2
      break;
    case 2:
      latch_state |= b; // energize coil 2 only
      break;
    case 3:
      latch_state |= b | c; // energize coil 2+3
      break;
    case 4:
      latch_state |= c; // energize coil 3 only
      break;
    case 5:
      latch_state |= c | d; // energize coil 3+4
      break;
    case 6:
      latch_state |= d; // energize coil 4 only
      break;
    case 7:
      latch_state |= d | a; // energize coil 1+4
      break;
    }
  }


  MC.latch_tx();
  return currentstep;
}
```

Adafruit_MAX81855.h:

```
/***************************************************************************
  This is a library for the Adafruit Thermocouple Sensor w/MAX31855K

  Designed specifically to work with the Adafruit Thermocouple Sensor
  ----> https://www.adafruit.com/products/269

  These displays use SPI to communicate, 3 pins are required to
  Interface Adafruit invests time and resources providing this open
  Source code, please support Adafruit and open-source hardware by
  purchasing products from Adafruit!

  Written by Limor Fried/Ladyada for Adafruit Industries.
  BSD license, all text above must be included in any redistribution
 ****************************************************************************/


#if (ARDUINO >= 100)
```

```
 #include "Arduino.h"
#else
 #include "WProgram.h"
#endif

class Adafruit_MAX31855 {
 public:
  Adafruit_MAX31855(int8_t SCLK, int8_t CS, int8_t MISO);

  double readInternal(void);
  double readCelsius(void);
  double readFarenheit(void);
  uint8_t readError();

 private:
  int8_t sclk, miso, cs;
  uint32_t spiread32(void);
};
```

Adafruit_MAX81855.cpp:

```
/***************************************************************************
  This is a library for the Adafruit Thermocouple Sensor w/MAX31855K

  Designed specifically to work with the Adafruit Thermocouple Sensor
  ----> https://www.adafruit.com/products/269

  These displays use SPI to communicate, 3 pins are required to
  Interface Adafruit invests time and resources providing this open
  source code, please support Adafruit and open-source hardware by
  purchasing products from Adafruit!

  Written by Limor Fried/Ladyada for Adafruit Industries.
  BSD license, all text above must be included in any redistribution
 ****************************************************************************/

#include "Adafruit_MAX31855.h"
#include <avr/pgmspace.h>
#include <util/delay.h>
#include <stdlib.h>


Adafruit_MAX31855::Adafruit_MAX31855(int8_t SCLK, int8_t CS, int8_t MISO)
{
  sclk = SCLK;
  cs = CS;
  miso = MISO;

  //define pin modes
  pinMode(cs, OUTPUT);
  pinMode(sclk, OUTPUT);
  pinMode(miso, INPUT);

  digitalWrite(cs, HIGH);
}
```

```
double Adafruit_MAX31855::readInternal(void) {
  uint32_t v;

  v = spiread32();

  // ignore bottom 4 bits - they're just thermocouple data
  v >>= 4;

  // pull the bottom 11 bits off
  float internal = v & 0x7FF;
  internal *= 0.0625; // LSB = 0.0625 degrees
  // check sign bit!
  if (v & 0x800)
    internal *= -1;
  //Serial.print("\tInternal Temp: "); Serial.println(internal);
  return internal;
}

double Adafruit_MAX31855::readCelsius(void) {

  int32_t v;

  v = spiread32();

  //Serial.print("0x"); Serial.println(v, HEX);

  /*
  float internal = (v >> 4) & 0x7FF;
  internal *= 0.0625;
  if ((v >> 4) & 0x800)
    internal *= -1;
  Serial.print("\tInternal Temp: "); Serial.println(internal);
  */

  if (v & 0x7) {
    // uh oh, a serious problem!
    return NAN;
  }

  // get rid of internal temp data, and any fault bits
  v >>= 18;
  //Serial.println(v, HEX);

  double centigrade = v;

  // LSB = 0.25 degrees C
  centigrade *= 0.25;
  return centigrade;
}

uint8_t Adafruit_MAX31855::readError() {
  return spiread32() & 0x7;
}

double Adafruit_MAX31855::readFarenheit(void) {
  float f = readCelsius();
```

```
  f *= 9.0;
  f /= 5.0;
  f += 32;
  return f;
}

uint32_t Adafruit_MAX31855::spiread32(void) {
  int i;
  uint32_t d = 0;

  digitalWrite(sclk, LOW);
  _delay_ms(1);
  digitalWrite(cs, LOW);
  _delay_ms(1);

  for (i=31; i>=0; i--)
  {
    digitalWrite(sclk, LOW);
    _delay_ms(1);
    d <<= 1;
    if (digitalRead(miso)) {
      d |= 1;
    }

    digitalWrite(sclk, HIGH);
    _delay_ms(1);
  }

  digitalWrite(cs, HIGH);
  //Serial.println(d, HEX);
  return d;
}
```

PID_v1.h:

```
#ifndef PID_v1_h
#define PID_v1_h
#define LIBRARY_VERSION 1.0.0

class PID
{


  public:

  //Constants used in some of the functions below
  #define AUTOMATIC 1
  #define MANUAL    0
  #define DIRECT  0
  #define REVERSE  1

  //commonly used functions
***************************************************************************
    // * constructor.  links the PID to the Input, Output, and
    PID(double*, double*, double*,
        double, double, double, int);
```

```
    //   Setpoint.  Initial tuning parameters are also set here
    // * sets PID to either Manual (0) or Auto (non-0)
    void SetMode(int Mode);

    // * performs the PID calculation.  it should be
    //   called every time loop() cycles. ON/OFF and
    //   calculation frequency can be set using SetMode
    //   SetSampleTime respectively
    bool Compute();

    //clamps the output to a specific range. 0-255 by default, but
    //it's likely the user will want to change this depending on
    //the application
    void SetOutputLimits(double, double);

  //available but not commonly used functions
********************************************************
    // * While most users will set the tunings once in the
    //   constructor, this function gives the user the option
    //   of changing tunings during runtime for Adaptive control
    void SetTunings(double, double, double);

    // * Sets the Direction, or "Action" of the controller. DIRECT
    //   means the output will increase when error is positive. REVERSE
    //   means the opposite.  it's very unlikely that this will be needed
    //   once it is set in the constructor.
    // * sets the frequency, in Milliseconds, with which
    void SetControllerDirection(int);

    //   the PID calculation is performed.  default is 100
    void SetSampleTime(int);

  //Display functions
****************************************************************
    double GetKp();   // These functions query the pid for interal values.
    double GetKi();   //  they were created mainly for the pid front-end,
    double GetKd();   // where it's important to know what is actually
    int GetMode();    //  inside the PID.
    int GetDirection();

  private:
    void Initialize();

    double dispKp;    // * we'll hold on to the tuning parameters in user-
    double dispKi;    //   entered format for display purposes
    double dispKd;
    double kp;        // * (P)roportional Tuning Parameter
    double ki;        // * (I)ntegral Tuning Parameter
    double kd;        // * (D)erivative Tuning Parameter

    int controllerDirection;

    double *myInput;   // * Pointers to the Input, Output, and Setpoint
    double *myOutput;  //   This creates a hard link between the variables
    double *mySetpoint;//   and the variables PID, freeing the user from
                       //   having to constantly tell us  what these
                       //   values are.  with pointers we'll just know.
```

```
    unsigned long lastTime;
    double ITerm, lastInput;

    unsigned long SampleTime;
    double outMin, outMax;
    bool inAuto;
};
#endif
```

PID_v1.cpp:

```
/**********************************************************************************************
 * Arduino PID Library - Version 1.0.1
 * by Brett Beauregard <br3ttb@gmail.com> brettbeauregard.com
 *
 * This Library is licensed under a GPLv3 License

**********************************************************************************************/

#if ARDUINO >= 100
  #include "Arduino.h"
#else
  #include "WProgram.h"
#endif

#include <PID_v1.h>

/*Constructor (...)*********************************************************
 *    The parameters specified here are those for which we can't set
 *    up reliable defaults, so we need to have the user set them.
***************************************************************************/
PID::PID(double* Input, double* Output, double* Setpoint,
        double Kp, double Ki, double Kd, int ControllerDirection)
{

    myOutput = Output;
    myInput = Input;
    mySetpoint = Setpoint;
    inAuto = false;

    PID::SetOutputLimits(0, 255);    //default output limit corresponds to
                                     //the arduino pwm limits

    SampleTime = 100;      //default Controller Sample Time is 0.1 seconds

    PID::SetControllerDirection(ControllerDirection);
    PID::SetTunings(Kp, Ki, Kd);

    lastTime = millis()-SampleTime;
}


/* Compute()
**********************************************************************
 *     This, as they say, is where the magic happens.  this function
 *     should be called every time "void loop()" executes.  the function
```

```
*    will decide for itself whether a new pid Output needs to be computed.
*    returns true when the output is computed, false when nothing has been
*    done.
**********************************************************************/
bool PID::Compute()
{
   if(!inAuto) return false;
   unsigned long now = millis();
   unsigned long timeChange = (now - lastTime);
   if(timeChange>=SampleTime)
   {
      /*Compute all the working error variables*/
      double input = *myInput;
      double error = *mySetpoint - input;
      ITerm+= (ki * error);
      if(ITerm > outMax) ITerm= outMax;
      else if(ITerm < outMin) ITerm= outMin;
      double dInput = (input - lastInput);

      /*Compute PID Output*/
      double output = kp * error + ITerm- kd * dInput;

      if(output > outMax) output = outMax;
      else if(output < outMin) output = outMin;
      *myOutput = output;

      /*Remember some variables for next time*/
      lastInput = input;
      lastTime = now;
      return true;
   }
   else return false;
}


/*SetTunings(...)*****************************************************
 * This function allows the controller's dynamic performance to be
 * adjusted. it's called automatically from the constructor, but tunings
 * can also be adjusted on the fly during normal operation
**********************************************************************/
void PID::SetTunings(double Kp, double Ki, double Kd)
{
   if (Kp<0 || Ki<0 || Kd<0) return;

   dispKp = Kp; dispKi = Ki; dispKd = Kd;

   double SampleTimeInSec = ((double)SampleTime)/1000;
   kp = Kp;
   ki = Ki * SampleTimeInSec;
   kd = Kd / SampleTimeInSec;

  if(controllerDirection ==REVERSE)
   {
      kp = (0 - kp);
      ki = (0 - ki);
      kd = (0 - kd);
   }
```

```
}

/*SetSampleTime(...)****************************************************
 * sets the period, in Milliseconds, at which the calculation is performed
*************************************************************************/
void PID::SetSampleTime(int NewSampleTime)
{
   if (NewSampleTime > 0)
   {
      double ratio  = (double)NewSampleTime
                      / (double)SampleTime;
      ki *= ratio;
      kd /= ratio;
      SampleTime = (unsigned long)NewSampleTime;
   }
}

/*
SetOutputLimits(...)***************************************************
 *     This function will be used far more often than SetInputLimits.
 *     while the input to the controller will generally be in the 0-1023
 *     range (which is the default already,)  the output will be a little
 *     different.  maybe they'll be doing a time window and will need 0-
 *     8000 or something.  or maybe they'll want to clamp it from 0-125.
 *     who knows.  at any rate, that can all be done here.
*************************************************************************/
void PID::SetOutputLimits(double Min, double Max)
{
   if(Min >= Max) return;
   outMin = Min;
   outMax = Max;

   if(inAuto)
   {
        if(*myOutput > outMax) *myOutput = outMax;
        else if(*myOutput < outMin) *myOutput = outMin;

        if(ITerm > outMax) ITerm= outMax;
        else if(ITerm < outMin) ITerm= outMin;
   }
}

/*SetMode(...)**********************************************************
 * Allows the controller Mode to be set to manual (0) or Automatic (non-
 * zero) when the transition from manual to auto occurs, the controller is
 * automatically initialized
*************************************************************************/
void PID::SetMode(int Mode)
{
    bool newAuto = (Mode == AUTOMATIC);
    if(newAuto == !inAuto)
    {  /*we just went from manual to auto*/
        PID::Initialize();
    }
    inAuto = newAuto;
}
```

```
/*Initialize()****************************************************************
 *  does all the things that need to happen to ensure a bumpless transfer
 *  from manual to automatic mode.
 ******************************************************************************/
void PID::Initialize()
{
   ITerm = *myOutput;
   lastInput = *myInput;
   if(ITerm > outMax) ITerm = outMax;
   else if(ITerm < outMin) ITerm = outMin;
}

/*SetControllerDirection(...)*************************************************
 * The PID will either be connected to a DIRECT acting process (+Output
 * leads to +Input) or a REVERSE acting process(+Output leads to -Input.)
 * we need to know which one, because otherwise we may increase the output
 * when we should be decreasing.  This is called from the constructor.
 ******************************************************************************/
void PID::SetControllerDirection(int Direction)
{
   if(inAuto && Direction !=controllerDirection)
   {
      kp = (0 - kp);
      ki = (0 - ki);
      kd = (0 - kd);
   }
   controllerDirection = Direction;
}

/* Status Functions*************************************************************
 * Just because you set the Kp=-1 doesn't mean it actually happened.
 * these functions query the internal state of the PID.  they're here for
 * display purposes.  this are the functions the PID Front-end uses for
 * example
 ******************************************************************************/
double PID::GetKp(){ return  dispKp; }
double PID::GetKi(){ return  dispKi;}
double PID::GetKd(){ return  dispKd;}
int PID::GetMode(){ return  inAuto ? AUTOMATIC : MANUAL;}
int PID::GetDirection(){ return controllerDirection;}
```

EEPROM.h:

```
/*
  EEPROM.h - EEPROM library
  Copyright (c) 2006 David A. Mellis.  All right reserved.

  This library is free software; you can redistribute it and/or
  modify it under the terms of the GNU Lesser General Public
  License as published by the Free Software Foundation; either
  version 2.1 of the License, or (at your option) any later version.

  This library is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
  Lesser General Public License for more details.
```

```
  You should have received a copy of the GNU Lesser General Public
  License along with this library; if not, write to the Free Software
  Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301
  USA
*/

#ifndef EEPROM_h
#define EEPROM_h

#include <inttypes.h>

class EEPROMClass
{
  public:
    uint8_t read(int);
    void write(int, uint8_t);
};

extern EEPROMClass EEPROM;

#endif
```

EEPROM.cpp:

```
/*
  EEPROM.cpp - EEPROM library
  Copyright (c) 2006 David A. Mellis.  All right reserved.

  This library is free software; you can redistribute it and/or
  modify it under the terms of the GNU Lesser General Public
  License as published by the Free Software Foundation; either
  version 2.1 of the License, or (at your option) any later version.

  This library is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
  Lesser General Public License for more details.

  You should have received a copy of the GNU Lesser General Public
  License along with this library; if not, write to the Free Software
  Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301
  USA
*/

/******************************************************************************
 * Includes
 ******************************************************************************/

#include <avr/eeprom.h>
#include "Arduino.h"
#include "EEPROM.h"

/******************************************************************************
 * Definitions
 ******************************************************************************/
```

```
/*************************************************************************
 * Constructors
 *************************************************************************/

/*************************************************************************
 * User API
 *************************************************************************/

uint8_t EEPROMClass::read(int address)
{
    return eeprom_read_byte((unsigned char *) address);
}

void EEPROMClass::write(int address, uint8_t value)
{
    eeprom_write_byte((unsigned char *) address, value);
}

EEPROMClass EEPROM;
```