THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF MECHANICAL ENGINEERING

# Design of Arduino Instrumentation for a Thermal Management Testbed

JONAH GLUNT
SPRING 2022

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Mechanical Engineering
with honors in Mechanical Engineering

Reviewed and approved* by the following:

Dr. Herschel Pangborn
Assistant Professor of Mechanical Engineering
Thesis Supervisor

Dr. Daniel Cortes
Assistant Professor of Mechanical Engineering
Honors Adviser

* Electronic approvals are on file.

# ABSTRACT

With modern innovations pushing toward faster and more powerful electric technology, increased efficiency in thermal management systems is required. Advanced control algorithms are being developed which are capable of operating these thermal systems at higher efficiency. While simulations are a powerful tool to test a new control approach, it is also necessary to verify these control algorithms experimentally with a physical thermal management setup.

For most lab spaces, however, it is not feasible to construct a full-scale thermal load and management system (a complete hybrid-electric vehicle, for example) to test the control algorithm. It is useful, therefore, to have a smaller scale setup of a thermal management system, often called a testbed. Such a testbed requires a data acquisition system capable of reading many different sensors at once and communicating with the controller. The standard data acquisition systems typically cost thousands of dollars and are designed to update at thousands of times per second, which is not necessary or cost-effective for a single-phase cooling loop.

This work aims to detail the design of a new data acquisition system, which runs entirely on a collection of Arduino microcontrollers. The Arduinos are in communication with a desktop computer running a control algorithm in Simulink. Various Arduino shields were designed to allow each Arduino to handle a specific task (such as measuring temperature sensors). Some of the low-level control has been delegated to the Arduino as well; for example, the Arduino in charge of measuring flow rate also uses proportional-integral (PI) control to match a desired flow rate refence by actuating the signal to a pump. This new testbed and data acquisition system will allow more complex and powerful control algorithms in Simulink to be physically verified on a single-phase thermal management loop.

# TABLE OF CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

# ACKNOWLEDGEMENTS

First and foremost, I am forever grateful for the love and support of my family. Thank you to my parents for their encouragement throughout all the years, and to my siblings for being my best friends through it all.

Next, I wish to sincerely thank Dr. Herschel Pangborn for the opportunity to learn and research within the PAC lab; without his support and mentorship, this thesis would not have been anywhere near possible. (Not to mention, his patience in navigating alongside my hectic schedule of music rehearsals and performances!). Thank you for guiding my path and modeling what impactful, integrous research looks like. I also wish to extend a special thanks to my lab-mates Jason Lord and Ian Rivera for their roles in this project. Jason started this project with Dr. Pangborn in Spring 2020 before it was put on abrupt hold due to the pandemic. He cataloged and ordered many of the parts for the testbed and laid the foundation for the physical construction. Ian aided in designing and printing many of the 3D-printed mounts. I am very excited to see the many ways in which this testbed will be used and improved in the future!

To all the friends I have made at Penn State, thank you for your never-ending encouragement and the fond memories. A special shout-out to the School of Music faculty and family of students who have helped me to grow as a person and musician over the last four years. I extend my gratitude to everyone who makes Penn State such a welcoming and rich community, and who has helped me to find my place and explore my passions.

# Chapter 1

# Literature Review

## Motivation for Building an Experimental Testbed

As consumer demands push electronic systems to become smaller yet more powerful, those systems generate more and more heat. Without proper thermal management, that heat can harm not only the electronics, but the user as well. For example, consumer demands require electric vehicles to have longer battery life and faster charge time, both of which will produce more heat in the system that needs to be managed [1]. So, to improve the efficiency and energy density while keeping the battery within its operating temperature limits, more research into thermal management systems is needed [2].

The need for higher efficiency promotes the use of various control algorithms to automate the cooling system, as opposed to operating the cooling system manually. One of the most promising advanced control strategies is model predictive control (MPC). When using MPC, the system evaluates its current parameters such as temperature, flow rate, and pressure, to predict what will happen in future timesteps; it then applies actuator commands to attempt to minimize future losses [3]. In the case of thermal management systems, this loss (sometimes referred to as cost) could be a measure of temperatures that are outside of the operating limits. MPC contrasts other control methods which minimize only the current losses without looking at how that affects future situations. MPC might cause the system to initially suffer a higher loss to minimize the total loss in the long-term [4].

For small-scale electrothermal systems like automotive and power electronic systems, it is common to use single-phase liquid cooling for thermal management. Single-phase cooling typically offers a higher heat capacity than air cooling for a given volume or mass [5]. Although two-phase cycles can have higher efficiency, the dynamics of phase-change heat transfer are much harder to model and control, hence the use of single-phase systems is promoted.

While modern simulation tools are very powerful for verifying complex control algorithms, it is also useful and necessary to experimentally verify the controllers on a physical system. Constructing a full power and thermal management system is both time-consuming and costly, a task unfeasible for most academic research. Therefore, it is useful to develop laboratory scale testbeds to study interactions between thermal components. Scaled testbeds are also safer for human use and study, while maintaining the same physical relationships and repeatability of the full thermal system [6].

**Previously Built Testbeds and Improvements to be Made**

Pangborn et. al [6] developed an experimental setup for a fuel thermal management system in 2017, designed to approximately 1/12 of a working aircraft. The testbed utilizes multiple heat loads at various frequencies to model different kinds of heat generated on an aircraft, and it uses de-ionized water as the working liquid. A drain pulls fluid out of the system to model the usage of fuel in an aircraft engine, and the system is cooled via a heat exchanger connected to a chiller. The data acquisition (DAQ) for this testbed was performed on National Instruments hardware running LabVIEW and MATLAB/Simulink. The testbed was used to experimentally verify a graph-based prediction model of an aircraft thermal system.

Pastor et al. [7] built a thermal testbed for the 2018 SoftCOM international conference. The testbed was used to examine thermal management systems for large computer servers in data centers. Their work showed that simulations often struggle to fully describe the relationship between electronics and a cooling system, and so they developed a testbed to measure cooling data from over 50 sensors to increase the accuracy of future simulations.

While these previous testbeds have led to great advancements in understanding of thermal management systems, there are multiple improvements that can be made. One such improvement is to replace their fixed architecture with a modular design. The previously mentioned testbeds are said to be "fixed" because the thermal components are rigidly attached in place. A modular design would allow the components to be removed and re-positioned to test a wide variety of thermal setups (for example, placing heat loads in parallel vs. series).

Huang et. al [8] developed a semi-modular testbed to verify their studies in aircraft thermal management via dimensional analysis. Their testbed can be arranged in multiple experimental configurations to match different cooling system layouts. In 2019, the Penn State Energy System Optimization and Control Lab built an elaborate experimental setup for a district heating network [9]. They developed control tools to improve the efficiency of heat and power networks in residential and industrial areas. This setup is also modular and can be re-organized to experiment with varying the number of buildings connected to a residential heating network. The setup consists of a boiler, heat loads representing buildings, and pipes to transport the water, and uses a National Instruments device as a DAQ to measure 32 thermocouples and 16 pressure transducers, and to operate four bypass valves.

Pangborn built another testbed at Illinois [10] which was modular in design and used to validate advanced control algorithms for vehicle energy management. For the DAQ, a National

Instruments CompactDAQ was used to operate sensors and actuators, and then communicate the data to MATLAB/Simulink. This testbed serves as a starting point for the testbed built for this thesis.

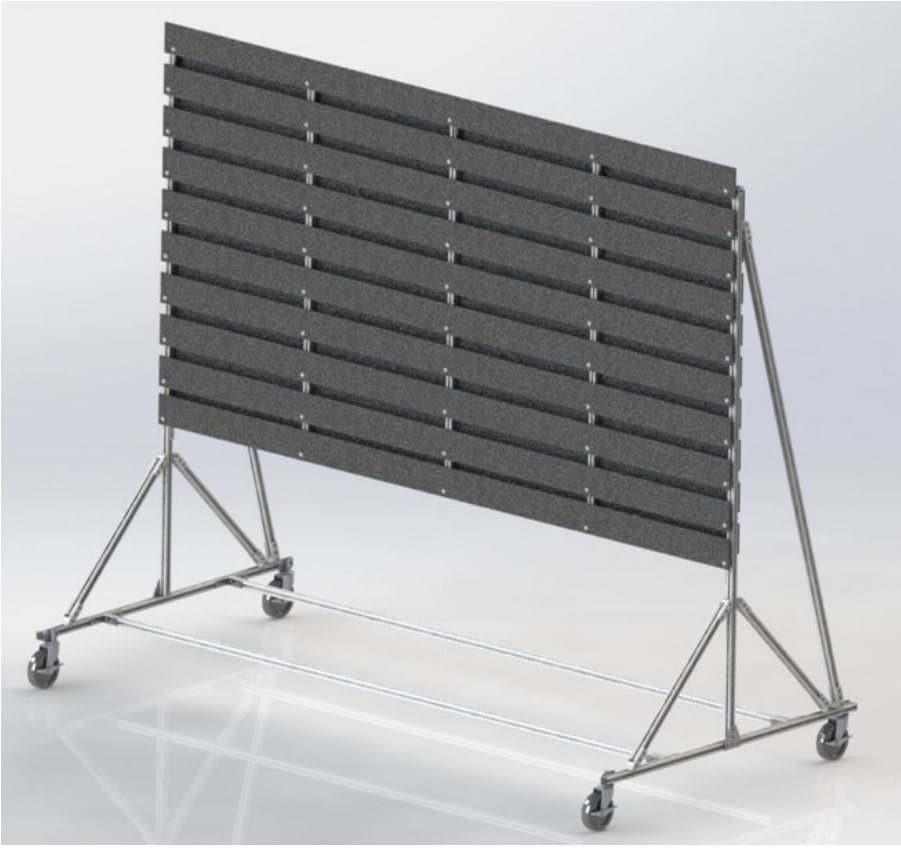

**Figure 1. Rendering of Pangborn's SOLIDWORKS model for Illinois testbed**

The data acquisition system (DAQ) for many of these experimental setups come from companies such as National Instruments or dSPACE. These high-end DAQs cost thousands of dollars and are designed to take measurements at thousands of times per second. However, in a single-phase thermal management system, where measurables such as temperature and flow rates are not changing so quickly, a sampling rate of around once per second would be adequate to capture the performance of the system. Therefore, resources invested in high-end DAQs could be

better put to use elsewhere in the design. One goal of this research project is to determine if an Arduino can work sufficiently as the DAQ for a thermal testbed. An Arduino Mega costs only around $45 and can easily take many measurements at a rate of once per second, so the possibility of replacing an expensive DAQ with a much cheaper Arduino is worth exploring.

**Figure 2. CompactDAQ as sold by National Instruments ($8000)** [11]


**Figure 3. MicroLabBox sold by dSPACE ($7500-$13000)** [12]

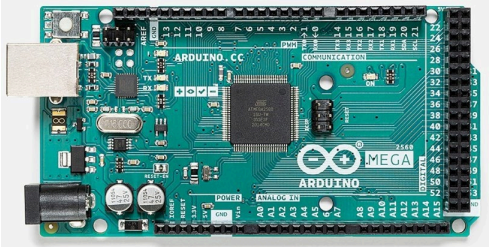


**Figure 4. Arduino Mega ($45)** [13]

# Chapter 2

# Construction of Testbed

## Construction of Testbed Stand

The physical stand of the testbed built for this thesis is fabricated to nearly match that constructed at Illinois by Pangborn et. al. The frame of the stand is made from 80/20 T-slot aluminum, which is structurally strong and easy to connect. On the front of the stand are 10 ABS plastic slats, onto which the thermal components can be hung. The slats are attached to aluminum U-channel and then secured to the mainframe via 80/20 T-slot nuts. A small spacer is also inserted into the U-channel so that the metal maintains shape and is not crushed when tightening the screw (Figure 5).

**Table 1. Bill of materials for testbed stand**

| Item | Quantity | Unit Price | Subtotal Price |
|---|---|---|---|
| T-slot 80/20 8 ft | 7 | $40.66 | $284.62 |
| T-slot 80/20 6 ft | 2 | $31.28 | $62.56 |
| T-slot 80/20 4ft | 5 | $22.87 | $114.35 |
| T-slot 80/20 2ft | 4 | $10.72 | $42.88 |
| 80/20 corner connector | 2 | $8.41 | $16.82 |
| 80/20 large T bracket | 2 | $10.70 | $21.40 |
| 80/20 90º bracket | 26 | $7.07 | $183.82 |
| 80/20 end cap | 4 | $1.80 | $7.20 |
| Aluminum U-Channel 1/2"x1"x8' | 12 | $21.85 | $262.20 |

| | | | |
|---|---|---|---|
| Panel fastener 1/4"-20, 1” (50 pack) | 2 | $10.90 | $21.80 |
| Panel spacer (100 pack) | 1 | $13.51 | $13.51 |
| High bond transfer tape (60 yards) | 1 | $54.32 | $54.32 |
| Drill bit, size 43 for 4-40 Tap | 1 | $25.89 | $25.89 |
| Smooth Laminating Roller 1" diameter, 4" wide | 1 | $12.94 | $12.94 |
| Friction-grip stem swivel casters | 4 | $38.88 | $155.52 |
| Caster fastener brackets (4 pack) | 1 | $10.97 | $10.97 |
| 1/4” tin-coated drill bit | 1 | $6.22 | $6.22 |
| T-slot framing nut single fastener (25 pack) | 4 | $5.99 | $23.96 |
| T-slot framing nut double fastener (10 pack) | 7 | $10.06 | $70.32 |
| Pivot arm assembly | 12 | $24.74 | $296.88 |
| Button head hex screw, 1/4"-20, 1/2" (50 pack) | 2 | $8.44 | $16.88 |
| ABS Plastic Sheet, 0.25” thick, 96” x 4” | 20 | $14.34 | $287.76 |
| | | **TOTAL** | **$1,992.92** |

**Figure 5. Assembly of plastic slats onto 80/20 frame**

Some minor alterations were made from Pangborn's Illinois design. The depth of the testbed has been reduced; this iteration is only two feet deep whereas the Illinois testbed was four feet deep. This change was made to allow easier transportation of the testbed while still maintaining vertical stability. The legs of the testbed were also cut shorter so that the total height is 6 ft 4 in, allowing the testbed to fit out of the door of the lab space where it was built (which is 6 ft 9 in). Lastly, the spacing of the plastic slats was adjusted so that they are all equidistant from one another, as opposed to the smaller gap between the bottom two slats of the Illinois testbed.

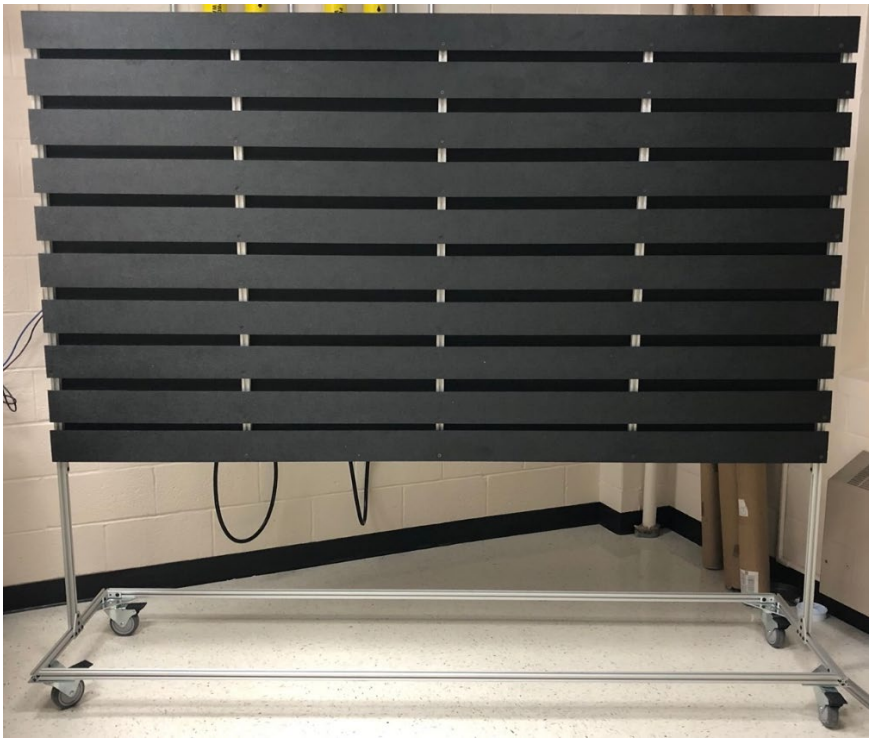

**Figure 6. Construction of testbed frame (before adding back supports and respacing plastic slats)**

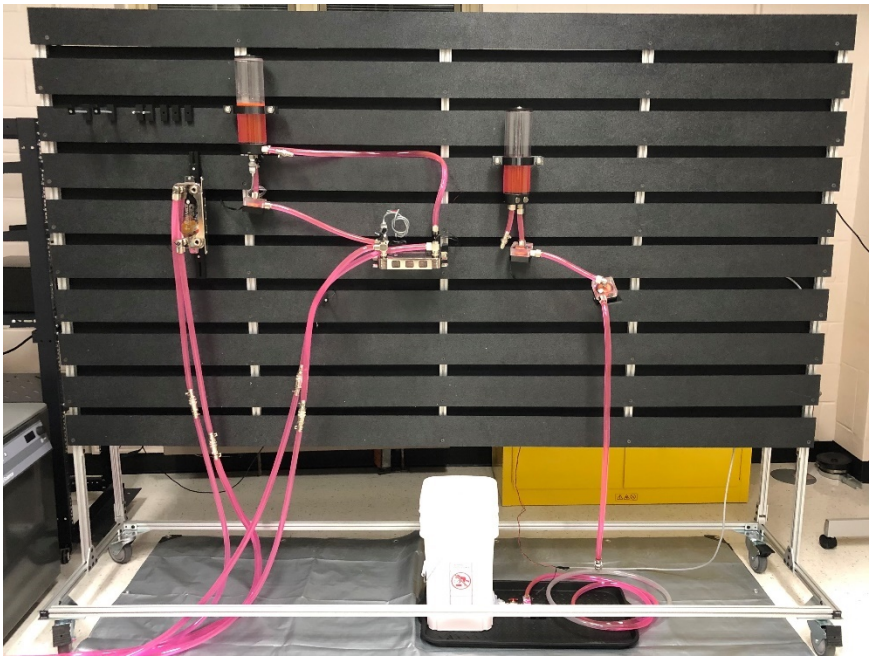

**Figure 7. Completed testbed with various components attached**

**Figure 8. Server rack behind testbed to house computer and eventually all circuitry**

## Modularity and 3D-Printed Mounts

For each of the thermal components to be hung on the testbed, a 3D-printed mount was designed by another student in the lab. Some of the mounts are custom made to match the geometry of the specific part, and others are generic flat plates that have a Velcro surface onto which components can be affixed. These mounts simply hook over the plastic slats on the testbed. Because of this, it is very simple to lift a component out of its current location and place it anywhere else on the testbed, facilitating modularity. Figure 9 shows an example of a small thermal system where fluid (commercial antifreeze) is pumped through a heat exchanger and

reservoir. The 3D-printed mounts for holding reservoir tanks and heat exchangers can be seen, as well as sensors for temperature, flow rate, and pressure.

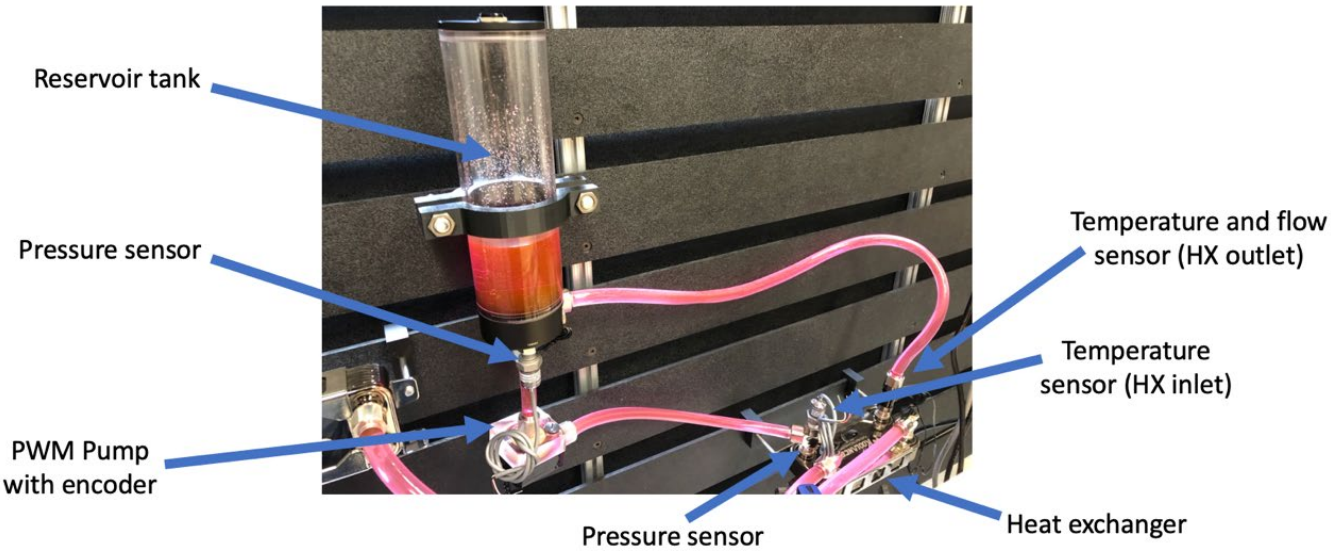


**Figure 9. Sample thermal system on testbed (components labeled)**

**Table 2. List of sensors and components used with testbed**

| **Component** | **Manufacturer and Part Number** | **Specifications** |
|---|---|---|
| Temperature sensor | Koolance SEN-AP008B | Temp. range: -40 to 120 °C |
| Flowmeter | Koolance INS-FM17N | Temp. range: 0 to 70 °C<br>Flow range: 1.0 t 15 LPM<br>(0.017 to 0.25 L/s) |
| Flowmeter and temperature sensor | Koolance SEN-FM18T10 | Temp. range: 0 to 70 °C<br>Flow range: 2.0 to 15 LPM<br>(0.033 to 0.25 L/s) |
| Chiller | DuraChill Air- and Water-Cooled, 1.5 HP | Temp range: 5 to 25 °C<br>Max. pressure: 6.9 bar |

| | | |
|---|---|---|
| | | Max. flow: 13.2 L/min |
| Pump | Lelukee 082201, from Amazon.com | Max. head: 3m<br>Max. flow: 500 L/hr<br>(0.139 L/s) |
| Heat exchanger | Koolance HXP-193 | Temp. range: -100 to 200 °C<br>Num. plates: 12<br>Max. pressure: 284.5 psi<br>(19.6 bar) |
| Reservoir | Koolance BDY-TK240X70 | Capacity: 924 mL<br>Max. pressure: 28.5 psi (1.97 bar)<br>Max. temp: 80 °C |

# Chapter 3

# Design of Testbed Instrumentation

## Use of Fritzing to Develop Custom PCBs

In Pangborn's previous testbed, the instrumentation circuitry was wired manually on a prototyping breadboard, which caused multiple issues. Firstly, as the pumps run off 12 volts at approximately 1 amp, powering several pumps with a single breadboard was found to exceed its power rating and cause it to fail. Secondly, when there are many sensors in use, the breadboarding becomes complicated very quickly and makes the circuitry harder to work with, more time consuming to modify and debug, and therefore more error prone. Lastly, adding Arduinos onto already complicated breadboards would be difficult. A more robust and user-friendly instrumentation design is needed.

For this project, printed circuit board (PCB) shields for various sensor types were designed. These shields sit directly on top of an Arduino Mega, plugging into all of its pins. On top of the shields are male MOLEX pins into which sensors can be directly plugged. MOLEX was chosen as the universal connector because it is already the factory connector on the flow sensors, and it supplies a mechanically strong yet simple two-pin connection. To design the PCB shields, the program Fritzing was used. Fritzing allows the user to design a circuit using a virtual breadboard and Arduino, and then automatically translates the connections to a PCB layout. However, the user must arrange the components on the PCB in such a way that all of the traces are possible, which can take some patience and iteration. The finished PCB designs can be ordered online from a PCB manufacturer, along with the necessary circuit components to be soldered to the boards by hand.

**Temperature Sensor Shield**

The temperature sensor that was used is a thermistor, which acts as a resistor whose resistance varies with temperature. To determine the temperature to which the sensor is subjected, its resistance must first be measured and then the temperature can be calculated from the calibration data. The resistance can most easily be measured using a voltage divider circuit as shown in Figure 10, where the resistor in question is placed in series with a resistor of known value and the voltage drop is measured. By applying Ohm's law for both resistors and the fact that resistors in series have the same current, it can be shown that:

$$V_{thermocouple} = V_{in} * \frac{R_{thermocouple}}{R_{thermocouple} + R_{known}} = 2.05\ V * \frac{R_{thermocouple}}{R_{thermocouple} + 10\ k\Omega} \quad (1)$$

$$R_{thermocouple} = 10\ k\Omega * \frac{V_{thermocouple}}{2.05\ V - V_{thermocouple}} \quad (2)$$

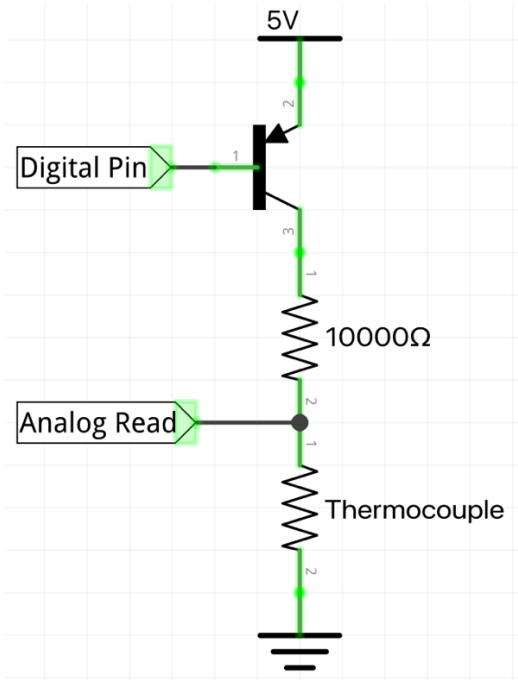


**Figure 10. Basic circuit diagram for temperature measurement**

In the PCB shield, a 10 kΩ resistor was used as the known resistor because that value corresponds to 25 ºC on the calibration data (see Appendix C), which falls centrally in the likely temperature range of fluids on the testbed. The circuit also includes a MOSFET transistor, which is switched on or off via a digital pin on the Arduino. This can be used to simulate a sensor fault. When the completed testbed is used to evaluate candidate control algorithms, this will make it possible to test how the controller responds to a sensor fault, i.e., whether it recognizes the outlier temperature as an error and acts accordingly. However, the MOSFET causes a significant voltage-drop measured to be 2.95 V on average, so in Equation 1, 2.05 V is used for $V_{in}$ instead of 5 V. Additionally, to increase the resolution of the measurements, the Arduino Mega is set to measure the voltage with an internal analog reference of 2.56 V. This allows the Arduino to allocate its 10-bit resolution to the range of 0-2.56 V (where all of the temperature measurements will lie) instead of the entire 0-5 V range, nearly doubling the resolution. An Arduino Mega has 16 analog read pins, so the completed shield was designed with 16 copies of the above circuit.

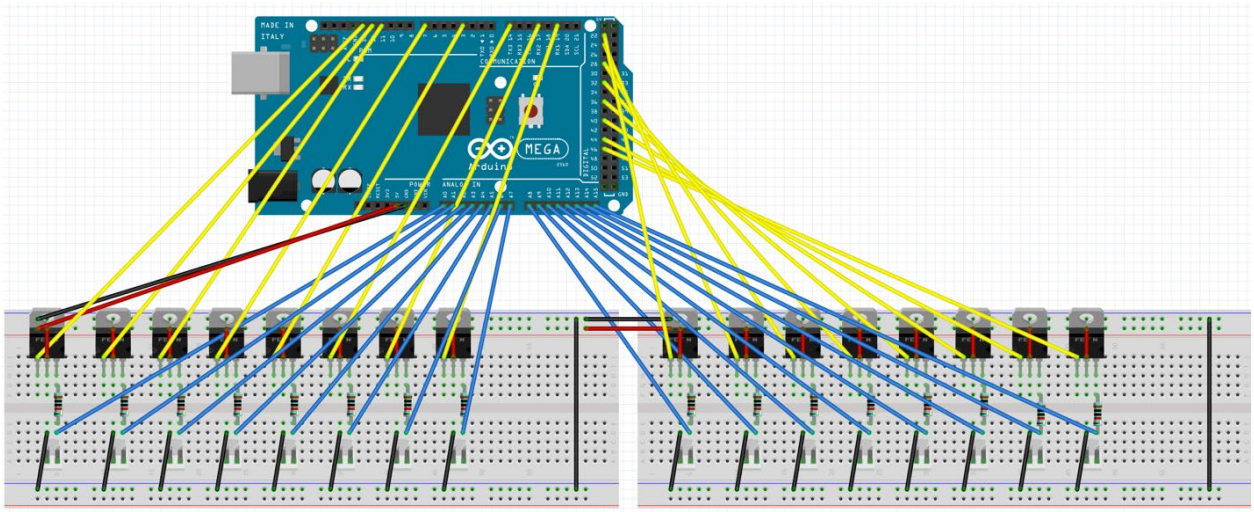


**Figure 11. Temperature circuit on Fritzing breadboard layout**

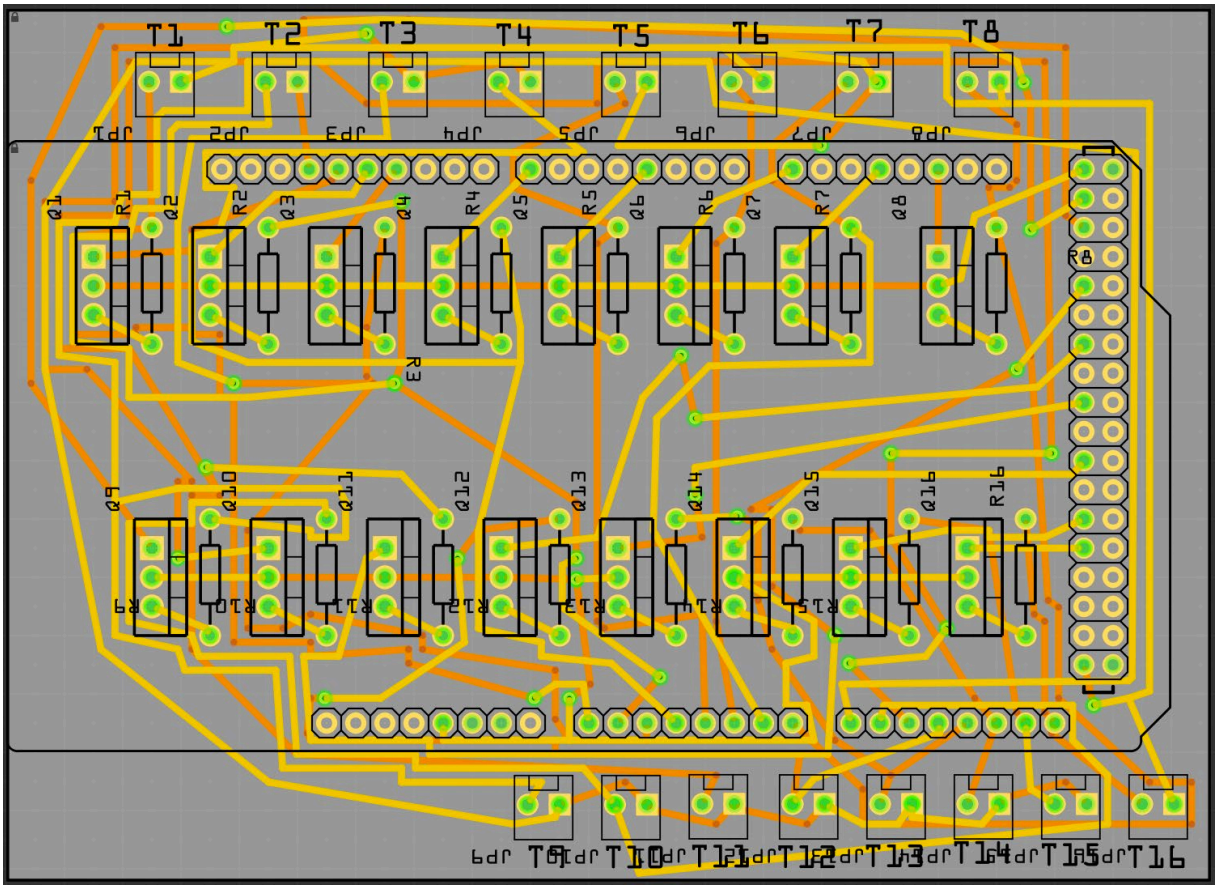


**Figure 12. Temperature circuit on Fritzing PCB layout**

The temperature sensor manufacturer provides a chart with resistance values for every 1ºC. Regression was performed with this data in MATLAB to obtain the equation (code found in Appendix B):

$$Temperature\ ºC = 256.7 * (Resistance\ k\Omega)^{-0.1471} - 157.9 \qquad (3)$$

While the resolution of the calibrated measurements is highly nonlinear (due to the power regression used), from experimentation in the typical operating range of 10-30 °C, the quantized difference between measurements is approximately 0.1 ºC. This value can be taken as the assumed resolution, which is adequate for a single-phase cooling loop.

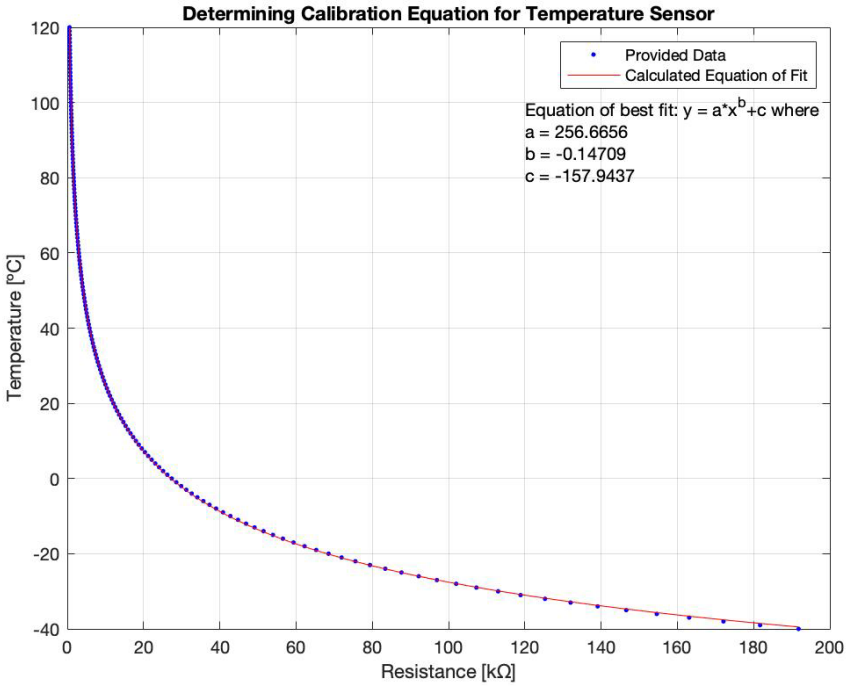


**Figure 13. Determining calibration equation for temperature sensor**

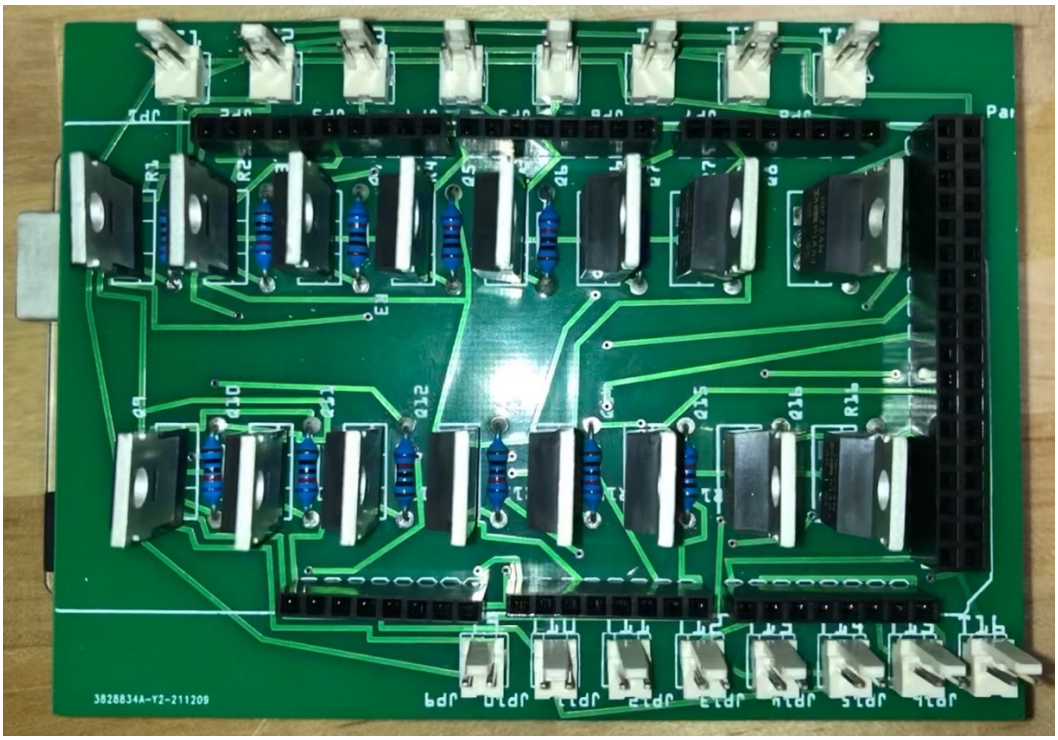

**Figure 14. Image of completed temperature sensor shield**

**Flow Rate Sensor and Pump Shield**

The flow rate and pump board connects to sensors to measure the flow rate and delivers PWM signals to control the speeds of the pumps. These functions are included on the same board so that control of the pumps to track a desired flow rate reference can be performed on the Arduino itself with a fast update rate, rather than needing to be included in a slower-updating Simulink controller running on a desktop computer to produce flow rate references. Controlling pumps via PWM signals is straightforward thanks to the Arduino's built-in PWM capability.

There are two different scenarios used to measure flow rates from the testbed, and they result in different instrumentation schemes. The pumps have their own internal encoder, which outputs a wave signal that can be read directly by an Arduino interrupt pin. Secondly, there are external flow rate sensors, which operate differently. Instead of outputting a measurable wave, they contain a magnetic reed switch that opens and closes as fluid rotates the blades. To measure the speed of rotation (and then convert to a flow rate), a small current must be passed through the sensor and then the output can be connected to an interrupt pin.

Measuring Flow from Pump Output

Measuring flow rate directly from the pump output is fairly straightforward, as the pump outputs a wave signal that can be directly read by the Arduino Mega. The duty cycle of the wave is proportional to the flow rate of the fluid. The pump output is first passed through a MOSFET (to allow for on/off control of the sensor and simulation of a fault), then connected to one of the six interrupt pins on the Arduino Mega. The Arduino Mega has six interruptible pins (pins 2, 3, 18, 19, 20, 21), so a total of six of these circuits can be operated on one shield. For that reason,

the board has connections for six pump PWM signals, although the Arduino Mega has 12 PWM pins (two of them are also interrupt pins however).

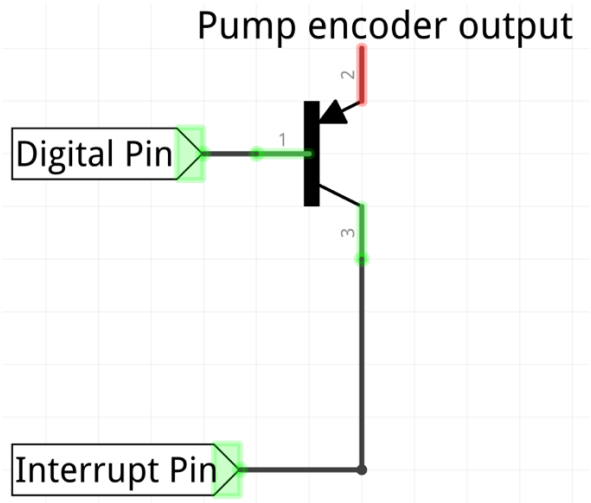


**Figure 15. Basic circuit diagram for pump encoder measurement**

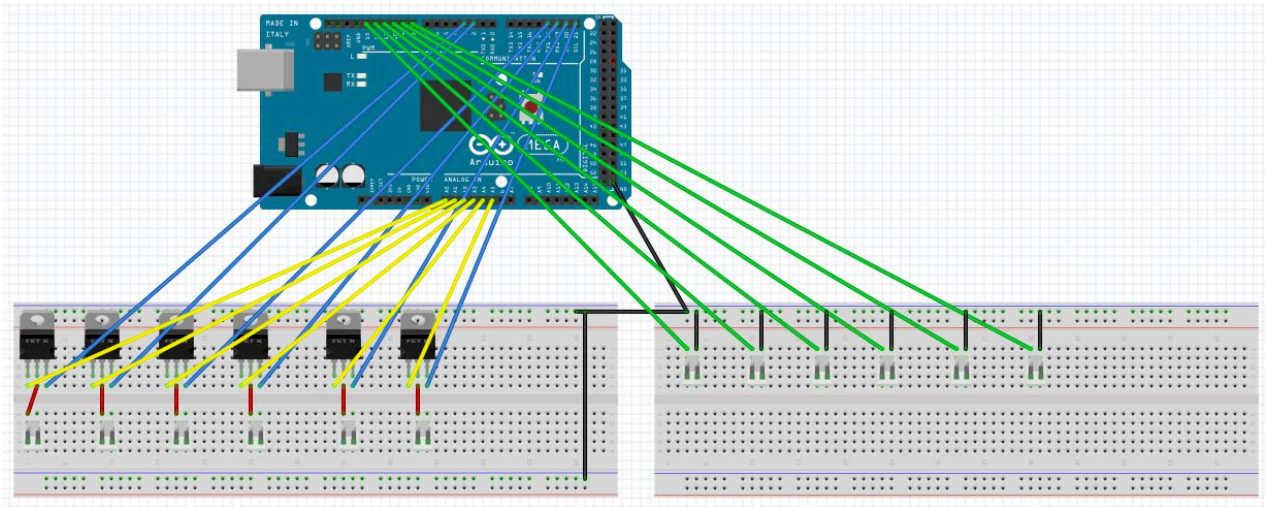

**Figure 16. Pump encoder and PWM control circuit on Fritzing breadboard layout**

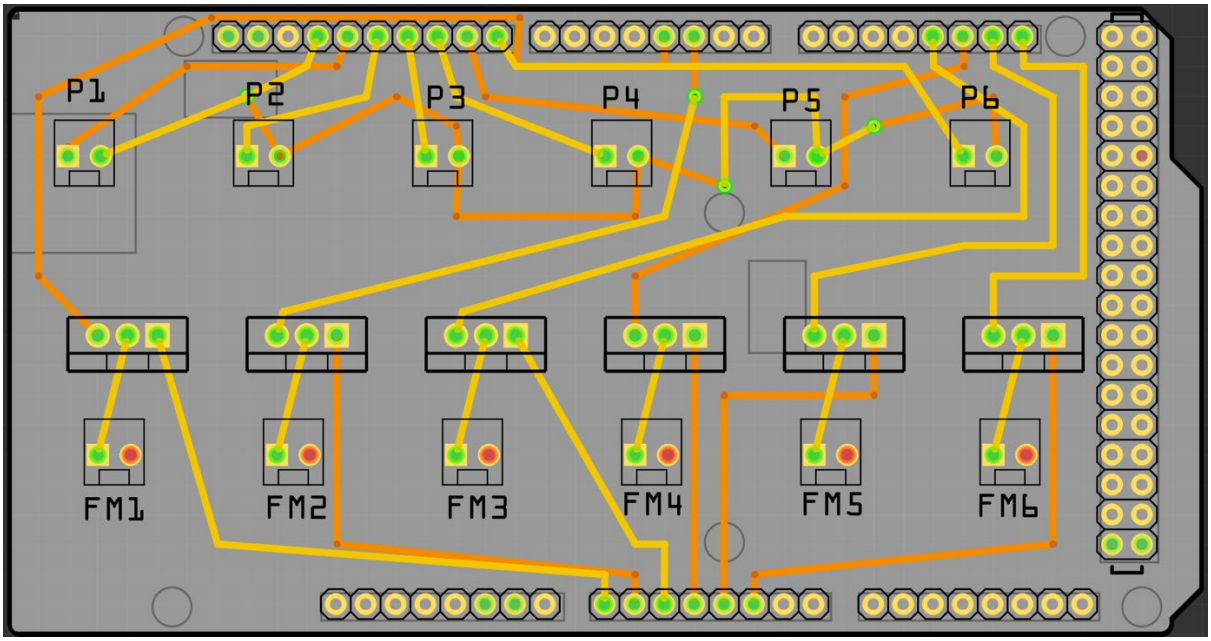


**Figure 17. Pump encoder and PWM control circuit on Fritzing PCB layout**

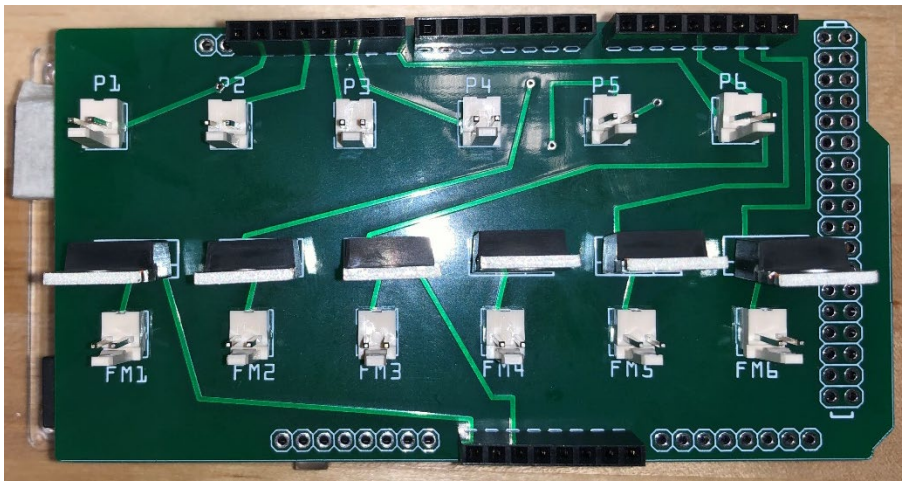


**Figure 18. Image of completed flow sensor and pump shield for pump encoder**

The Arduino measures the time that the signal is HIGH and LOW. By adding these two times, the period is obtained, then the positive duty cycle is calculated by dividing the time spent HIGH by the period. The pump manufacturer unfortunately does not provide a calibration from

the duty cycle to a volumetric flow rate, so that was determined experimentally through comparison with another flowmeter. It was found that for flow rates above 0.025 L/s, the duty cycle of the pumps output is approximately ten times the flow rate (see Experimentally Verifying Flowmeter Readings for more details).

## Measuring Flow from External Flowmeter

Measuring flow rate from the separate flowmeter sensors is slightly more complex than from the pump encoder itself. To measure when the encoder switch closes, the sensor's input is connected to the Arduino 5 V (after passing through a MOSFET to again allow simulation of sensor failure) and the output is connected to ground through a current limiting resistor, as laid out in Figure 19. One of the Arduino interruptible pins reads the voltage of the output of the switch. When the switch is open, no current flows through the circuit, so the interrupt pin always reads LOW. Once the switch is closed, current flows and the interrupt pin reads HIGH.

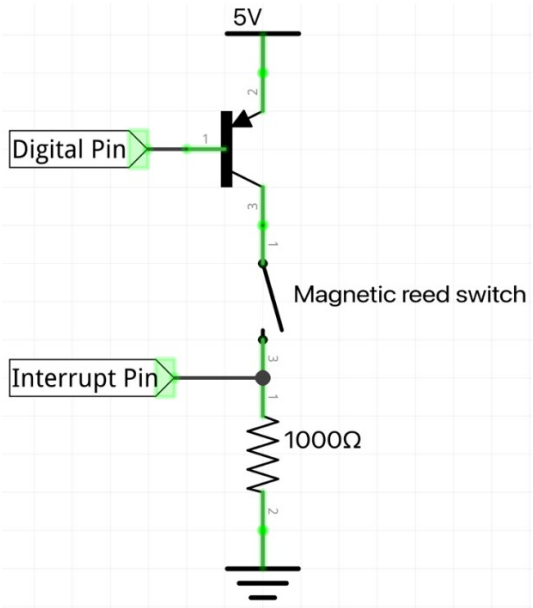


**Figure 19. Basic circuit diagram for measurement of external flowmeter**

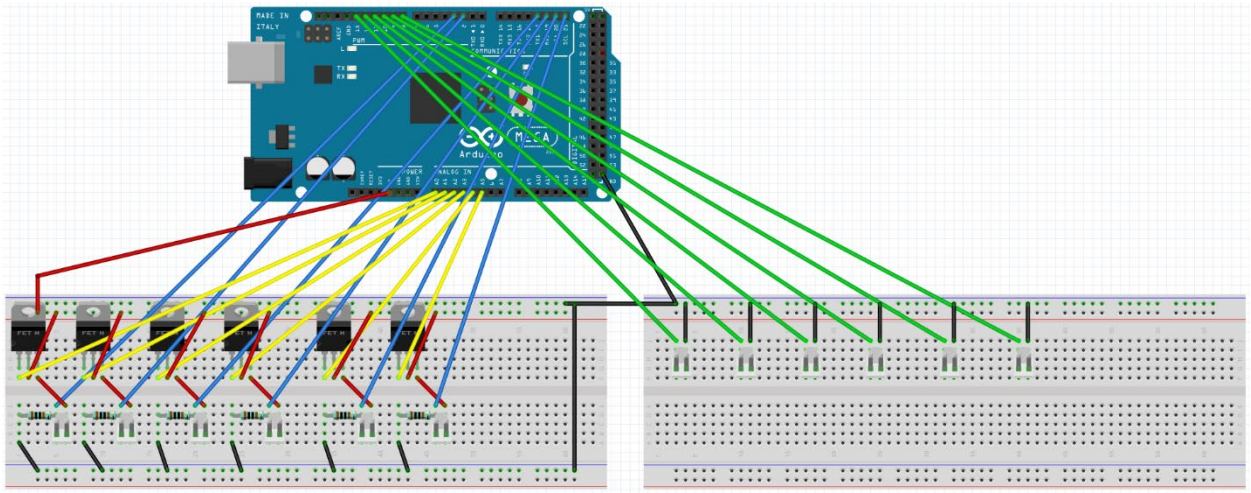

Figure 20. Flowmeter and pump circuit on Fritzing breadboard layout

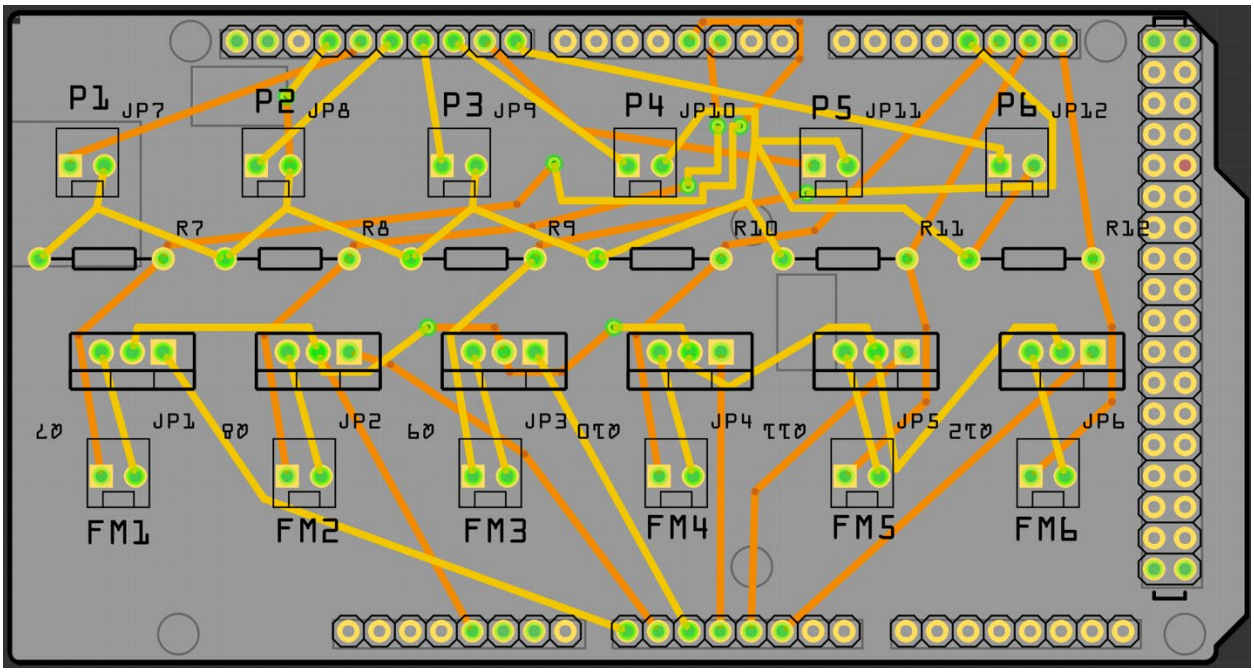


Figure 21. Flowmeter and pump circuit on Fritzing PCB layout

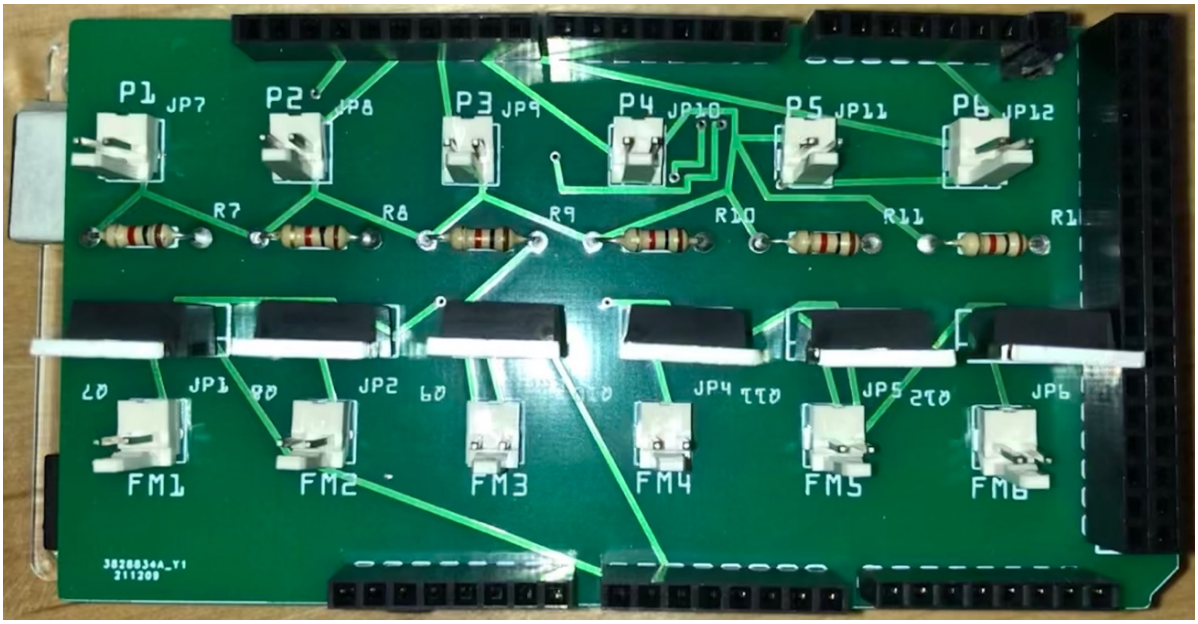


**Figure 22. Image of completed flow sensor and pump shield for external flowmeter**

The changing of the state of the switch triggers an interrupt function on the Arduino, which it adds as one count of the encoder. By dividing the number of counts over a small time period, the Arduino calculates the speed of the encoder in counts per second (Hz). However, because the interrupt was triggered on both rising and falling edges, the frequency must be divided by two to find the true frequency of rotation of the encoder. The manufacturer provided calibration data which is used to convert the frequency to a volumetric flow rate in liters per second (see Appendix C). Note that the two flowmeters used in this testbed have slightly different calibration coefficients.

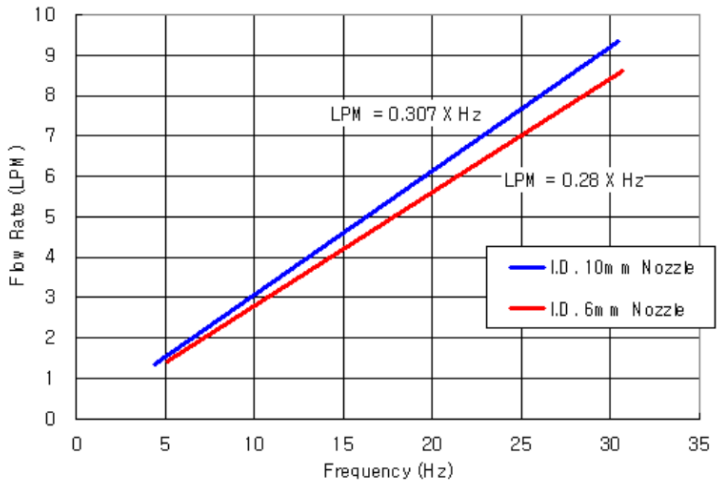


**Figure 23. Calibration curve provided for Koolance sensor INS-FM17N (individual flowmeter)** [14]

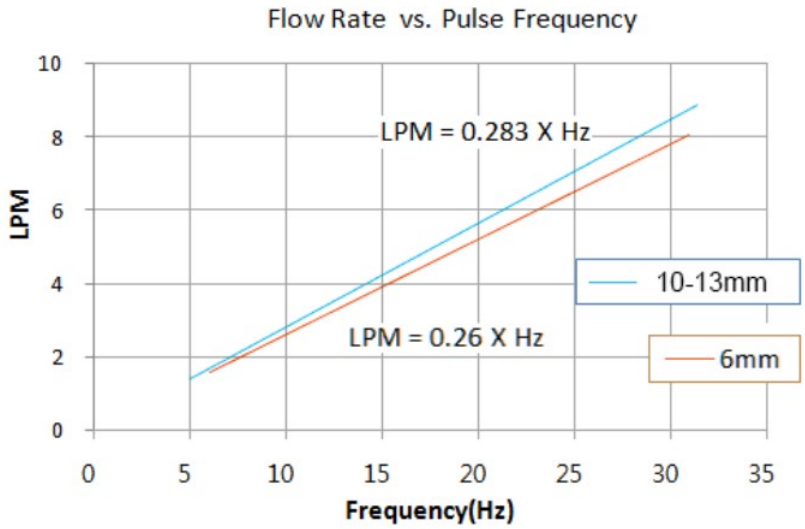


**Figure 24. Calibration curve provided for Koolance SEN-FM18T10 (flowmeter and temperature sensor)** [15]

# Chapter 4

## Serial Communication of Temperature Readings

### Overall Communication Scheme

For communication between Simulink and an Arduino measuring temperatures, there are two important sets of information that need to be sent back and forth: the on/off state of each MOSFET and the temperature readings from the sensors. The Arduino is constantly measuring and storing the temperature reading of each sensor. Every update period, Simulink sends a message to the Arduino with desired on/off state of each MOSFET. When the Arduino receives that message, it updates its own stored version of the on/off states to match what Simulink sent and returns back what it now has stored as the states (as a way for Simulink to check that the message was properly received). Then the Arduino also sends back all the measured temperature values.

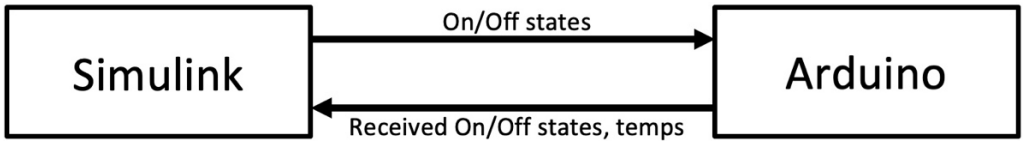


**Figure 25. Flowchart for serial communication of temperature data**

### Developing Arduino Code

As discussed in Chapter 3, the Arduino determines all of the temperatures by measuring the voltage drop across the sensor pins, converting the voltage drop to a resistance, and using the calibration curve to convert the resistance to a temperature value. The Arduino code for temperature communication (Temp_Shield) does this process for every sensor, every iteration of

the “loop()” function. The Arduino also has stored a 1x16 array of booleans as to whether or not the MOSFET activation pin should be turned on for a given sensor or not.

Each iteration of the “loop()” function, the Arduino also checks if there is any information on the serial port. If there is, then it reads all the information, and checks if it is in the expected format sent by the Simulink code. The expected message to receive starts and ends with the character ‘T’ (for temperature) and is 18 characters long. The middle 16 characters should be either a ‘1’ or a ‘0’, where ‘1’ indicates that the given MOSFET control pin should be turned on, and ‘0’ off. For example, if Simulink wanted to activate the MOSFETS for sensors 1 and 3 while leaving the others turned off, it would send the string ‘T1010000000000000T’. The Arduino updates its stored version of the on/off states, then sends that information and all 16 temperature readings back to Simulink, starting and ending with the character ‘t’ and inserting a comma between each character. For the above example, the Arduino would return the string ‘t,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0, […16 temperatures…], t’.

When sending serial information from the Arduino, the “Serial.print()” function is most useful because it can automatically convert nearly any data type to a string of characters, including the booleans and floats needing to be sent here, and send them across the serial stream in a readable format. If using the “Serial.write()” function, the data must first be converted to bytes of information to be sent, and then Simulink must convert those bytes back to characters upon receiving the message. For the last character to be sent in the message, the command “Serial.println()” is used so that a carriage return character is added to the end and that string of characters is officially ended.

**Developing MATLAB Code**

The Simulink communication uses a custom Simulink block running a MATLAB function (Temp_Meas_Fcn_1.m) written to match the communication method outlined above. The MATLAB function takes a 17x1 vector as the input, where the first value is the simulation time and the remaining 16 are the on/off states to be sent. On the first run of the function, a persistent variable for the serial communication is created. As of MATLAB R2021b, the best and most recent serial communication method is using a "serialport" object, which is created by specifying the port number and the baud rate (for example, "c = serialport('COM6', 9600)"). Using this method, it is very easy to write information across the serial line ("writeline(c, message)"), check if there is any information waiting to be read ("c.NumBytesAvailable"), and read from the line ("readline(c)").

After sending the on/off information to the Arduino (starting and ending the message with the character 'T'), MATLAB expects to receive a message back. It reads the message, separating it into an array via the commas in between values, and checks that it starts and ends with 't' and is the expected size (32 values). If all of those conditions are met, then it converts the array of characters into an array of doubles indicating the received on/off states and temperatures from the Arduino. If no message is received back, then it prints a message to the command line stating the error and sets the output to the default value -1 to represent an error in the communication.

There are some instances in which the Arduino does not send a float number as the temperature measurement. If a given MOSFET is turned on while there is no sensor connected to the pins (or the sensor connected is broken), then the Arduino reads that as a negative resistance and ends up sending "NaN" (not a number) across the serial port (because the calibration curve

attempted to take the root of a negative number). In the case where a sensor is connected but the MOSFET is turned off, then the Arduino reads that as a very large resistance and sometimes sends “inf” (infinite) across the communication line. Neither of these values, NaN or inf, work in Simulink, so the MATLAB code replaces them with the value -2. In the future when the testbed is used to validate a Simulink controller with fault design, if values of -1 or -2 are seen as a temperature output, or if the output is a very large outlier, the controller needs to be able to recognize these are not actual temperature readings but a representation of one of multiple different errors.

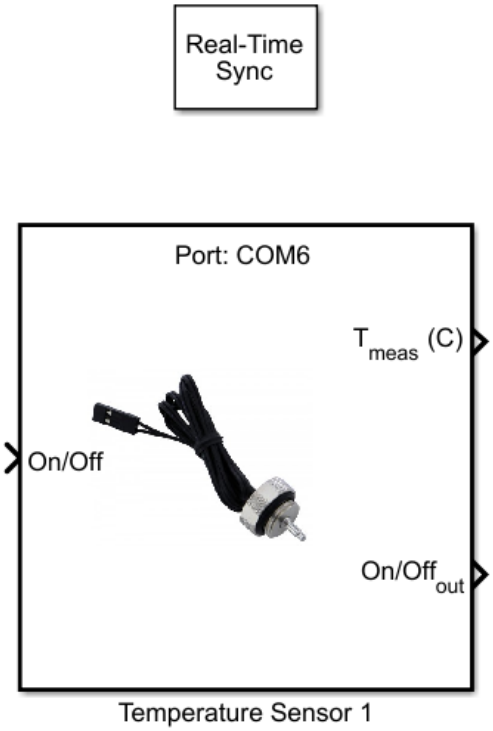


**Figure 26. Simulink block for communication with temperature Arduino**

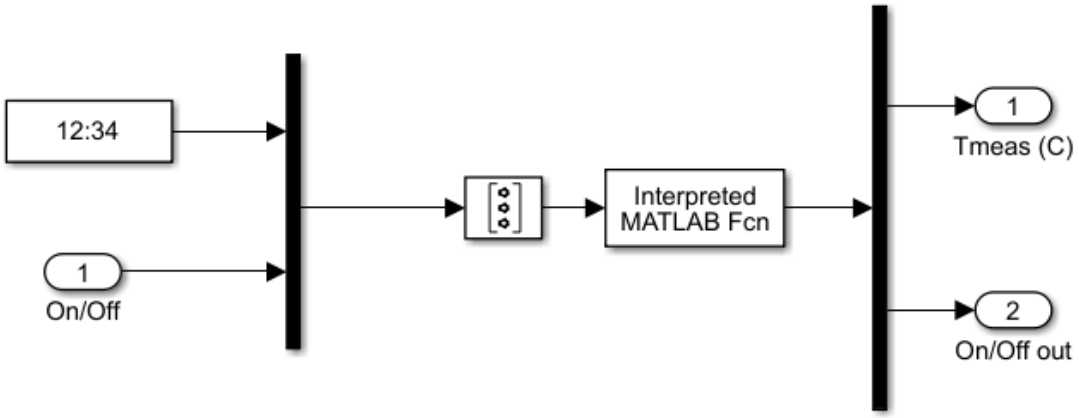


**Figure 27. Under the mask of the Simulink block for temperature**

**Experimentally Verifying Temperature Readings**

The temperature sensor measurements were verified in Simulink by connecting temperature sensors to flow outputs of the chillers. Chiller 1 was set to 17 °C and Chiller 2 to 23 °C. The Arduino measured and communicated measured temperatures of 17.35 °C and 23.69 °C, respectively (Figure 28). Simulink activated those first two MOSFETs on the Arduino, while all the others were left turned off. The other 14 temperature readings were either -2 (indicating the Arduino sent the value “inf”), or very large numbers outside the normal operating range of fluids on the testbed. Both of the actual temperature readings are higher than values set by the chillers. This is likely because the point of measurement on the testbed is quite a distance away from the point of measurement in the chillers, so the readings may not match identically. Over the distance traveled, the fluid may be slighted heated by ambient heat in the room. Multiple experiments like this were performed over a temperature range of 10-35 °C and consistent results were shown.

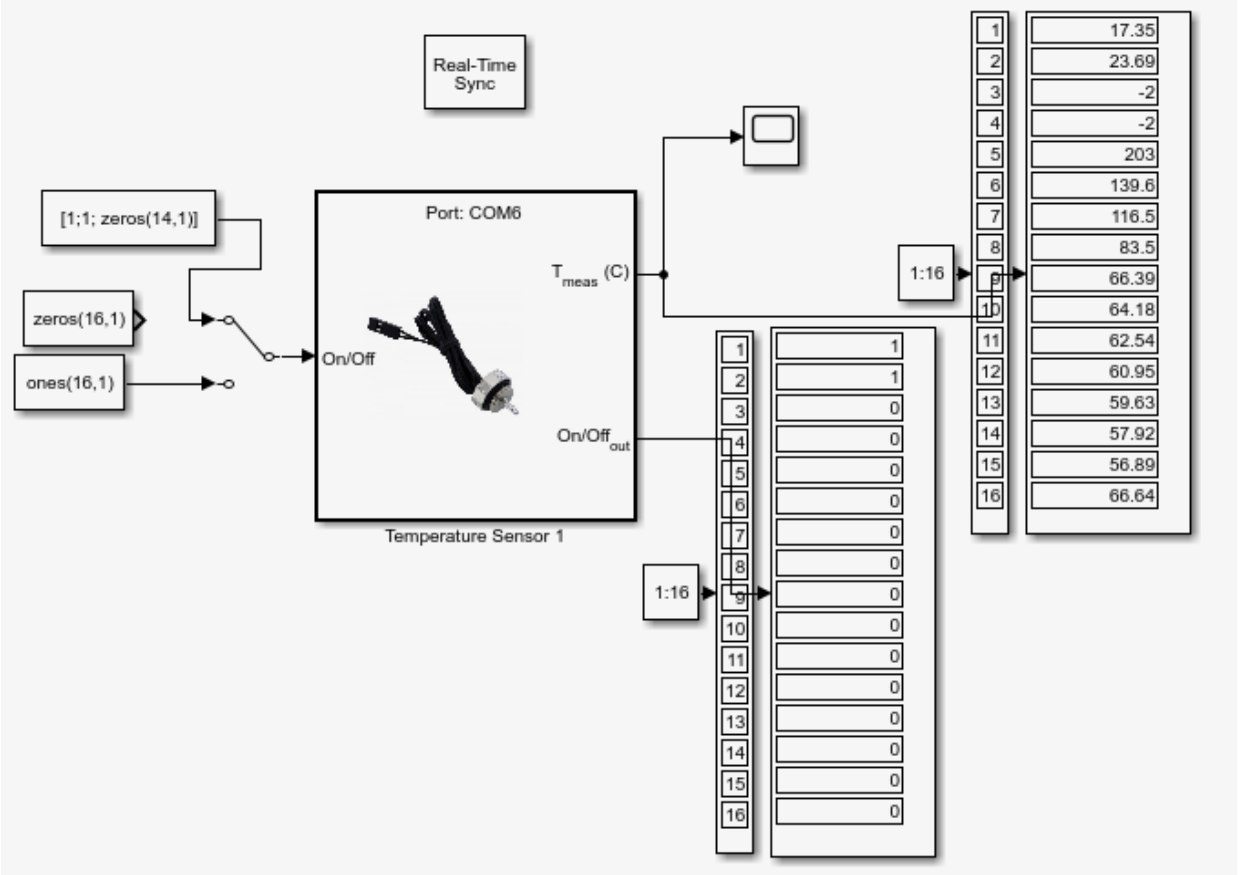


**Figure 28. Experimental verification of temperature sensor measurements**

# Chapter 5

## Serial Communication of Flow Rate Readings and Pump Control

### Overall Communication Scheme

Similarly to when measuring temperatures, Simulink sends the Arduino the desired MOSFET on/off state of each flow sensor. In addition, Simulink sends a desired flow rate for each pump, and the Arduino performs PI control with anti-windup to match the pump speed to that desired flow rate. The Arduino sends back to Simulink the on/off state which it has currently stored, as well as the reference and measured flow rates for each sensor.

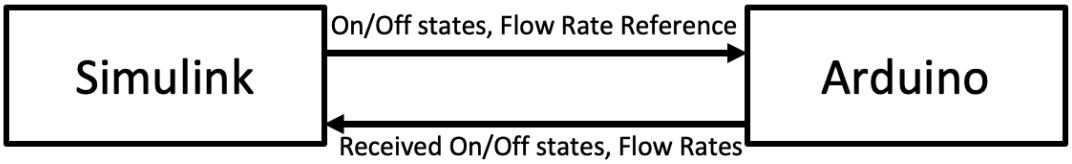


**Figure 29. Flowchart for serial communication of flow rate and pump control data**

### Developing Arduino Code

The flow rates are measured as detailed in Chapter 3, noting the different measurement techniques for the pump's output versus an external Koolance flowmeter. Both Arduino codes for serial communication of flow rates (Flow_Pump_for_External_Flowmeters or Flow_Pump_for_Pump_Encoder_Output) operate in a similar manner as discussed in Chapter 4 for the temperature sensors. The Arduino waits to receive a message from Simulink which it expects to have 14 comma separated values. The first and last character are a signature indicating

that the entire message has been sent correctly, in this case, ‘F’ (for flow rate). The next six values are binary numbers indicating the on/off state of each sensor’s MOSFET, and the remaining six values are the flow rate references for each pump.

Due to the methods of measuring flow rates (either counting times for each individual wave pulse or averaging the pulses over time), the readings are naturally quite noisy. To compensate for this, a digital filter is used to smooth out the measured flow rates. The filter places 95% weight on the previously filtered measurement, and 2.5% weight each on the current and previous raw readings. This is approximately a first-order Butterworth filter for a sampling rate of 20 Hz and a cutoff frequency of 0.1 Hz. This filter successfully reduces the noise of the measurement, but it does cause a slight delay as the moving average depends heavily on previous readings instead of current readings.

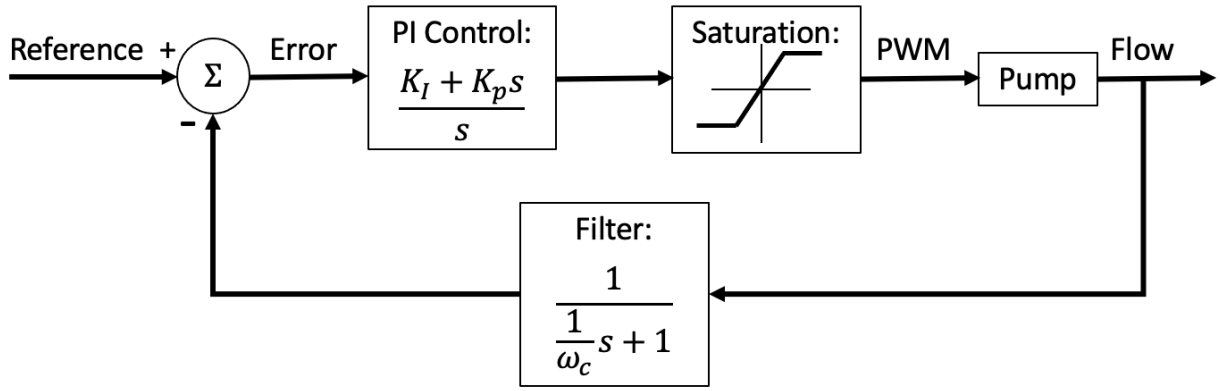


**Figure 30. Block diagram for PI control and filtering of flow rates**

After obtaining the filtered flow rate, the Arduino uses PI control to determine the PWM signal for the corresponding pump (sensor 1 corresponds with pump 1, etc.) to match the reference flow rate provided by Simulink. First, the error is determined by subtracting the filtered

flow rate from the reference. The integral term is calculated iteratively by adding to itself the product of the error, $K_i$, and the time interval:

```
integral[i] = integral[i] + error[i]*Ki*dt/1000;
```

A common issue of integral control that needs to be addressed is anti-windup. If the reference is beyond the achievable performance of the system (for example, Simulink asking for a flow of 10 L/s when the pump's maximum output is 0.138 L/s), then even once the system reaches its steady state, there will be error that continues to compound in the integral term. When the set point then drops down into a value in the feasible output range, the integral term will be so large that the controller experiences a significant delay in reaching that reference (the controller is "wound-up" too high). A similar scenario can occur if the reference point is below an achievable value. To prevent that from occurring, after calculating the integral term, it is then compared to the minimum and maximum bounds of the feasible PWM output range (0 and 255 respectively). If the integral term is larger than the maximum, then it remains equal to the maximum. Similarly, if it is smaller than the minimum, then it remains equal to the minimum. This process is also referred to as "saturation." Some pumps on the market are not responsive across the entire 0-255 PWM range. For example, a given pump may not move any fluid below a PWM of 50 and maxes out around 200. Then, the maximum and minimum values in the Arduino code should be adjusted to the appropriate PWM values that cause the minimum and maximum response in the pump.

The proportional term is calculated as the product of the error with $K_p$. The PWM output signal is then the sum of the proportional and integral terms. Once more the output must be saturated. Even though the integral term was already checked to eliminate windup, the summation of the two terms can again cause the output to be outside of the PWM range. If the

calculated PWM is either above the maximum or below the minimum, then it is replaced with the appropriate boundary value.

Through experimentation, appropriate values for the control parameters $K_i$ and $K_p$ were determined to be 500 and 1000, respectively. The seemingly large magnitudes of these constants arise from converting flow rates on the scale of 0.01 L/s to PWM signals of the scale 100, a difference of four orders of magnitude. This PI control algorithm has proven effective at matching flow reference supplied to the Arduino. There is, however, a noticeable delay in the response, compounded from the individual delays caused by filtering and PI control. On average, it takes the Arduino ten seconds to match a step change in the reference. For the thermal systems that this testbed is intended to handle, where temperatures may change on a timescale of tens of seconds, a flow rate response time of approximately ten seconds should not pose any significant issues. There are several future improvements that can be made to the PI control algorithm to promote faster response times (see Suggestions for Future Work).

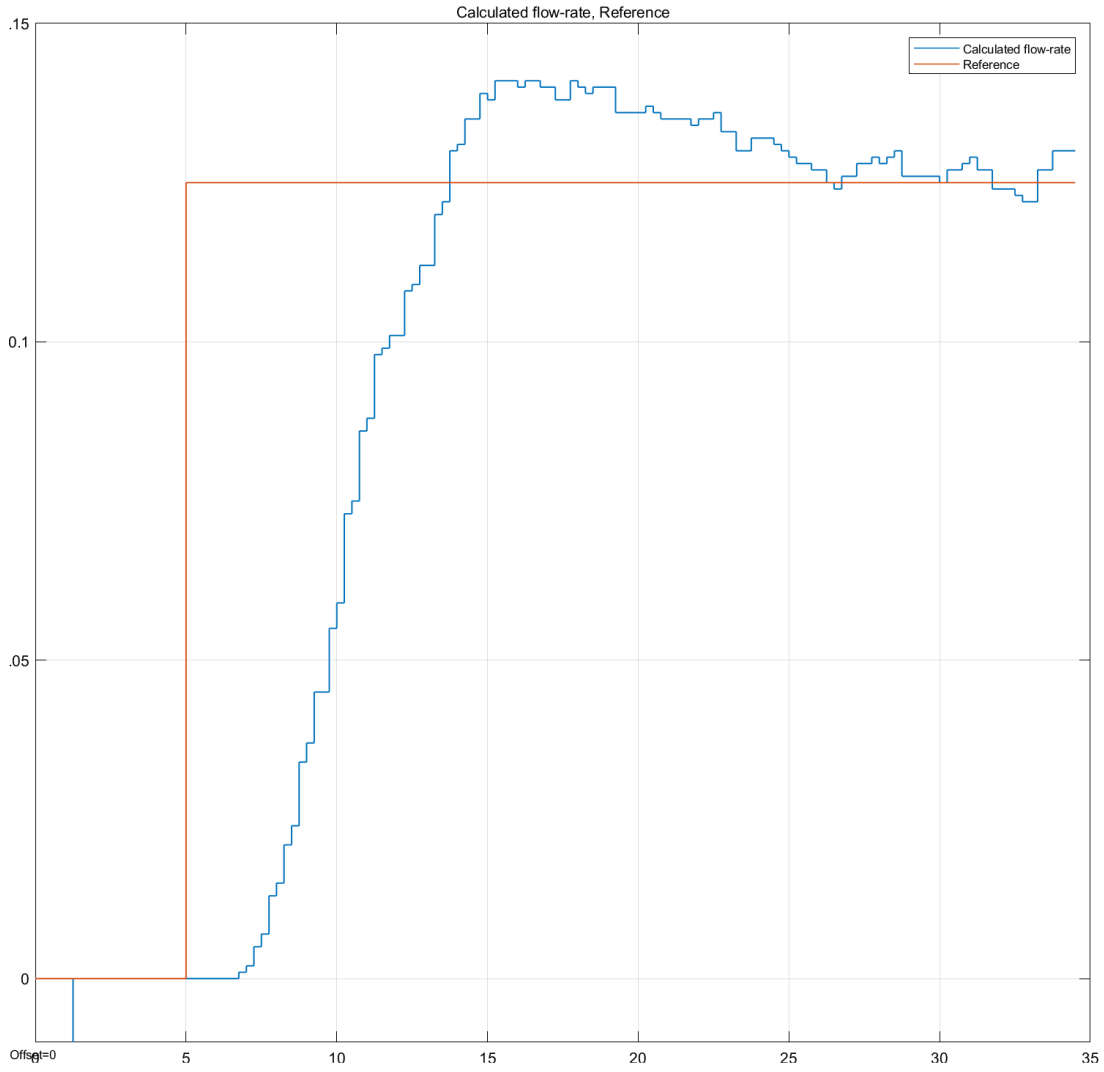


**Figure 31. Arduino controlled pump response (blue) to a step reference (yellow), as measured by an external flowmeter**

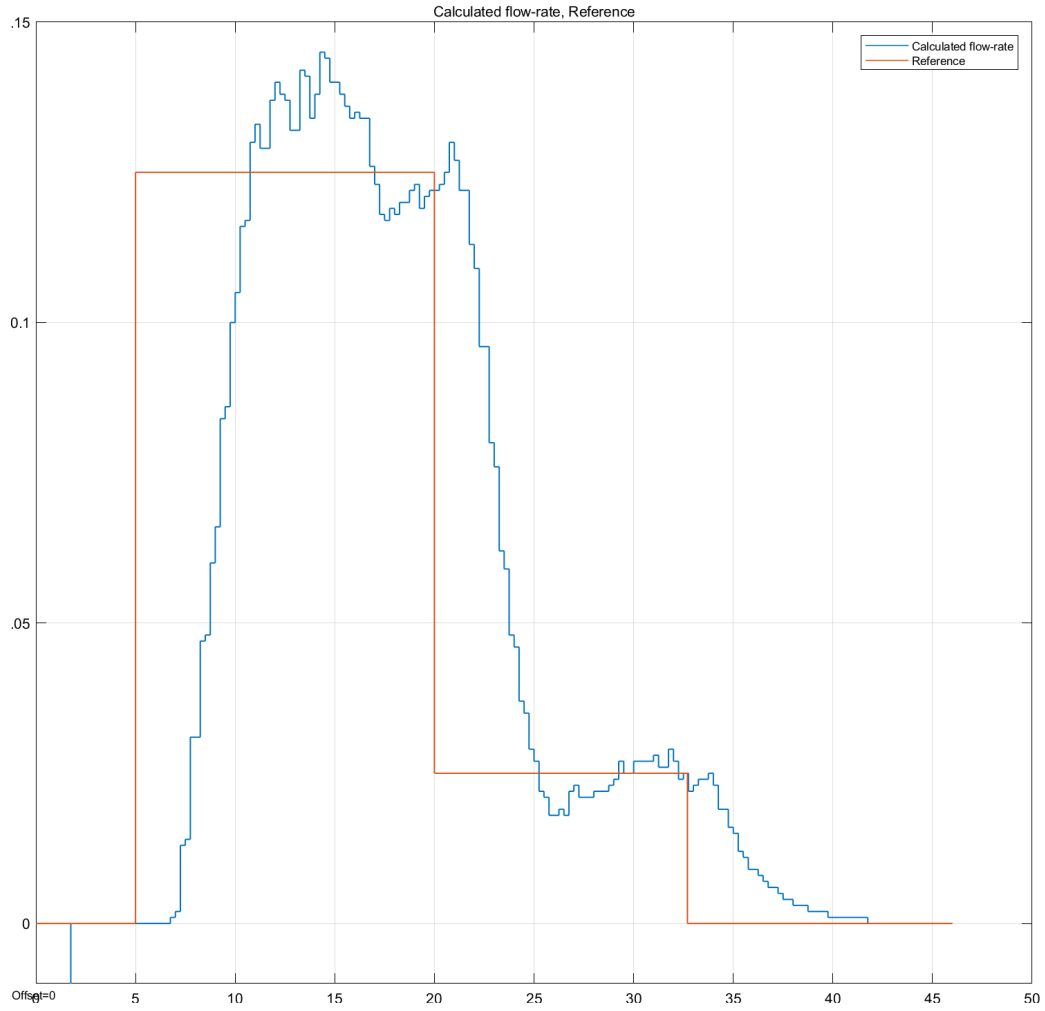


**Figure 32. Arduino controlled pump response (blue) to multiple step references (yellow), as measured by an external flowmeter**

## Developing MATLAB Code

The MATLAB function for communicating flow rate information (Flow_Meas_Fcn_1.m) takes a 17x1 vector as the input. The first value is the simulation time, the next six are the desired MOSFET on/off states, and the final six are the desired flow rate references. In the same fashion as the MATLAB function for temperature communication, a persistent "serialport" object is created to send and receive data over the serial line.

MATLAB sends the on/off values and flow references (starting and ending the message with ‘F’, and delimiting each value with a comma). The expected message returned by the Arduino starts and ends with the character ‘f’ and is also comma delimited. After separating the message by the commas and removing the signatures, the expected data has 18 values: six on/off states, six flow references, and six filtered flow rates. MATLAB converts these values from strings to numbers and outputs them. Similarly to the temperature communication, MATLAB uses default values of -1 to replace any values that might be “NaN” or “inf”, although these are not expected from the Arduino.

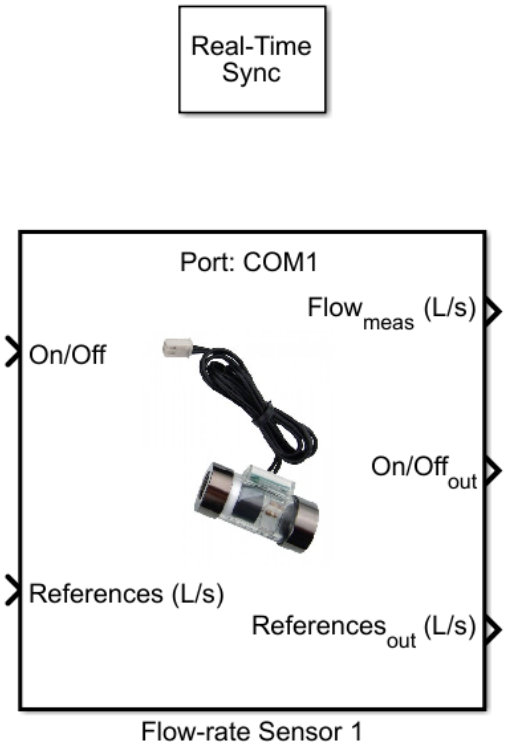


**Figure 33. Simulink block for communicating with pump/flow Arduino**

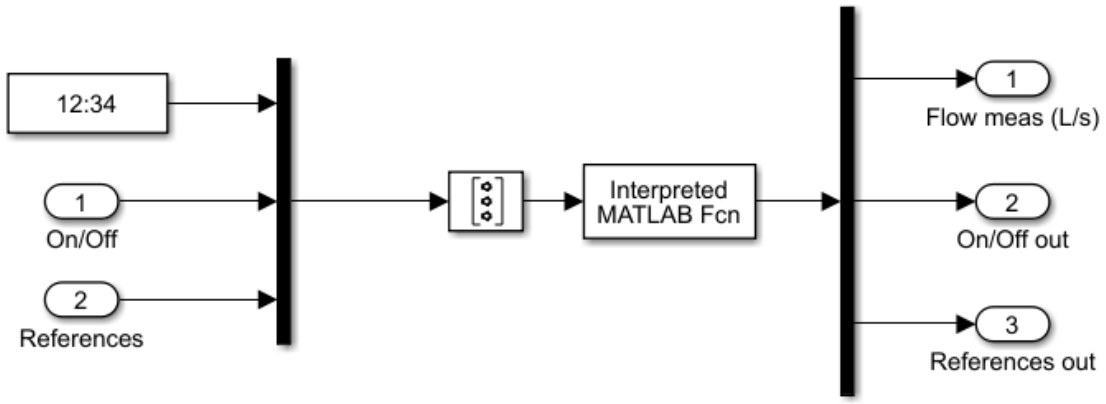


**Figure 34. Under the mask of the Simulink block for flow rates**

In the current instrumentation, the Arduino takes its measurements at a constant frequency defined in its code. But, the user has control over the communication rate between the Arduino and desktop from the Simulink block. Whenever the MATLAB function is called, that is when Simulink sends a message to the Arduino and the Arduino responds back. So, by changing the update rate of the Simulink block, the user can determine how often the devices communicate. Of course, there would not be any point to having them communicate faster than the Arduino is updating as it would be sending redundant information.

One noteworthy result of this circuitry and code is the behavior when the MOSFET for a sensor is turned off to simulate a fault. Because this results in no flow being measured, the Arduino believes the flow to be zero. If the flow reference is any positive value, then the integral control starts pushing the PWM upward to try and reach steady-state, resulting in eventually reaching the maximum PWM output (as the measured flow will not increase). So, when simulating a fault, Simulink will receive a constant flow rate of zero despite any reference point, meanwhile the Arduino starts pushing the pump at maximum speed.

**Experimentally Verifying Flowmeter Readings**

To verify proper reading and calibration of the external flowmeter, an experiment was performed to compare the results of measuring flow rate via four different sensors (see results in Table 3). The three methods of measuring flow outlined in this thesis (via INS-FM17N and SEN-AP008B sensors as well as the pump's duty cycle output) were compared against a calibrated Atrato Titan-740 flowmeter borrowed from another laboratory. The results showed accuracy and agreement among between the two Koolance sensors and the Atrato sensor. For flow rates 0.0225 L/s and above, the pump's duty cycle output is found to be ten times the flow rate measured by the Atrato sensor. For each sensor type, the normalized error residual was calculated. While this experiment was not performed over all possible flow rates of the sensors (due to limitations in the pumps speed as it overcomes gravity and pushes fluid through three sensors), it is sufficient to verify that the Arduinos are capable of performing this task to a reasonable degree of accuracy.

**Table 3. Measuring flow rate with four different methods for verification**

| Trial | Atrato (L/s) | INS-FM17N (L/s) | SEN-AP008B (L/s) | Pump Duty Cycle Output | (Pump Duty Cycle Output)/10 |
|---|---|---|---|---|---|
| 1 | 0.0222 | 0.0213 | 0.0227 | 0.2858 | 0.02858 |
| 2 | 0.023 | 0.0234 | 0.0245 | 0.2951 | 0.02951 |
| 3 | 0.0255 | 0.0262 | 0.0287 | 0.2586 | 0.02586 |
| 4 | 0.0307 | 0.0313 | 0.0321 | 0.3199 | 0.03199 |
| 5 | 0.0334 | 0.0346 | 0.0365 | 0.3482 | 0.03482 |
| 6 | 0.0434 | 0.0456 | 0.0472 | 0.4561 | 0.04561 |

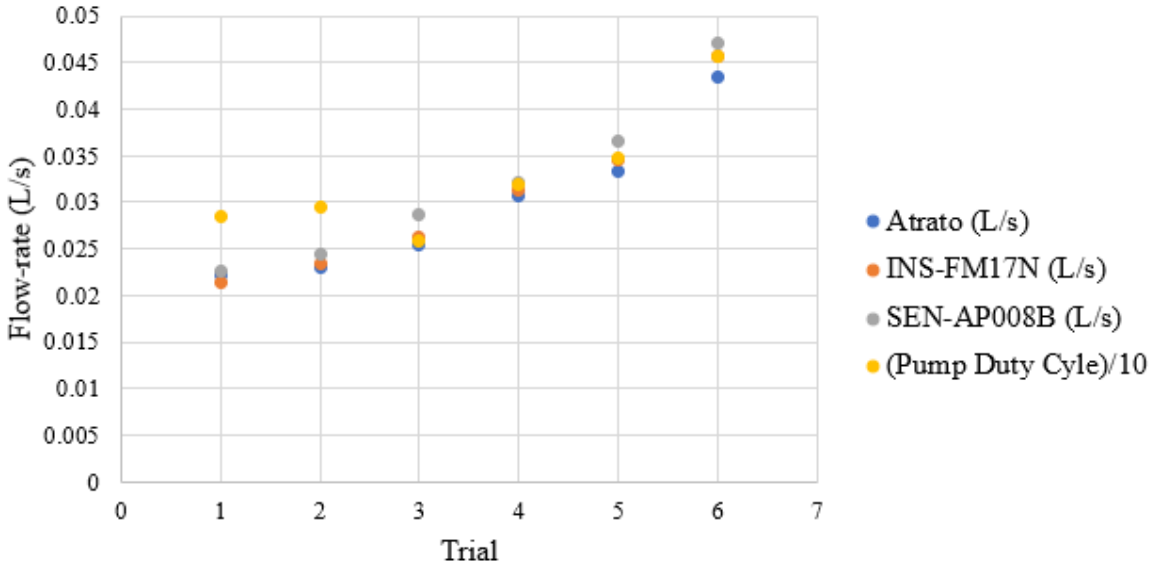


**Figure 35. Comparing multiple methods of flow measurements**

$$Normalized\ Error\ Residual = \frac{\sum_{trials}(Atrato\ reading - Arduino\ reading)^2}{\sum_{trials}(Atrato\ reading)^2} \quad (4)$$

**Table 4. Normalized error residual for each flow rate sensor type tested**

| **Sensor:** | INS-FM17N | SEN-AP008B | (Pump Duty Cycle Output)/10 |
|---|---|---|---|
| **Normalized Error Residual:** | 0.00144 | 0.00690 | 0.0163 |

# Chapter 6

# Conclusions

## Summary of Work

This thesis presents the design and construction of a thermal management testbed with instrumentation performed by Arduino microcontrollers. The testbed is modular, meaning components and sensors can be arranged in any geometry. The use of the testbed is mainly to verify advanced Simulink controllers on a physical thermal system. While the testbed is able to measure temperatures and flow rates to a reasonable accuracy, there are still multiple improvements and additions to be made to increase its capabilities even further. The most exciting work yet to be done is using the testbed to learn more about the performance of advanced control systems.

## Analysis on the Viability of Arduino Instrumentation for a Testbed

After testing the Arduino shields, this system of instrumentation seems very promising. The temperature readings are already quite accurate, and there are still multiple ways to improve the calibration and consistency across multiple sensors (detailed in Suggestions for Future Work). The flow rate measurements and pump control are consistent, with the pumps showing an appropriate response for time a single-phase system. The overall communication scheme between multiple Arduinos and Simulink is quite effective and allows Simulink to focus on running the control algorithm while the Arduinos focusing on measuring, filtering, and actuating.

Once a control system (such as an advanced MPC) is developed in Simulink, it will be very straightforward to drop in the necessary Simulink blocks for Arduino communication to test the controller on the testbed.

Perhaps most significantly, the overall cost of this instrumentation scheme is significantly reduced compared to higher-end DAQs. For the cost of a couple of Arduino Megas, basic circuitry components, and the printing and shipping of custom PCBs, this instrumentation system shows great promise for its accuracy and adaptability. Table 5 details the cost of the minimum materials required for the instrumentation of 16 temperature sensors, six external flowmeters, and 12 pumps (on a total of three Arduino shields). Note that the cost below does not include price of any sensors or thermal components, as those would be necessary whether using Arduino instrumentation or a higher-end DAQ. Additionally, while three PCBs is the minimum required, the manufacturer that was used sold them as packs of five identical boards for $2 total. So, a total of 15 boards were ordered for a total of $6, which yields plenty extra boards for adding more Arduinos in the future or making custom modifications.

**Table 5. Total cost of Arduino instrumentation**

| Item | Quantity | Unit Price | Subtotal |
|---|---|---|---|
| Arduino Mega | 3 | $40.30 | $120.90 |
| 10 kΩ resistor | 16 | $0.13 | $2.08 |
| 1 kΩ resistor | 12 | $0.01 | $0.12 |
| Molex male pinheads | 40 | $0.83 | $33.20 |
| Molex female connectors | 40 | $0.34 | $13.60 |
| Molex contacts | 40 | $0.20 | $8.00 |

| | | | |
|---|---|---|---|
| N-channel MOSFET | 28 | $0.74 | $20.72 |
| Fritzing PCB order | 3 | $0.40 | $1.20 |
| | | **TOTAL** | **$199.82** |

Will the Arduino soon take over the market of high-end expensive DAQs? Well no, those still have their many advantages and uses for projects that require much faster update rates. For lab scale setups, however, it is a very fitting replacement. The Arduino instrumentation system will allow academic labs to invest resources into higher-end sensors and equipment to build more complicated thermal setups. With the added benefit of being able to adapt and alter the source codes as needed, the Arduino-Simulink system complements the modularity of the entire testbed.

**Suggestions for Future Work**

One minor issue in the current instrumentation is inconsistency among the temperature measurements. Currently, when two sensors are measuring the temperature of the same fluid, their readings can differ by up to 0.5 ºC. This is because of small inconsistencies in the voltage drop across the MOSFETs in the circuitry. By using a multimeter to test them, it seems that when 5 V power is passed through the MOSFETs, their output voltages vary from 2.00-2.10 V. Since that voltage is used in converting the voltage drop across the sensor's resistor to a temperature value, when the value in the code is off from the actual value by even 0.1 V, it can cause an approximate 0.5 ºC error in the measurement. To fix this, after soldering together one of the temperature shields, the output voltage of each MOSFET should be individually measured,

and then in the Arduino code, use each MOSFET's unique output voltage to calculate the resistance for that sensor.

An "ease of use" improvement for the testbed is to simplify the method for connecting sensors to the Arduinos. Currently, each sensor and pump are crimped to a MOLEX pin which then plugs directly into the Arduino shield, but this causes a messy tangle of wires once there are multiple things all plugged in. Additionally, the MOLEX pins have a very firm connection, which is great for the circuitry, but makes it tricky to constantly plug and unplug them. Instead, all of the Arduino circuitry could be on a server rack behind the testbed and the MOLEX pins on the Arduino shields connected to a strip of USB ports. Then, the sensors' wires all need to be configured to a male USB plug instead of a female MOLEX connector. The USB connection would be much easier to disconnect and reconnect while testing the sensors in various configurations, without losing any stability in the circuitry connections (or risk in harming the wires from struggling to disconnect MOLEX pins constantly). Hiding the Arduino circuitry behind the USB hub on the server rack also affords a cleaner, more put-together look, and USB extension cables are widely available on the market (whereas MOLEX are not).

Another step in developing this testbed instrumentation further is to design and create more Arduino shields for other types of sensors, most notably a pressure sensor. Most pressure sensors on the market output an analog voltage, so to measure this with an Arduino, one could connect the sensor output to an analog read pin. The Arduino shield for measuring pressure sensors should have 16 locations for sensor attachments, as the Arduino Mega has 16 analog read pins. The pressure sensors are useful for measuring head-loss through various geometries, pressure loss through a heat exchanger, and the height of fluid in a reservoir tank.

A possible future improvement for the instrumentation would be employing a slightly more advanced control algorithm than PI on the Arduino controlling pump flow. The heavy filtering causes the delay time in the responses, and while the delay does not threaten to pose any immediate issues, there are some methods which may help lessen it. One approach is to replace the constant filter with an adaptive one. The Arduino can approximate the derivative of the reference signal to detect when it is changing and reduce the weighting of the filter to allow for a faster response. Another adaptive filter technique that could be used is to alter the number of terms in the moving average.

A second method to reduce the delay is to add feedforward to the Arduino control scheme. One can perform an experiment to find the PWM signal corresponding to many different flow rates, and then when a reference is given, automatically jump to that PWM and use the PI to fine-tune and maintain steady state against disturbances. The caveat is that this pre-determined PWM response will depend on the current thermal system running; factors such as head loss through flow resisting elements and the number of pumps attached will affect what PWM speed is necessary to reach a given flow. So, there is no fully modular version of feedforward control that will work in every instance. But, if a specific fluid loop is setup as a case steady and it is important to have the best flow rate response times as possible, feedforward should promote a faster response time.

# Appendix A

# Arduino Code

## Temp_Shield

```
/*
  Jonah Glunt
  Penn State PAC Lab
  Last Updated: 21 March 2022

  Arduino Code for the Temperature Sensor Board

  This program reads the temperatures of all 16 pins. When it receives a
serial message from
  Simulink telling it which MOSFETs to "turn on", it does so, and responds
back with the current
  state of the MOSFETS and all of the measured temperatures.
*/

// number of locations for sensors on the board
const int NUMSENSORS = 16;
// analog pins connected to sensors
const int readPins[16] = {A0, A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11,
A12, A13, A14, A15};
// digital pins that control MOSFETS
const int controlPins[16] = {13, 12, 11, 7, 3, 14, 17, 19, 22, 24, 28, 32,
36, 40, 44, 46};

// where the voltage measured by analog read pins will temporarily be stored
float voltage;
// where the calculated resistance of a sensor will be temporarily stored
float resistance;
// where the temperature readings will be stored
float temp[NUMSENSORS];
// where the MOSFET on/off states will be stored
bool fetOnOff[NUMSENSORS];

// expected length of message to be received from Simulink
const int MSGLENGTH = NUMSENSORS + 2;
// character expected at beginning and end of each message from Simulink
const char SIGNATURE = 'T';
// character attached to beginning and end of all outgoing messages to
Simulink
const char signature = 't';

void setup() {
  // connect the AREF pin to an internal 2.56V reference
  analogReference(INTERNAL2V56);
  Serial.begin(9600);
  for(int i = 0; i < NUMSENSORS; i ++) {
```

```
    // default each MOSFET state to off
    fetOnOff[i] = false;
    pinMode(readPins[i], INPUT);
    pinMode(controlPins[i], OUTPUT);
    // turn off all of the MOSFET control pins
    digitalWrite(controlPins[i], LOW);
    // default the temperature measurements to zero
    temp[i] = 0;
  } // end for
} // end setup()

void loop() {
////////////////////////////////
// Read and update all measurements

  // Iterate through all sensors
  for(int i = 0; i < NUMSENSORS; i ++) {
      // activate the current MOSFET pin if it is meant to be
      if(fetOnOff[i]) {
        digitalWrite(controlPins[i], HIGH);
      } // end if
      // read the voltage across the sensor
      voltage = analogRead(readPins[i])*2.56/1023;
      // convert that voltage to resistance using Ohm's law
      resistance = 10*voltage/(2.05-voltage);
      // use the calibration curve to convert the resistance to a temperature
      temp[i] = resToTemp(i, resistance);
      // turn off the current pin if it was turned on
      if(fetOnOff[i]) {
        digitalWrite(controlPins[i], LOW);
      } // end if
  } // end for

////////////////////////////////
// Serial communication

  // While there is data on the serial line
  while (Serial.available() > 0) {
    // index variable for the message position
    static unsigned int msg_pos = 0;
    // character array to store the incoming message
    static char msg[MSGLENGTH];

    // read the current character on the serial line
    char inByte = Serial.read();

    // if the current character does not signify the end of the message, and
the message is within the maximum length
    if (inByte != '\n' && (MSGLENGTH - 1 - msg_pos >= 0) ) {
      // add the character to the received message and increase the index by
one
      msg[msg_pos] = inByte;
      msg_pos ++;
    } else {
      // check that the message is the correct length and has correct
indentifying characters at beginning and end
```

```
      if (msg_pos == MSGLENGTH && msg[0] == SIGNATURE && msg[MSGLENGTH - 1]
== SIGNATURE) {
        // change the stored MOSFET on/off status to what was received in the
message
        for (int i = 0; i < NUMSENSORS; i ++) {
          if(msg[i+1] == '0') { fetOnOff[i] = false; }
          else { fetOnOff[i] = true; }
        } // end for

        // sent the currently stored data for the MOSFET on/off states and
the temperatures
        sendData();
      } // end if

      // reset the message index to zero for the next message
      msg_pos = 0;
    } // end if-else
  } // end while
} // end loop()

/*
 *  This function calibrates the measured resistance to a temperature using
 *  a curve calculated in Matlab from manufacturer's data. The first input
 *  is the sensor number, in case later updates to this code include
different
 *  calibration curves for each sensor (maybe different models of sensors,
 *  or different circuitry components that cause a noticeable inaccuracy).
*/
float resToTemp(int sensorNum, float res) {
  float calcTemp = 256.7 * pow(res, -0.1471) - 157.9;
  return calcTemp;
} // end resToTemp(int, float)

/*
 * This function sends the currently stored data for the MOSFET on/off
 * states and temperature readings, with each value separated by a comma.
 */
void sendData() {
  // Start the message with the Arduino's signature
  Serial.print(signature);
  Serial.print(',');
  for(int i = 0; i < NUMSENSORS; i ++) {
    Serial.print(fetOnOff[i]);
    Serial.print(',');
  }
  for(int i = 0; i < NUMSENSORS; i ++) {
    Serial.print(temp[i]);
    Serial.print(',');
  }
  // End the message with the Arduino's signature.
  Serial.println(signature);
}
```

**Flow_Pump_for_External_Flowmeters**

```
/*
  Jonah Glunt
  Penn State PAC Lab
  Last Updated: 24 March 2022

  Arduino Code for the External Flow Rate Sensor Board

  This program reads the flow-rates from six external Koolance flowmeters.
When it receives a serial message from
  Simulink telling it which MOSFETS to "turn on" and a set of flow-rate
references, it
  updates its stored values and response back with the current MOSFET states,
the
  flow rate references, and filtered flow rates.
*/

// number of locations for sensors on the board
const int NUMSENSORS = 6;
// interrupt pins attached to sensors
const int encoderPins[NUMSENSORS] =        { 3, 2,18,19,20,21};
// digital pin that control MOSFETS
const int encoderControlPins[NUMSENSORS] = {A0,A1,A2,A3,A4,A5};
// PWM output pins
const int PWMPins[NUMSENSORS] =            {13,12,11,10, 9, 8};

// where the time of previous loop iteration will be stored
unsigned long prevTime = 0;
// where the time of the current loop iteration will be stored
unsigned long currTime = 0;
// where the MOSFET on/off states will be stored
bool fetOnOff[NUMSENSORS];
// where number of counts for each sensor will be stored
int counts[NUMSENSORS];
// where the calculated raw flow rates will be stored
float rawData[NUMSENSORS];
// where the previous raw data will be stored
float prevRawData[NUMSENSORS];

// remembering factor for the digital filter; the weight on the previously
filtered value
const float rememberingFactor = 0.95;
// where the calculated filtered flow rates will be stored
float filteredData[NUMSENSORS];
// where the previously filtered data will be stored
float prevFilteredData[NUMSENSORS];

// where the flow references will be stored
float flowSetPoints[NUMSENSORS];
// length of time window for Arduino to measure counts; reciprocal of
updating frequency
const float dt = 100; // milliseconds
// constant for integral control
```

```
const float Ki = 500;
// constant for proportional control
const float Kp = 1000;
// minimum PWM output to get response from pump
const int MIN = 0;
// maximum PWM output to get response from pump
const int MAX = 255;
// where the error between filtered flow rate and reference will be stored
float error[NUMSENSORS];
// where the integral control term will be stored
float integral[NUMSENSORS];
// where the calculated PWM control outputs will be stored
float PWMOutputs[NUMSENSORS];

// character expected at beginning and end of each message from Simulink
const char SIGNATURE = 'F';
// character attached to beginning and end of all outgoing messages to
Simulink
const char signature = 'f';

void setup() {
  Serial.begin(9600);
  for(int i = 0; i < NUMSENSORS; i ++) {
    pinMode(encoderPins[i], INPUT);
    pinMode(encoderControlPins[i], OUTPUT);
    pinMode(PWMPins[i], OUTPUT);
    // default each MOSFET state to off
    fetOnOff[i] = false;
    digitalWrite(encoderControlPins[i], LOW);
    // default all PWM output pins to off
    digitalWrite(PWMPins[i], LOW);

    // default all data to zero
    counts[i] = 0;
    prevRawData[i] = 0;
    prevFilteredData[i] = 0;
    flowSetPoints[i] = 0;
    integral[i] = 0;

    // attach interrupt to each encoder reading pin
    attachInterrupt(digitalPinToInterrupt(encoderPins[0]),
interruptFunction1, CHANGE);
    attachInterrupt(digitalPinToInterrupt(encoderPins[1]),
interruptFunction2, CHANGE);
    attachInterrupt(digitalPinToInterrupt(encoderPins[2]),
interruptFunction3, CHANGE);
    attachInterrupt(digitalPinToInterrupt(encoderPins[3]),
interruptFunction4, CHANGE);
    attachInterrupt(digitalPinToInterrupt(encoderPins[4]),
interruptFunction5, CHANGE);
    attachInterrupt(digitalPinToInterrupt(encoderPins[5]),
interruptFunction6, CHANGE);
  } // end for

  prevTime = millis();
} // end setup()
```

```
void loop() {
  // store the current time
  currTime = millis();

  // if it has been at least dt seconds since last update
  if(currTime - prevTime > dt) {
    // update the previous time to the new time
    prevTime = currTime;

    // iteate through all sensors
    for(int i = 0; i < NUMSENSORS; i++) {

      ////////////////////////////
      // Read all flow-rates sensors

      // set the MOSFET to the correct state
      digitalWrite(encoderControlPins[i], fetOnOff[i]);

      // out of the two equations below, uncomment the correct one for the
sensor being used
      //rawData[i] = counts[i]/dt*1000.0*0.283/60/2; // for SEN-FM18T10
sensor, L/s
      rawData[i] = counts[i]/dt*1000.0*0.307/60/2; // for INS-FM17N sensor,
L/s

      // reset the counts to zero
      counts[i] = 0;

      // filter the data to remove noise
      filteredData[i] = rememberingFactor*prevFilteredData[i] + (1-
rememberingFactor)/2*(rawData[i]+prevRawData[i]); // L/s
      // store the current data in the previous data for next iteration
      prevFilteredData[i] = filteredData[i];
      prevRawData[i] = rawData[i];

      ////////////////////////////////////////////////////
      // Perform PI control to match the flow-rate references

      // caculate error between reference and actual flow
      error[i] = flowSetPoints[i] - filteredData[i];
      // calculate integral control term
      integral[i] = integral[i] + error[i]*Ki*dt/1000;
      // perform anti-windup on integral control term
      if(integral[i] > MAX) {
        integral[i] = MAX;
      } else if(integral[i] < MIN) {
        integral[i] = MIN;
      } // end if/else-if

      // calculate PWM outputs by combining proportional and integral control
      PWMOutputs[i] = Kp*error[i] + integral[i];
      // perform anti-windup on final PWM outputs
      if(PWMOutputs[i] > MAX) {
        PWMOutputs[i] = MAX;
      } else if(PWMOutputs[i] < MIN) {
```

```
        PWMOutputs[i] = MIN;
      } // end if/else-if
      // write the calculated PWM output
      analogWrite(PWMPins[i], PWMOutputs[i]);
    } // end for
  } // end if

  ////////////////////////////////////////////
  // Serial communication

  // while there is data on the serial line
  while(Serial.available() > 0) {
    // index variable for the message positoin
    static unsigned int msg_pos = 0;
    // character array to sore the incoming message
    static char msg[1000];

    // read the current character on the serial line
    char inByte = Serial.read();

    // if the current character does not signify the end of the message, and
the message is within the maximum length
    if(inByte != '\n' && msg_pos < 1000) {
      // add the character to the received message and increase the index by
on
      msg[msg_pos] = inByte;
      msg_pos ++;
    }
    else {
      // check that the message has correct identifying characters at
beginning and end
      if(msg[0] == SIGNATURE && msg[msg_pos - 1] == SIGNATURE) {
        // separate the message as delimitted by commas
        char *strings[NUMSENSORS*2 + 2];
        char *ptr = NULL;
        byte index = 0;
        ptr = strtok(msg,",");
        while(ptr != NULL) {
          strings[index] = ptr;
          index++;
          ptr = strtok(NULL, ",");
        }

        // change the stored MOSFET on/off to what was received in the
message
        for(int i = 1; i <= NUMSENSORS; i ++) {
          if(*strings[i] == '1') {fetOnOff[i-1] = true; }
          else {fetOnOff[i-1] = false; }
        } // end for

        //change the stored reference set point to what was received in the
message
        for(int i = NUMSENSORS + 1; i <= 2*NUMSENSORS; i ++) {
          flowSetPoints[i - NUMSENSORS - 1] = atof(strings[i]);
        }
```

```
        // send the currently stored data for MOSFET on/off, flow references,
and filtered flow
        sendData();
      } // end if

      // reset the message index to zero for the next message
      msg_pos = 0;
    } // end else-if
  } // end while
} // end loop()

// Interrupt function for sensor 1
void interruptFunction1() {
  counts[0] = counts[0] + 1;
}

// Interrupt function for sensor 2
void interruptFunction2() {
  counts[1] = counts[1] + 1;
}

// Interrupt function for sensor 3
void interruptFunction3() {
  counts[2] = counts[2] + 1;
}

// Interrupt function for sensor 4
void interruptFunction4() {
  counts[3] = counts[3] + 1;
}

// Interrupt function for sensor 5
void interruptFunction5() {
  counts[4] = counts[4] + 1;
}

// Interrupt function for sensor 6
void interruptFunction6() {
  counts[5] = counts[5] + 1;
}

/*
 * This function sends the currently stored data for the MOSFET on/off
 * states, flow references, and filtered flow, with each value
 * separated by a comma.
 */
void sendData() {
  Serial.print(signature);
  Serial.print(',');
  for(int i = 0; i < NUMSENSORS; i ++) {
    Serial.print(fetOnOff[i]);
    Serial.print(',');
  }
  for(int i = 0; i < NUMSENSORS; i ++) {
    Serial.print(flowSetPoints[i],3);
    Serial.print(',');
```

```
  }
  for(int i = 0; i < NUMSENSORS; i ++) {
    Serial.print(filteredData[i],3);
    Serial.print(',');
  }
  Serial.println(signature);
}
```

## Flow_Pump_for_Pump_Encoder_Output

```
/*
  Jonah Glunt
  Penn State PAC Lab
  Last Updated: 24 March 2022

  Arduino Code for the Pump Encoder Sensor Board

  This program reads the flow-rates of all 6 pins. When it receives a serial
message from
  Simulink telling it which mosfets to "turn on" and a set of flow-rate
references, it
  updates its stored values and response back with the current MOSFET states,
the
  flow-rate references, and filtered flow-rates.
*/

// number of locations for sensors on the board
const int NUMSENSORS = 6;
// interrupt pins attached to sensors
const int encoderPins[NUMSENSORS] =        { 3, 2,18,19,20,21};
// digital pin that control MOSFETS
const int encoderControlPins[NUMSENSORS] = {A0,A1,A2,A3,A4,A5};
// PWM output pins
const int PWMPins[NUMSENSORS] =            {13,12,11,10, 9, 8};

// where the time of previous loop iteration will be stored
unsigned long prevTime = 0;
// where the time of the current loop iteration will be stored
unsigned long currTime = 0;
// where the MOSFET on/off states will be stored
bool fetOnOff[NUMSENSORS];
// where the on and off time of each sensor's wave will be stored
int onTime[NUMSENSORS];
int offTime[NUMSENSORS];
// where the calculated raw flow rates will be stored
float rawData[NUMSENSORS];
// where the previous raw data will be stored
float prevRawData[NUMSENSORS];

// remembering factor for the digital filter; the weight on the previously
filtered value
const float rememberingFactor = 0.95;
```

```
// where the calculated filtered flow rates will be stored
float filteredData[NUMSENSORS];
// where the previously filtered data will be stored
float prevFilteredData[NUMSENSORS];

// where the flow references will be stored
float flowSetPoints[NUMSENSORS];
// length of time window for Arduino to measure counts; reciprocal of
updating frequency
const float dt = 100; // milliseconds
// constant for integral control
const float Ki = 500;
// constant for proportional control
const float Kp = 1000;
// minimum PWM output to get response from pump
const int MIN = 0;
// maximum PWM output to get response from pump
const int MAX = 255;
// where the error between filtered flow rate and reference will be stored
float error[NUMSENSORS];
// where the integral control term will be stored
float integral[NUMSENSORS];
// where the calculated PWM control outputs will be stored
float PWMOutputs[NUMSENSORS];

// character expected at beginning and end of each message from Simulink
const char SIGNATURE = 'F';
// character attached to beginning and end of all outgoing messages to
Simulink
const char signature = 'f';

void setup() {
  Serial.begin(9600);
  for(int i = 0; i < NUMSENSORS; i ++) {
    pinMode(encoderPins[i], INPUT);
    pinMode(encoderControlPins[i], OUTPUT);
    pinMode(PWMPins[i], OUTPUT);
    // default each MOSFET state to off
    fetOnOff[i] = false;
    digitalWrite(encoderControlPins[i], LOW);
    // default all PWM output pins to off
    digitalWrite(PWMPins[i], LOW);

    // default all data to zero
    onTime[i] = 0;
    offTime[i] = 0;
    prevRawData[i] = 0;
    prevFilteredData[i] = 0;
    flowSetPoints[i] = 0;
    integral[i] = 0;
  } // end for

  prevTime = millis();
} // end setup()

void loop() {
```

```
  // store the current time
  currTime = millis();

  // if it has been at least dt seconds since last update
  if(currTime - prevTime > dt) {
    // update the previous time to the new time
    prevTime = currTime;

    // iteate through all sensors
    for(int i = 0; i < NUMSENSORS; i++) {

      ////////////////////////////
      // Read all flow-rates sensors

      // set the MOSFET to the correct state
      digitalWrite(encoderControlPins[i], fetOnOff[i]);

      // record time when pump output goes HIGH
      onTime[i] = pulseIn(encoderPins[i], HIGH);
      // record time when pump output goes LOW
      offTime[i] = pulseIn(encoderPins[i], LOW);

      // the flow rate in L/s is approximately the duty cycle divided by 10
      rawData[i] = onTime[i]/(onTime[i]+offTime[i]) / 10; // L/s

      // filter the data to remove noise
      filteredData[i] = rememberingFactor*prevFilteredData[i] + (1-
rememberingFactor)/2*(rawData[i]+prevRawData[i]); // L/s
      // store the current data in the previous data for next iteration
      prevFilteredData[i] = filteredData[i];
      prevRawData[i] = rawData[i];

      ////////////////////////////////////////////////////
      // Perform PI control to match the flow-rate references

      // caculate error between reference and actual flow
      error[i] = flowSetPoints[i] - filteredData[i];
      // calculate integral control term
      integral[i] = integral[i] + error[i]*Ki*dt/1000;
      // perform anti-windup on integral control term
      if(integral[i] > MAX) {
        integral[i] = MAX;
      } else if(integral[i] < MIN) {
        integral[i] = MIN;
      } // end if/else-if

      // calculate PWM outputs by combining proportional and integral control
      PWMOutputs[i] = Kp*error[i] + integral[i];
      // perform anti-windup on final PWM outputs
      if(PWMOutputs[i] > MAX) {
        PWMOutputs[i] = MAX;
      } else if(PWMOutputs[i] < MIN) {
        PWMOutputs[i] = MIN;
      } // end if/else-if
      // write the calculated PWM output
      analogWrite(PWMPins[i], PWMOutputs[i]);
```

```
    } // end for
  } // end if

  ////////////////////////////////////////////
  // Serial communication

  // while there is data on the serial line
  while(Serial.available() > 0) {
    // index variable for the message positoin
    static unsigned int msg_pos = 0;
    // character array to sore the incoming message
    static char msg[1000];

    // read the current character on the serial line
    char inByte = Serial.read();

    // if the current character does not signify the end of the message, and
the message is within the maximum length
    if(inByte != '\n' && msg_pos < 1000) {
      // add the character to the received message and increase the index by
on
      msg[msg_pos] = inByte;
      msg_pos ++;
    }
    else {
      // check that the message has correct identifying characters at
beginning and end
      if(msg[0] == SIGNATURE && msg[msg_pos - 1] == SIGNATURE) {
        // separate the message as delimitted by commas
        char *strings[NUMSENSORS*2 + 2];
        char *ptr = NULL;
        byte index = 0;
        ptr = strtok(msg,",");
        while(ptr != NULL) {
          strings[index] = ptr;
          index++;
          ptr = strtok(NULL, ",");
        }

        // change the stored MOSFET on/off to what was received in the
message
        for(int i = 1; i <= NUMSENSORS; i ++) {
          if(*strings[i] == '1') {fetOnOff[i-1] = true; }
          else {fetOnOff[i-1] = false; }
        } // end for

        //change the stored reference set point to what was received in the
message
        for(int i = NUMSENSORS + 1; i <= 2*NUMSENSORS; i ++) {
          flowSetPoints[i - NUMSENSORS - 1] = atof(strings[i]);
        }

        // send the currently stored data for MOSFET on/off, flow references,
and filtered flow
        sendData();
      } // end if
```

```
      // reset the message index to zero for the next message
      msg_pos = 0;
    } // end else-if
  } // end while
} // end loop()

/*
 * This function sends the currently stored data for the MOSFET on/off
 * states, flow references, and filtered flow, with each value
 * separated by a comma.
 */
void sendData() {
  Serial.print(signature);
  Serial.print(',');
  for(int i = 0; i < NUMSENSORS; i ++) {
    Serial.print(fetOnOff[i]);
    Serial.print(',');
  }
  for(int i = 0; i < NUMSENSORS; i ++) {
    Serial.print(flowSetPoints[i],3);
    Serial.print(',');
  }
  for(int i = 0; i < NUMSENSORS; i ++) {
    Serial.print(filteredData[i],3);
    Serial.print(',');
  }
  Serial.println(signature);
}
```

# Appendix B

# MATLAB Code

## TempSensorCurveFitting.m

```
%{
Auto-generated by MATLAB on 15-Oct-2021 16:01:42
Modified for use by Jonah Glunt
Penn State PAC Lab
Last modified: 30-Jan-2022

Creates an equation of best-fit for the
calibration data for Koolance SEN-AP008G.

Manufacturer's data can be found here:
https://koolance.com/files/products/manuals/Koolance_sen-ap008g_specifications.pdf
%}

clc, close all, clear all

% Load the saved data as an array
load tempsensordata;

% Parse the wanted information from the 6 columns provided
resistance = tempsensordata(:,3);
temp = tempsensordata(:,1);

% Ensure the data is prepared for the curve fitting tool.
% Checks that x and y data have same number of elements,
% converts all numbers to doubles, and gives warning if
% any of the values are NaN or Inf.
[xData, yData] = prepareCurveData( resistance, temp );

% Set up fittype and options.
% After trying various fits, the 'power2' was found to be the best fit.
ft = fittype( 'power2' );
opts = fitoptions( 'Method', 'NonlinearLeastSquares' );
opts.Display = 'Off';
opts.StartPoint = [85.059278087179 -0.473435648121428 4.61094049450347];

% Fit model to data.
[temp_fit, gof] = fit( xData, yData, ft, opts );

% Plot fit with data.
figure('name','R-T Fit for Koolance SEN-AP008G');
h = plot( temp_fit, xData, yData );
legend( h, 'Provided Data', 'Calculated Equation of Fit', 'Location', 'NorthEast', 'Interpreter', 'none' );
% Label axes
```

```
xlabel( 'Resistance [kΩ]', 'Interpreter', 'none' );
ylabel( 'Temperature [ºC]', 'Interpreter', 'none' );
title('Determining Calibration Equation for Temperature Sensor');
grid on

% Display equation on plot.
txt = {'Equation of best fit: y = a*x^b+c where', ['a = ', num2str(temp_fit.a)], ['b = ', num2str(temp_fit.b)], ['c
= ', num2str(temp_fit.c)]};
text(120,90,txt)
```

## Temp_Meas_Fcn_1.m

```
function y = temp_meas_fcn1(u,com)
%{
Jonah Glunt
Penn State PAC Lab
Last updated: 21 March 2022

This function is for serial communication with an Arduino measuring
temperature sensors. When Simulink calls this function, a message is sent
to the Arduino with Simulink's desired MOSFET on/off states. The Arduino
responds with its version of the states and all of the temperature
readings.

Input u: a 17x1 vector
Input com: number of COM port Arduino is connected to
Output y: a 32x1 vector containing Arduino's data
%}

% Create persistent serial port
persistent c

% Open serial port
if isempty(c)
    try
        c = serialport(['COM',num2str(com)],9600);
    catch
        fprintf('COM %d: Issue connecting to Arduino temperature board on com
port ',com)
    end
end

% Set default values for temp and on/off to be -1
Tmeas = -1*ones(16,1);
on_off_out = -1*ones(16,1);

% Check input is proper size
if ~isempty(u) && isequal(size(u),[17 1])
    % The first value in u is the time, the remaining 16 are MOSFET on/off
    time = u(1);
```

```
    on_off = round(u(2:17));

    % After enough time to initialize things
    if time>1
        % Send the serial port message, beginning and ending with signature
        msg = ['T' sprintf('%d',on_off) 'T'];
        writeline(c,msg);

        % If there is data on the serial port from the Arduino
        if c.NumBytesAvailable > 0
            % Read the message
            read = readline(c);
            % Separate the message by the delimitting commas
            split_msg = split(read, ',');
            % Check the message starts and ends with the Arduino's
            % signature
            if split_msg(1) ~= 't' || split_msg(end) ~= ['t' char(13)]
                fprintf('An error has occured in receiving complete set of
on/off information\n')
            else
                % Store the received on/off and temp values
                on_off_out = str2double(split_msg(2:17));
                Tmeas = str2double(split_msg(18:33));
                % Check that the received on/off values match those sent
                if sum(on_off_out == on_off) ~= 16
                    fprintf('The Arduino did not properly receive or change
to the new MOSFET on/off setting\n')
                end
            end
        else
            fprintf('MATLAB did not receive message from Arduino\n');
        end
    end
else
    fprintf('The on/off input was either empty, or the wrong type/size. The
input must be a 16x1 vector\n')
end

% Remove any values of NaN or inf from the received data
Tmeas(isinf(Tmeas))= -2;
Tmeas(isnan(Tmeas))= -2;

% Output values
y = [Tmeas;on_off_out];
end
```

**Flow_Meas_Fcn_1.m**

```
function output = flow_meas_fcn1(u, com) %, fetOnOff, flowReferences)
```

```
%{
Jonah Glunt
Penn State PAC Lab
Last updated: 21 March 2022

This function is for serial communication with an Arduino measuring
flow rate sensors. When Simulink calls this function, a message is sent
to the Arduino with Simulink's desired MOSFET on/off states and flow rate
references. The Arduino responds with its version of the states, the
refereces, and the measured and filtered flow rates.

Input u: a 13x1 vector
Input com: number of COM port Arduino is connected to
Output y: a 18x1 vector containing Arduino's data
%}

% Create a persistent serial port
persistent sp

% Open serial port
if isempty(sp)
    try
        sp = serialport(['COM',num2str(com)],9600);
    catch
        fprintf('COM %d: Issue connecting to Arduino on com port ',com)
    end
    %pause(1)
end

% Set default values for on/off, references, and flow rates to be -1
fetOnOffReceived = NaN(6,1);
flowRateReferencesReceived = NaN(6,1);
flowRates = NaN(6,1);

% Check input is proper size
if ~isempty(u) && isequal(size(u), [13 1])
    % The first value in u is the time
    time = u(1);
    % The next six are the MOSFET on/off
    fetOnOff = round(u(2:7));
    % The final six are the flow references
    flowReferences = round(u(8:13), 3);

    % After enough time has passed to initialize things
    if time > 1
        % Send the serial port message, beginning and ending with signature
        msg = ['F,' sprintf('%d,',fetOnOff) sprintf('%.3f,', flowReferences)
'F'];
        writeline(sp,msg);

        % If there is data on the serial port from the Arduino
        if sp.NumBytesAvailable > 0
            % Read the message
            read = readline(sp);
```

```
            % Separate the message by the delimitting commas
            split_msg = split(read, ',');
            % Check the message starts and ends with the Arduino's
            % signature
            if split_msg(1) ~= 'f' || split_msg(end) ~= ['f' char(13)]
                fprintf('An error has occured in receiving complete set of
information\n')
            else
                % Store the received data from the Arduino
                fetOnOffReceived = str2double(split_msg(2:7));
                flowRateReferencesReceived = str2double(split_msg(8:13));
                flowRates = str2double(split_msg(14:19));
                % Check that the received data matches those sent
                if sum(fetOnOffReceived == fetOnOff) ~= 6
                    fprintf('The Arduino did not properly receive or change
to the new MOSFET on/off settings\n')
                end
                if sum(flowRateReferencesReceived == flowReferences) ~= 6
                    fprintf('The Arduino did not properly receive or change
to the new flow reference settings\n')
                end
            end
        else
            fprintf('MATLAB did not receive message from Arduino\n');
        end
    end
else
    fprintf('The input was either empty, or the wrong type/size. The input
must be a 13x1 vector\n')
end

% Remove any values of NaN or inf from the received data
fetOnOffReceived(isinf(fetOnOffReceived)) = -1;
fetOnOffReceived(isnan(fetOnOffReceived)) = -1;
flowRates(isinf(flowRates)) = -1;
flowRates(isnan(flowRates)) = -1;
flowRateReferencesReceived(isinf(flowRateReferencesReceived)) = -1;
flowRateReferencesReceived(isnan(flowRateReferencesReceived)) = -1;

% Output values
output = [flowRates; fetOnOffReceived; flowRateReferencesReceived];
end
```

# Appendix C

## Calibration Data from Manufacturers

### Koolance SEN-AP008G (temperature sensor)

Sensor description can be found online at:

https://koolance.com/coolant-temperature-sensor-plug-10k-ohm

Calibration data can be found online at:

https://koolance.com/files/products/manuals/Koolance_sen-ap008g_specifications.pdf

### Koolance INS-FM18T10 (flowmeter and temperature sensor)

Sensor description can be found online at:

https://koolance.com/coolant-flow-meter-stainless-steel-with-temperature-sensor-sen-fm18t10

The included temperature sensor is the same as the SEN-AP008G.

Calibration data for the flowmeter can be found online at:

https://koolance.com/files/products/manuals/manual_ins-fm18_d140eng.pdf

# BIBLIOGRAPHY

[1] A. Lazrak, J.-F. Formiqué, and J.-F. Robin, "An Innovative Practical Battery Thermal Management System Based on Phase Change Materials: Numerical and Experimental Investigations," *Applied Thermal Engineering*, vol. 128, pp. 20–32, Jan. 2018.

[2] G. Xia, L. Cao, and G. Bi, "A Review on Battery Thermal Management in Electric Vehicle Application," *Journal of Power Sources*, vol. 367, pp. 90–105, Nov. 2017, doi: 10.1016/j.jpowsour.2017.09.046.

[3] S. R. T. Peddada, D. R. Herber, H. C. Pangborn, A. G. Alleyne, and J. T. Allison, "Optimal Flow Control and Single Split Architecture Exploration for Fluid-Based Thermal Management," *Journal of Mechanical Design, Transactions of the ASME*, vol. 141, no. 8, pp. 1–12, Aug. 2019, doi: 10.1115/1.4043203.

[4] J. P. Koeln, H. C. Pangborn, M. A. Williams, M. L. Kawamura, and A. G. Alleyne, "Hierarchical Control of Aircraft Electro-Thermal Systems," *IEEE Transactions on Control Systems Technology*, vol. 28, no. 4, pp. 1218–1232, 2020, doi: 10.1109/TCST.2019.2905221.

[5] S. R. Reddy, M. A. Ebadian, and C. X. Lin, "A Review of PV-T Systems: Thermal Management and Efficiency with Single Phase Cooling," *International Journal of Heat and Mass Transfer*, vol. 91. 2015. doi: 10.1016/j.ijheatmasstransfer.2015.07.134.

[6] H. C. Pangborn, J. E. Hey, T. O. Deppen, A. G. Alleyne, and T. S. Fisher, "Hardware-in-the-Loop Validation of Advanced Fuel Thermal Management Control," *Journal of Thermophysics and Heat Transfer*, vol. 31, no. 4, pp. 901–909, Oct. 2017, doi: 10.2514/1.T5055.

[7] J. Pastor and J. M. Menaud, "SeDuCe: A Testbed for Research on Thermal and Power Management in Datacenters," *26th International Conference on Software, Telecommunications and Computer Networks (SoftCOM), Split-Supetar Croatia, Sept. 13-15, 2018*, Sep. 2018, doi: 10.23919/SOFTCOM.2018.8555773.

[8] G. P. Huang *et al.*, "Dimensional analysis, modeling, and experimental validation of an aircraft fuel thermal management system," *Journal of Thermophysics and Heat Transfer*, vol. 33, no. 4, pp. 983–993, 2019, doi: 10.2514/1.T5660.

[9] M. Krieger, D. Stecklair, S. Peluso, and S. Stockar, "Design and Verification of a Small-Scale District Heating Network Experiment," *Proceedings of the ASME 2019 Conference, Park City, Utah, Oct. 8-11, 2019*, [Online]. Available: http://asmedigitalcollection.asme.org/DSCC/proceedings-pdf/DSCC2019/59155/V002T23A002/6455398/v002t23a002-dscc2019-9101.pdf

[10] H. Pangborn, "Hierarchical Control for Multi-Domain Coordination of Vehicle Energy Systems with Switched Dynamics," Ph.D. dissertation, Mechanical Engineering, University of Illinois at Urbana-Champaign, Illinois, 2019.

[11] "CompactDAQ Systems." https://www.ni.com/en-us/shop/compactdaq.html (accessed Mar. 21, 2022).

[12] "MicroLabSpace - dSpace." https://www.dspace.com/en/inc/home/products/hw/microlabbox.cfm (accessed Apr. 05, 2021).

[13] "Arduino Mega 2560 REV3." https://store.arduino.cc/usa/mega-2560-r3?gclid=CjwKCAjwjbCDBhAwEiwAiudBy9ggl07cZQgUiGXC4hWiYVgyJU5ZPuynOTOzgwSCYQjy4j3D5rYXDhoCawAQAvD_BwE (accessed Apr. 05, 2021).

[14] “INS-FM17N Coolant Flow Meter.” https://koolance.com/ins-fm17n-coolant-flow-meter (accessed Mar. 20, 2022).

[15] “SEN-FM18T10 Coolant Flow Meter with Temperature Sensor, 10K Ohm.” https://koolance.com/coolant-flow-meter-stainless-steel-with-temperature-sensor-sen-fm18t10 (accessed Mar. 20, 2022).

# ACADEMIC VITA

**JONAH GLUNT**

## EDUCATION

**Pennsylvania State University**, University Park, PA
*Mechanical Engineering*, 2018-2022
*Minors in Chemistry, Piano Performance*

## RESEARCH EXPERIENCE

**Pangborn Advanced Controls Laboratory**, University Park, PA
*Undergraduate Researcher*, 2021-2022
In completion of Schreyer Honors Thesis.

## WORK EXPERIENCE

**SMS-Group**, Pittsburgh, PA
*Engineering Sales Intern – Flat Rolling Division*, Summer 2021

**Michael Baker International**, Pittsburgh, PA
*Civil/Environmental Engineering Intern*, Summer 2019

## AWARDS AND HONORS

**2021** Polymathic Studies Scholarship for the College of Engineering
Jury Recognition and Honors Awards for Piano Performance
Selected Performer in Empow(h)er and African American Music Festivals
Penn State Glee Club Bruce Trinkley Award for Artistry
**2020** Paul Morrow Endowed Scholarship in the College of Engineering
Jury Recognition Award for Piano Performance
**2019** Hallowell Scholarship for the College of Engineering
Jury Recognition and Honors Awards for Piano Performance
**2018** Hallowell Scholarship for the College of Engineering
Pennsylvania State University President's Freshman Award