Low-Cost Instrumentation Development Using Independent, Self-Contained Microcontrollers

By

Susan Richmond

Master's Thesis

(Geological Engineering)

University of Wisconsin – Madison

2019

Abstract

This thesis describes the development and testing of low-cost geophysical and field instrumentation using microcontrollers. While open-source microcontroller boards have traditionally been developed for hobbyists and artists, the quality of modern electronics and associated sensors has opened a significant number of opportunities for their use in the collection of research-quality data. During my master studies, I have designed, programmed, tested and implemented microcontroller-based systems to digitize an analog electromagnetic instrument, implement a sensor array for field monitoring of physical/chemical properties of water, develop seismic sensor arrays for both active and passive sensing of vibrations, and construct a lab-scale electrical resistivity tomographer. For all these developments, I considered the capabilities of different boards, shields, and sensors, including microcontroller and ADC sampling rates and resolution, sensor sensitivity, microcontroller-sensor communications, and data formatting and storage. The results of my studies are a suite of tools that can be deployed both in the field and in the laboratory for both educational and research purposes. Following the spirit of the open-source community, this thesis documents the step-by-step process for the selection of all components, the programs used for controlling the boards and sensors, and discusses the limitations of the developed systems. I hope that my efforts will help in engaging a much larger community in near-surface geophysical education and research activities.

Acknowledgements

Thank you to my advisor, Dante Fratta, for all his help and support the past two years,

to Dave Hart and Bill Likos for serving on my committee,

to Zach Fiscus for being a constant source of comfort and support,

to my family for all their love and support over the years,

and the Wisconsin Department of Transportation for supporting my studies.

Table of Contents

List of Tables

List of Figures

1 Introduction

Many types of instrumentation are commonly used to perform meaningful research in engineering. Unfortunately, professionally developed and engineered systems are typically expensive, which can make them inaccessible to people without financial resources, such as those in developing countries. As an alternative to these professional, but expensive instruments, we have developed or improved four sets of instruments using microcontrollers that are much more affordable and can perform many of the same applications as the more traditional, professional systems.

Electromagnetic sensors can be used to perform near surface imaging of water content, layer distributions, and contaminants. Using a microcontroller, we were able to upgrade an old, analog electromagnetic sensor (EM-31 from Geonics Limited) to a digital system. Our system records the ground conductivity data from the EM-31, along with the GPS coordinates and real time of each measurement.

In order to perform rapid environmental assessment of streams and lakes, we used a microcontroller to develop our own water quality monitoring system. This system uses a GPS module and several different probes to record several different geographically located water quality properties and parameters. These properties and parameters include temperature, fluid conductivity, pH, dissolved oxygen, nitrate concentration, chloride concentration, and a rough approximation of turbidity.

Seismic sensors can be used for a wide variety of applications, including vibration monitoring, near-surface seismic imaging, and horizontal to vertical spectral ratio (HVSR) techniques. Our system uses a microcontroller paired with a 24-bit analog to digital converter (ADC) to record seismic signals from three orthogonal axes with high precision and sampling

rates up to 250 Hz. In addition, a GPS module is used to record the location (GPS coordinates) of the sensor and to correlate the recorded signals to real time so that signals may be meaningfully compared between separate sensors.

Finally, we developed our own electrical resistivity sensor, which can be used for laboratory environmental and hydrogeological imaging. Our sensor uses measurements of current supplied to and potential difference across different electrodes to calculate apparent resistivity. In addition, our system automatically switches between electrode pairs to complete measurements for a 16-electrode array and outputs a file that is ready to be used with Res2Dinv software (from Geotomo Software) which can be used to analyze the data.

2 Introduction to Microcontrollers and Sensors

2.1 How Microcontrollers Work

Microcontrollers are devices that contain all the functions of a computer integrated onto one chip (Davies, 2008; Bolanakis, 2018). They were originally developed from microprocessors but made to increase reliability and simplicity by integrating functions on the same chip as the processor, rather than relying on other components to provide those functions. There are several differences between microprocessors and microcontrollers, but one of the most important is the purpose for which they are designed. Microprocessors are typically designed to increase computing power while microcontrollers are designed to control other systems, require low power, and are relatively inexpensive. Another important difference is that microcontrollers can have many functions built into them that microprocessors require additional components to perform (Davies, 2008).

Typical microcontrollers contain at least the following functions: central processing unit (CPU), memory, input and output ports, address and data buses, and a clock. These functions are connected and communicate to one another as shown in Figure 2.1.1. The CPU acts as the brain of the system, controlling the other functions as programmed (Davies, 2008; Bolanakis, 2018). The CPU communicates with the other components through the address and data buses, sending either instructions or data to be stored (Davies, 2008). The CPU receives its instructions from the downloaded program stored in the nonvolatile (read-only) memory (or ROM) and uses the random-access memory (or RAM) to store data. The clock is used to ensure all the functions remain synchronized with one another and instructions are performed at the appropriate times. Lastly, the input and output ports allow the CPU to communicate with external systems (Davies, 2008; Bolanakis, 2018).

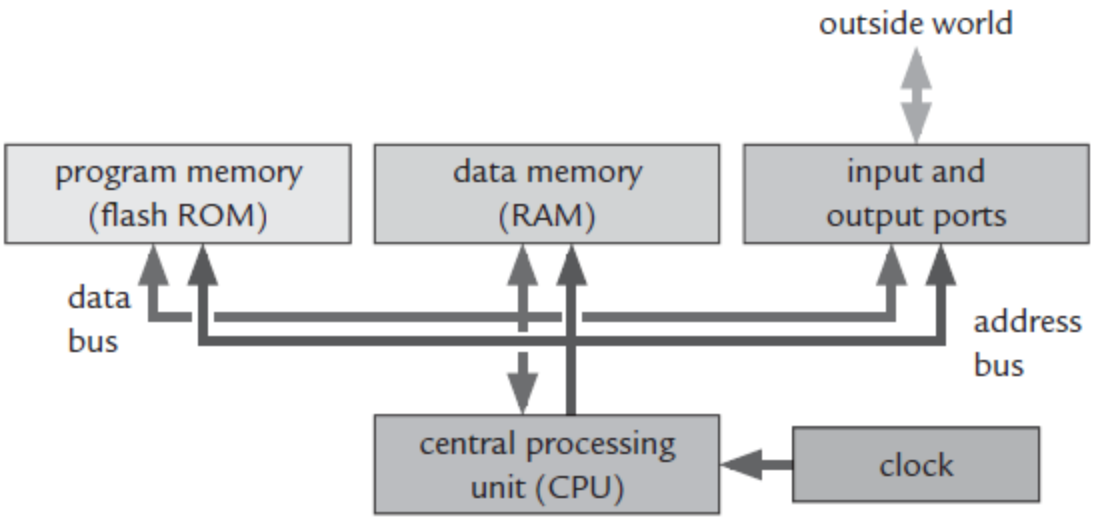


Figure 2.1.1: Simple schematic showing the essential functions of a microcontroller (Davies, 2008).

In addition to these basic functions, microcontrollers may also include several optional functions. These commonly include timers, an analog-to-digital converter, and/or a real-time clock. It is also common to include communication interfaces (e.g., serial peripheral interface (SPI), inter-integrated circuit (I^2C), universal serial bus (USB), etc.) that can be used to connect with and control other devices (Davies, 2008; Bolanakis, 2018). In addition, a monitor, background debugger, and embedded emulator may be included to allow the microcontroller to communicate with a computer and download the program it is meant to run (Davies, 2008). Several different companies offer a wide selection of microcontrollers with many of these optional functions. In addition, many of these companies develop products that are relatively inexpensive, making them an excellent choice for developing low-cost instrumentation alternatives that can be utilized by many.

Arduino and PJRC are two such companies. One of the biggest advantages to using Arduino microcontrollers is that Arduino provides many resources and guides for beginners, they have developed a large community for sharing ideas and problems, and all their software and

hardware is open source. This makes it very easy to use Arduino even without having any experience with programming. In addition, because Arduino is open source, there is an extensive collection of codes and libraries that can be found online for a wide variety of applications and devices. Often these codes and libraries can be used as a foundation when designing a project, making it even easier to develop a code to meet the needs of the design.

While Arduino does offer several microcontrollers, they are relatively limited in their capabilities and may not always be suitable for very advanced projects. Fortunately, other brands offer microcontrollers with even greater specifications, such as PJRC which produces Teensy microcontrollers. Just like Arduino, PJRC has many resources, guides, a large community, and is open source. A major advantage of Teensy microcontrollers is that an add-on for the Arduino software, Teensyduino, exists which allows them to use codes that were written for Arduino microcontrollers. This allows all the resources available for Arduino to be used with Teensy microcontrollers as well.

2.2 Capabilities and Specifications

One of the most basic microcontrollers is the Arduino Uno microcontroller. The Arduino Uno microcontroller provides a good option when advanced, additional features are not needed. It uses an ATmega328P microcontroller which has a clock speed of 16 MHz and 32 kB of flash memory. It also has an additional 2 kB of SRAM and 1 kB of EEPROM. It accepts an input voltage of 6 to 20 V (with recommended input of 7 to 12 V), which can be supplied from either a USB connection or through an external power supply, and functions with an operating voltage of 5 V. The Arduino Uno microcontroller has 14 digital pins, 6 of which are capable of PWM output, and 6 analog pins with an ADC with 10 bits of resolution. Some of these pins can also serve additional functions, including one each of hardware UART serial, SPI, and I^2C

communication and two external interrupts (Arduino, 2019b). Figure 2.2.1 shows the Arduino Uno microcontroller with some of these features highlighted.

Another useful microcontroller offered by Arduino is the Arduino Mega 2560 microcontroller. The Arduino Mega 2560 microcontroller has some similarities to the Arduino Uno microcontroller but is designed for more advanced projects and offers the capability of many more connections, both digital, analog, and for communication. It uses an ATmega2560 microcontroller, which has the same 16 MHz clock speed as the Arduino Uno microcontroller's, but has more memory: 256 kB of flash memory, 8 kB of SRAM, and 4 kB of EEPROM. It also accepts the same input voltage and has the same operating voltage as the Arduino Uno microcontroller. However, it has 54 digital pins compared to the Arduino Uno microcontroller's 14. Fifteen of the Arduino Mega 2560 microcontroller's digital pins are capable of PWM output.

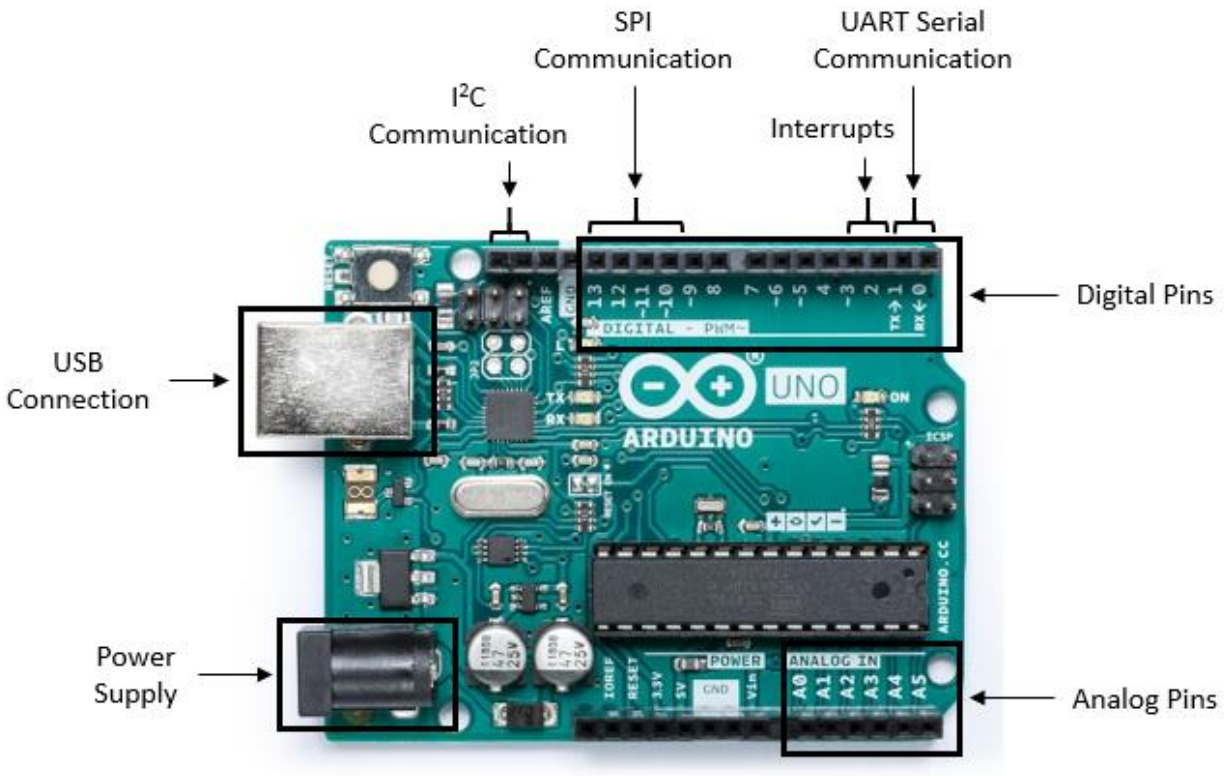


Figure 2.2.1: The Arduino Uno microcontroller and its connections (Arduino, 2019b).

It has 16 analog pins, which is more than the Arduino Uno microcontroller, but an ADC with the same resolution as the Arduino Uno microcontroller's. In addition, it is capable of four hardware UART serial communications, one SPI communication, one I^2C communication, and six interrupts (Arduino, 2019a). Figure 2.2.2 shows the Arduino Mega 2560 microcontroller with some of these features highlighted.

One microcontroller offered by PJRC is the Teensy 3.6 microcontroller, which not only offers greater specifications than the Arduino Mega 2560 microcontroller, but a greater capacity for connections as well. The Teensy 3.6 microcontroller uses the MK66FX1M0VMD18 microcontroller, which has a 180 MHz processor, 1 MB of flash memory, 256 kB of RAM, and 4 kB of EEPROM, which far exceeds the Arduino Mega 2560 microcontroller. The Teensy 3.6 microcontroller accepts an input voltage of 3.6 to 6.0 V and functions with an operating voltage of 3.3 V, rather than the 5 V that Arduino typically uses. Unlike the Arduino microcontrollers,

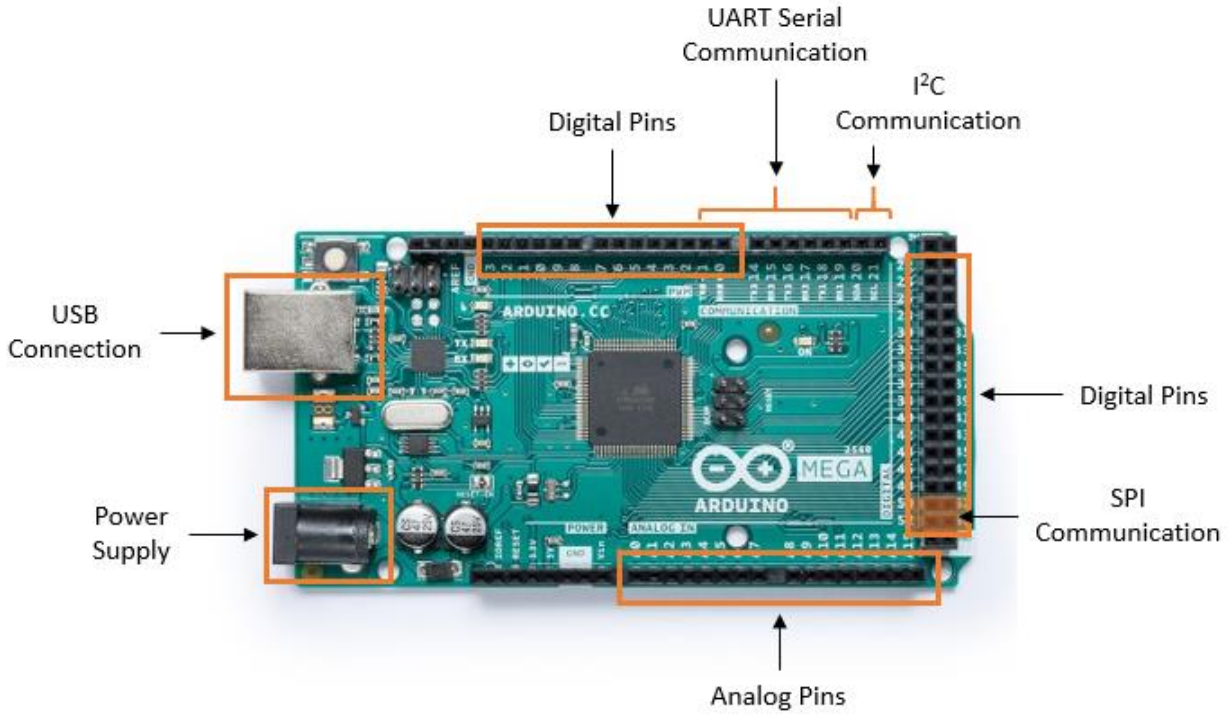


Figure 2.2.2: The Arduino Mega 2560 microcontroller and its connections (Arduino, 2019a).

the pins on the Teensy 3.6 microcontroller can perform several different functions, rather than one or two. This means that while the Teensy 3.6 microcontroller is capable of the following functions, it is not necessarily capable of performing them simultaneously. The Teensy 3.6 microcontroller has 62 digital pins which can all be used as interrupts and 22 of those pins can provide PWM output. The Teensy 3.6 microcontroller also has 25 analog input pins with two separate ADCs of 13-bit resolution and 2 analog output pins with digital to analog converters of 12-bit resolution. It is capable of six hardware UART serial communications, three SPI communications, and four I^2C communications. In addition, it also comes with a real-time clock, a built-in micro SD card port, and more (SparkFun Electronics, 2019c). Figure 2.2.3 shows the Teensy 3.6 microcontroller with the different functions each of the pins is capable of highlighted. The capabilities of all the microcontrollers described above are compared in Table 2.2.1.

Table 2.2.1: Comparison of microcontroller specifications (Arduino, 2019a; Arduino, 2019b; SparkFun Electronics, 2019c).

	Arduino Uno	**Arduino Mega 2560**	**Teensy 3.6**
Microcontroller	ATmega328P	ATmega2560	MK66FX1M0VMD18
Clock Speed	16 MHz	16 MHz	180 MHz
Flash Memory	32 kB	256 kB	1 MB
SRAM	2 kB	8 kB	256 kB
EEPROM	1 kB	4 kB	4 kB
Operating Voltage	5 V	5 V	3.3 V
Digital Pins	14	54	62
PWM Output Pins	6	15	62
Analog Pins	6	16	25 input; 2 output
Resolution	10-bit	10-bit	13-bit input; 12-bit output
Hardware UART Serial Communication	1	4	6
SPI Communication	1	1	3
I^2C Communication	1	1	4

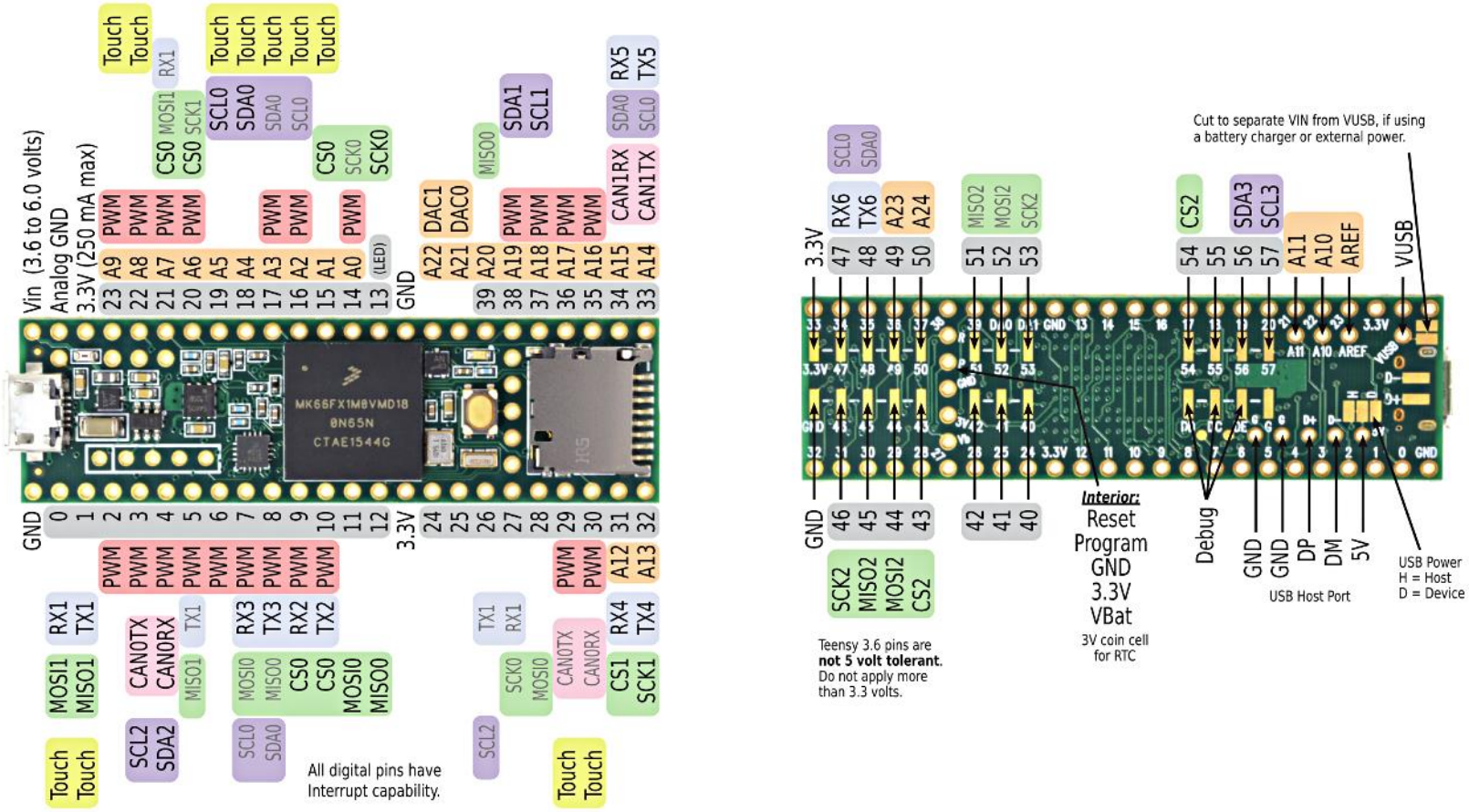


Figure 2.2.3: Pin assignments for the Teensy 3.6 microcontroller: gray – digital pins, orange – analog pins, blue – hardware UART, green – SPI, and purple – I^2C. (PJRC, 2019).

2.3 Sensor Physical Principles and Operations/Specifications

Sensors are devices that measure a physical variable, often by converting that physical variable to an electrical signal that can be easily measured and recorded (Regtien et al., 2004; Wilson, 2005; Yoon, 2016). In the following section, the basic principles behind several different types of sensors are described.

2.3.1 Electromagnetic Methods

Electromagnetic induction instruments use electromagnetic waves to measure the conductivity of the subsurface (Telford et al., 1990). Sediment and rock type, porosity, grain size, clay content, fractures, water quality, and presence of contaminants all influence the electrical conductivity of the subsurface, making electromagnetic instruments well suited for determining the depth to the water table, detecting the presence of contaminants, groundwater modelling, and geologic mapping. In addition, the fact that electromagnetic instruments are noninvasive and do not require direct contact with the ground allows for data to be collected easily and quickly, with no need to manage long cables (Fitterman and Labson, 2005).

The behavior of electromagnetic waves is governed by Maxwell's equations and electromagnetic induction instruments take advantage of that behavior (Telford et al., 1990; Fitterman and Labson, 2005). These instruments typically consist of an alternating current (AC) power source, a transmitter coil, a receiver coil, and some form of indicator to provide measurement readings (Telford et al., 1990). The transmitter sends an AC signal through a coil, which, according to Ampere-Maxwell's law, induces a time-varying magnetic field, called the primary field. Ampere-Maxwell's law states that an electric current or a time-varying electric field induces a magnetic field according to the following equation (Telford et al., 1990; Fitterman and Labson, 2005).

$$\Delta \times \boldsymbol{H} = \boldsymbol{J} + \frac{\partial \boldsymbol{D}}{\partial t} \quad (2.3.1)$$

Where $\boldsymbol{H}$ is the magnetic field intensity, $\boldsymbol{J}$ is the current density, $\boldsymbol{D}$ is the electric displacement, and t is time. According to this equation, the induced magnetic field will have a magnetic field strength that is proportional to and in phase with the electric current.

Because the transmitter sends an AC signal into the coil, the generated primary magnetic field varies in time. According to Faraday-Lenz's law, the time-varying magnetic field induces an electric field in the subsurface. The following equation describes Faraday-Lenz's law (Telford et al., 1990; Fitterman and Labson, 2005).

$$\Delta \times \boldsymbol{E} = -\frac{\partial \boldsymbol{B}}{\partial t} \quad (2.3.2)$$

Where $\boldsymbol{E}$ is the electric field intensity and $\boldsymbol{B}$ is the magnetic induction. This equation indicates that a time-varying magnetic field induces an electric field with a field strength proportional to the rate of change of the magnetic field. Since the electric field is proportional to the rate of change of the magnetic field, rather than the magnetic field itself, there is a shift in phase between the two fields. Note that an electric field and an electric current are related by the following equation for homogeneous, isotropic media (Telford et al., 1990; Fitterman and Labson, 2005).

$$\boldsymbol{J} = \sigma \boldsymbol{E} \quad (2.3.3)$$

Where σ is electrical conductivity. This means that if an electric field is induced then an electric current with the same phase is induced as well. Following Ampere-Maxwell's law, the induced current induces an additional magnetic field. This additional field is called the secondary magnetic field. The secondary magnetic field is in phase with the induced current but shifted in phase from the primary field (Telford et al., 1990; Fitterman and Labson, 2005).

Both the primary and secondary magnetic fields interact with the receiver coil. Since they both vary with time and follow Faraday-Lenz's law, they induce an electric field, and therefore a current, in the receiver coil. The current can be related to the current density using the cross-sectional area the current passes through in the following integration (Telford et al., 1990).

$$I = \int \boldsymbol{J}\, d\boldsymbol{A} \tag{2.3.4}$$

Where I is the current and $d\boldsymbol{A}$ is the cross-sectional area the current passes through. By inducing a current, both fields also generate a voltage in the coil, which can be related to current through Ohm's law.

$$V = IR \tag{2.3.5}$$

Where V is the voltage and R is the resistance. The measured voltage is also related to the electric or magnetic fields through the following equation (Fitterman and Labson, 2005).

$$V = \int \boldsymbol{E} \cdot d\boldsymbol{s} = -\frac{d}{dt}\int \boldsymbol{B} \cdot \boldsymbol{n} dA = -\mu_0 \int \frac{d}{dt}\boldsymbol{H} \cdot \boldsymbol{n} dA \tag{2.3.6}$$

Where $d\boldsymbol{s}$ is the segment length, $\boldsymbol{n}$ is the vector component normal to the receiver loop, μ_0 is the magnetic permeability of free space ($4\pi \cdot 10^{-7}\ H/m$), and the integral is integrated along the length of either the dipole (for an electric field) or the loop (for a magnetic field). The receiver records the voltage generated by both fields and the influence of the secondary field causes the recorded signal to differ from the original, transmitted signal by a difference in amplitude, phase, or both (Telford et al., 1990). This difference can be used to obtain information about the subsurface.

Typically, the secondary field is very weak relative to the primary field, which can lead to the signal from the primary field completely overshadowing the signal from the secondary field in the receiver coil. Since the secondary field contains the information about the subsurface, many instruments use some method of cancelling the influence of the primary field on the

receiver coil so the signal from the secondary field can be measured directly. One method of doing this is to use an artificial signal in the receiver coil with the same frequency and amplitude, but opposite phase as the transmitted signal so it completely cancels the signal from the primary field (Telford et al., 1990).

Electromagnetic instruments do not measure the conductivity directly, rather, they measure the voltage produced in the receiver coil and relate this voltage to an apparent conductivity. The apparent conductivity is the conductivity that a homogenous half space would need to have for the instrument to measure the same conductivity as the real system using the same frequency and intercoil spacing. The apparent conductivity that is measured depends on the geometry and conductivity of the subsurface as well as the frequency, coil geometry, and intercoil spacing of the instrument. For example, for coils oriented in a horizontal, coplanar geometry where the skin depth is much greater than the intercoil spacing, the apparent conductivity can be determined with the following equation (Fitterman and Labson, 2005).

$$\sigma_a = \frac{4}{\mu_0 \omega r^2} Q\left[\frac{Z}{Z_0}\right] \tag{2.3.7}$$

Where σ_a is the apparent conductivity, $\omega = 2\pi f$ is the angular frequency, r is the intercoil spacing, $Q[\,]$ indicates the quadrature (or imaginary) component, Z is the mutual impedance (the ratio of the voltage in the receiver coil produced by the current flowing in the transmitter coil divided by the current flowing in the transmitter coil), and Z_0 is the mutual impedance of free space. Once the apparent conductivity is determined using an appropriate equation, various equations and characteristic curves that have been calculated for simplified subsurface geometries can be used to determine the structure, location, and electrical conductivities that the subsurface or anomalies are composed of (Telford et al., 1990).

The EM-31 by Geonics can measure apparent conductivity for several different ranges, including 3, 10, 30, 100, 300, and 1,000 mS/m. It has an accuracy of +5 percent at 20 mS/m and a precision of +2 percent of the full scale. It uses an operating frequency of 9.8 kHz, has a fixed intercoil spacing of 3.66 m, and noise levels of 0.1 mS/m (Geonics Limited, 1991).

2.3.2 Thermocouple

There are several different types of sensors that can be used to measure temperature, the most common of which include resistive temperature sensors, thermistors, and thermocouples. These instruments rely on the fact that the resistance of metals and semiconductors varies with temperature. As temperature increases, the resistance of metals increases and the resistance of semiconductors decreases. Resistive temperature sensors and thermistors use these relationships to determine temperature by measuring the resistance of a metal or semiconductor, respectively (Regtien et al., 2004; Ripka and Tipek, 2007; Yoon, 2016). Thermocouples rely on the Seebeck effect, which describes the behavior of two dissimilar metal conductors connected at two junctions exposed to different temperatures. In this configuration, a potential difference is generated that is proportional to the temperature difference between the two junctions. Thermocouples use this relationship to determine the temperature (Pollock, 1971; ASTM, 1981; Regtien et al., 2004; Ripka and Tipek, 2007; Yoon, 2016).

Each of these temperature sensors has different advantages and disadvantages. Typical response curves for resistive temperature sensors, thermistors, and thermocouples are shown in Figure 2.3.1. Thermistors are the most sensitive, but do not have a linear response and have a smaller temperature range (Regtien et al., 2004; Ripka and Tipek, 2007; Yoon, 2016). While the temperature range may be smaller, thermistors are typically valid for temperatures ranging from -100 to 350 ºC, which is large enough for many applications (Regtien et al., 2004).

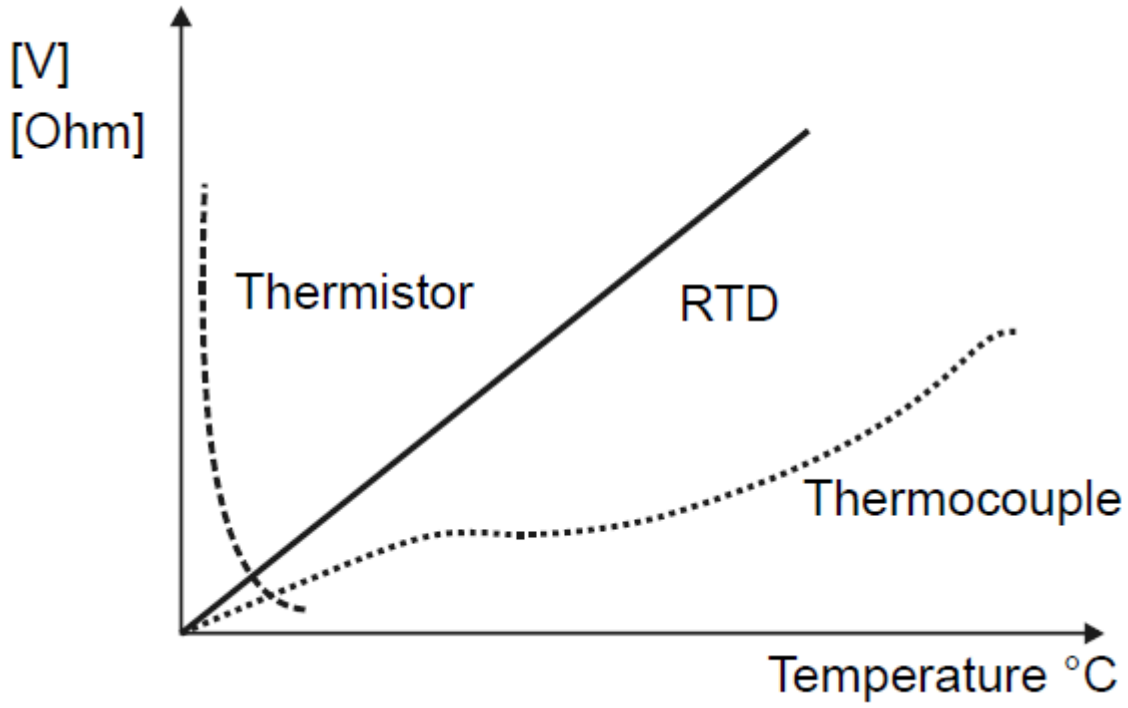


Figure 2.3.1: Response curves for common types of temperature sensors (Ripka and Tipek, 2007).

Resistive temperature sensors are not as sensitive as thermistors, but they have a linear response and are more accurate (Regtien et al., 2004; Ripka and Tipek, 2007). Thermocouples are the least sensitive, but they are fairly linear at the lower end of their temperature range (up to about 1500 ºC; Yoon, 2016) and are often the most commonly used because of their simplicity and versatility (Regtien et al., 2004; Ripka and Tipek, 2007; Yoon, 2016).

A thermocouple is composed of two different metal semiconductors connected at two junctions to form a circuit (Pollock, 1971; ASTM, 1981; Regtien et al., 2004; Ripka and Tipek, 2007; Yoon, 2016). Two thermal effects describe the changes in heat that occur within this circuit. The Peltier effect describes the change in heat at the junctions while the Thomson effect describes the change in heat within each of the conductors (Pollock, 1971). According to the Peltier effect, when an electric current crosses a junction of two different metals, heat

proportional to the current will either be absorbed or released, depending on the direction of the current (Pollock, 1971; ASTM, 1981; Ripka and Tipek, 2007). In addition, the Thomson effect says that when both heat and electric current flow in a conductor, heat will either be absorbed or released within the conductor depending on whether the heat and current are flowing in the same or opposite directions. When added together, these two thermal effects combine to form the electrical Seebeck effect (Pollock, 1971). The Seebeck effect indicates that when the two junctions of a thermocouple are held at different temperatures, a potential difference will develop, and a current will be induced (Pollock, 1971; ASTM, 1981; Yoon, 2016).

For any two conductors, the measured potential difference can be related to the temperature of the two junctions according to the following equation (Pollock, 1971; Ripka and Tipek, 2007).

$$V = C_1(T_1 - T_2) - C_2(T_1^2 - T_2^2) \tag{2.3.8}$$

Where V is the voltage, T_1 and T_2 are the temperatures of the two junctions, and C_1 and C_2 are constants that depend on the two metals forming the thermocouple. If the metals are correctly chosen, C_2 becomes very small and the relationship between voltage and temperature difference becomes nearly linear. Only combinations of metals that meet this condition are typically used to make thermocouples (Pollock, 1971). Table 2.3.1 shows the types of metals used in the most common thermocouples along with typical sensitivities and ranges.

For most thermocouples, the relationship between voltage and temperature can be simplified as the linear approximation of the previous equation.

$$V = C(T_1 - T_2) \tag{2.3.9}$$

However, even with this simplification, the constant, C, and the temperature of one junction must be known to determine the temperature at the other junction based on the measured voltage. The

Table 2.3.1: Composition and typical sensitivity and range for common types of thermocouples (Regtien et al., 2004).

Thermocouple Type	Composition	Sensitivity (µV/K)	Range (ºC)
K (chromel-alumel)	90% Ni + 10% Cr 94% Ni + 2% Al + rests	39 (at 0 ºC)	-184 to 1260
J (iron-constantan)	Fe 60% Cu + 40% Ni	45 (at 0 ºC)	-210 to 760
E (chromel-constantan)	90% Ni + 10% Cr 60% Cu + 40% Ni	60 (at 0 ºC)	-200 to 900
S (platinum-rhodium)	Pt 90% Pt + 10% Rh	10 (at 1000 ºC)	-50 to 1600
R (platinum-rhodium)	Pt 87% Pt + 13% Rh	14 (at 1600 ºC)	-50 to 1600
B (platinum-rhodium)	70% Pt + 30% Rh 94% Pt + 6% Rh	10 (at 1600 ºC)	0 to 1800
T (copper-constantan)	Cu 60% Cu + 40% Ni	40 (at 0ºC)	-200 to 400

constant can be determined based on the thermocouple type and calibration, however, to know the temperature at one of the junctions, the temperature at that junction must be kept constant at a known value. This is typically how thermocouples are used: one junction, called the reference junction, is kept at a constant, known temperature (typically 0 ºC) while the other is used to measure temperature (Pollock, 1971; Regtien et al., 2004; Ripka and Tipek, 2007).

Analog Devices' K type thermocouple amplifier (AD8495) can be used to interface a K type thermocouple with a microcontroller. The K type thermocouple amplifier is capable of measuring temperatures ranging from -250 to 750 °C with an accuracy of +/- 2 °C. It also applies cold junction compensation to reduce errors caused by fluctuations in temperature at the reference junction. It has a response time of 99.9 percent of the total response in only 40 µs.

Temperature can be determined from the measured voltage using the following equation (Analog Devices, 2011).

$$T = \frac{V_{out} - V_{ref}}{0.005 \frac{V}{°C}} \quad (2.3.10)$$

Where T is the temperature in °C, V_{out} is the measured voltage, and V_{ref} is the voltage at the reference pin. Alternatively, if calibration is desired, a two-point calibration can be used to determine the linear relationship relating measured voltage and temperature. When used with the glass braid insulated K type thermocouple with a bare wire tip from SparkFun Electronics (SEN-00251 ROHS), the temperature range that can be measured becomes limited to -73 to 482 °C, otherwise the glass braid insulation may become damaged (SparkFun Electronics, 2019b).

2.3.3 Electrical Conductivity Probe

The electrical conductivity of a solution is determined by sending a current through that solution and measuring its conductance (Radiometer Analytical SAS, 2004; Down and Lehr, 2005). When two electrodes are immersed in a solution and an electric current is applied, the ions in solution are attracted to the positively and negatively charged electrodes. This attraction causes cations to move towards the negatively charged electrode and anions to move toward the positively charged electrode, allowing the current to pass through the solution. By measuring the applied current and the voltage drop that occurs across the electrodes, the resistance of the solution can be determined from Ohm's law. Then the conductance of the solution, which is inversely proportional to its resistance, can be calculated with the following equation (Radiometer Analytical SAS, 2004; Regtien et al., 2004).

$$G = \frac{1}{R} \quad (2.3.11)$$

Where G is conductance. The measured conductivity is dependent on both the conductance of the solution and the geometry of the two electrodes used to measure it. For a specific conductivity meter, a cell constant is defined according to the following equation, describing the geometry of its electrodes (Radiometer Analytical SAS, 2004; Regtien et al., 2004; Down and Lehr, 2005).

$$K = \frac{l}{A} \tag{2.3.12}$$

Where K is the cell constant, l is the distance between the two electrodes, and A is the area of the electrodes. Using the appropriate cell constant, the electrical conductivity of the solution can be calculated from the measured conductance with the following equation (Radiometer Analytical SAS, 2004; Regtien et al., 2004).

$$\sigma = GK \tag{2.3.13}$$

In this way, electrical conductivity instruments can measure the conductivity of a solution.

When considering the design of a conductivity sensor, the effects of polarization at the electrodes must be considered (Radiometer Analytical SAS, 2004; Down and Lehr, 2005). Polarization occurs at the electrodes as the ions attracted to each of the electrodes begin to accumulate. This results in the development of a resistance at each of the electrodes. This means that the potential difference measured across the two electrodes of the sensor corresponds to the sum of the resistances of the solution and the polarization of the electrodes, as depicted in the representative circuit in Figure 2.3.2 (Radiometer Analytical SAS, 2004). Unfortunately, this means that the measured conductance will not correspond to the conductivity of the solution, as desired (Radiometer Analytical SAS, 2004; Regtien et al., 2004). An AC can be used to reduce the effects of polarization (Radiometer Analytical SAS, 2004). The difference in polarization resistance caused by AC and direct current (DC) is depicted in Figure 2.3.3. Using AC allows the current to flow through the double layer capacitance of the

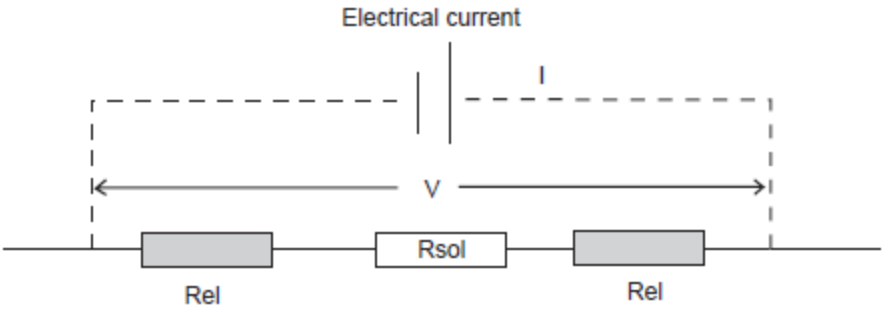


Figure 2.3.2: Circuit representing the components of a two-electrode electrical conductivity sensor (Radiometer Analytical SAS, 2004).

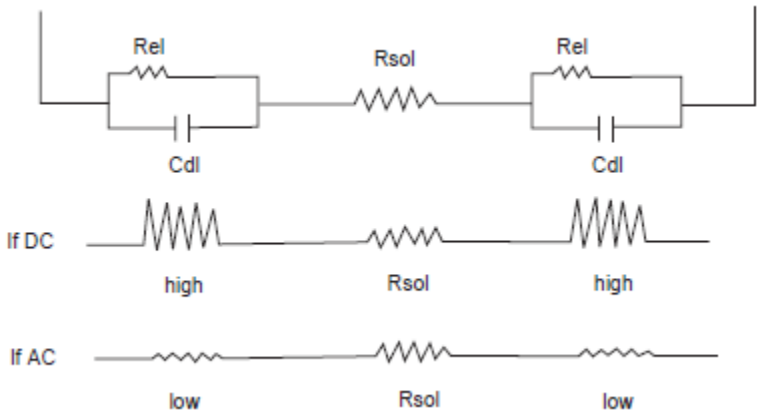


Figure 2.3.3: Effect of polarization caused by AC and DC (Radiometer Analytical SAS, 2004).

electrodes, rather than contributing to their resistance (Radiometer Analytical SAS, 2004). For this reason, electrical conductivity sensors use AC, rather than DC (Radiometer Analytical SAS, 2004; Regtien et al., 2004).

Another way to reduce polarization is to use four electrodes. In this method, the outer two electrodes are used to send the current through the solution while the inner two electrodes are used to measure the potential difference (Radiometer Analytical SAS, 2004; Regtien et al., 2004). A representative circuit for this system is shown in Figure 2.3.4. No polarization occurs at the inner electrodes because only a very small amount of current is needed to measure the voltage (Radiometer Analytical SAS, 2004). Since the only polarization occurs at the outer two electrodes, the resistance measured by the inner electrodes corresponds to the resistance of the

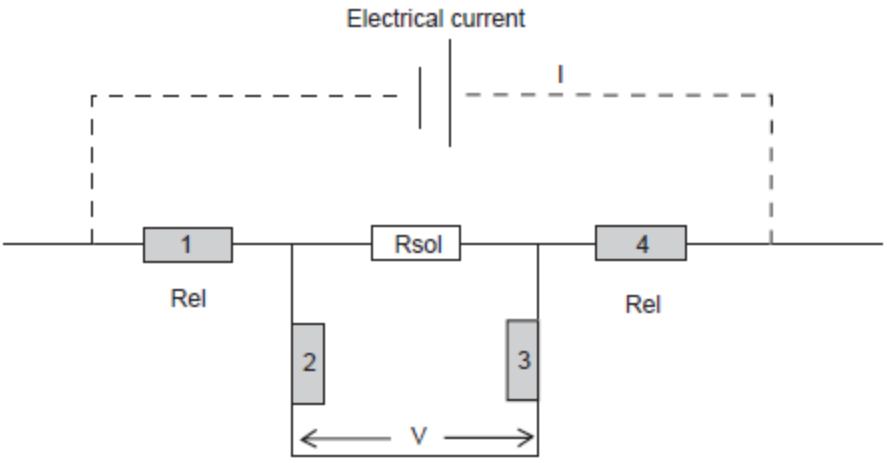


Figure 2.3.4: Circuit representing the components of a four-electrode electrical conductivity sensor (Radiometer Analytical SAS, 2004).

solution only (Radiometer Analytical SAS, 2004; Regtien et al., 2004). This measured voltage can then be directly related to the electrical conductivity of the solution. There are some disadvantages to this method, including increased complexity of design and implementation and an undesirable signal-to-noise ratio caused by high impedance of the inner electrodes (Regtien et al., 2004).

AtlasScientific offers a conductivity probe that uses two, graphite plate electrodes with AC and a cell constant of 1.0. It has a range of 5 to 200,000 µS/cm and a response time of 90 percent of the total response in one second. It can be fully submerged, withstand a maximum pressure of 1,380 kPa and a depth of 60 m, and works over a temperature range of 1 to 110 ºC. Due to the stability of the graphite electrodes, the conductivity probe only requires recalibration about once every 10 years, which is also the expected lifetime of the probe, and does not require maintenance. However, should material build up on the probe, soft coatings can be cleaned using a brush and hard coatings can be removed with a chemical cleaner (AtlasScientific, 2017d).

The conductivity probe can be combined with a conductivity EZO^{TM} circuit provided by AtlasScientific to interface with a microcontroller using either UART serial or I^2C

communication. When used together, measurements can be taken once every second with an accuracy of +/- 2 % and resolution that scales with the conductivity. The resolution is designed to provide four significant digits for conductivities greater than or equal to 10 µS/cm and three significant digits for conductivities less than 10 µS/cm, with any non-significant digits set to 0 (AtlasScientific, 2017c). The resolution for different conductivity ranges is shown in Table 2.3.2. Measurements can be output in terms of conductivity, total dissolved solids, salinity, or specific gravity. In addition, the AtlasScientific conductivity EZO™ circuit can apply and store one or two-point calibration as well as temperature compensation. When using the AtlasScientific conductivity EZO™ circuit, the effects of electrical noise must be considered. The conductivity EZO™ circuit acts as both a source of electrical noise and is also sensitive to it. The effect of electrical noise caused by the conductivity EZO™ circuit on other sensors must be considered, and if electrical noise is expected to be caused by other sources, the conductivity EZO™ circuit may require voltage isolation (AtlasScientific, 2017c).

Table 2.3.2: Resolution of AtlasScientific conductivity EZO™ circuit for different conductivity values. Values were reported as µS but are assumed to be µS/cm (AtlasScientific, 2017c).

Conductivity Range (µS/cm)	Resolution (µS/cm)
0.07 – 99.99	0.01
100.1 – 999.9	0.1
1,000 – 9,999	1.0
10,000 – 99,990	10
100,000 – 999,900	100

2.3.4 Dissolved Oxygen

Sensors that are used to measure dissolved oxygen can be divided into two main categories: optical and electrochemical. Optical dissolved oxygen sensors can further be divided into sensors that measure the lifetime of luminescence and those that measure the intensity of

luminescence of a chemical dye. The presence of dissolved oxygen decreases the intensity and lifetime of luminescence of chemical dyes, a process known as quenching (YSI, 2009). The steady-state luminescence of a chemical dye containing dissolved oxygen can be described by the Stern-Volmer equation (Lakowicz, 1992; YSI, 2009).

$$\frac{I_0}{I} = \frac{\tau_0}{\tau} = 1 + k_q \tau_0 * O_2 \qquad (2.3.14)$$

Where I_o is the intensity of luminescence when the dye contains no dissolved oxygen, I is the intensity with dissolved oxygen, τ_0 is the lifetime of luminescence without dissolved oxygen, τ is the lifetime of luminescence with dissolved oxygen, k_q is the quenching rate coefficient, and O_2 is the concentration of dissolved oxygen. This equation is applicable when the luminescence of the dye is homogeneous, a dynamic mechanism is the only source of quenching, and τ_0 is a monoexponential decay time (Lakowicz, 1992).

Both intensity and lifetime-based dissolved oxygen sensors use the basic principles of the Stern-Volmer equation to measure the concentration of dissolved oxygen, as depicted in Figure 2.3.5 (YSI, 2009). First, the sensor emits a blue light that causes the chemical dye to luminesce and emit red light. As the dye luminesces, dissolved oxygen diffuses through a membrane, or some other barrier, into the chemical dye and influences the intensity and lifetime of its luminescence according to the Stern-Volmer equation. Then, the sensor uses a photodiode to

Figure 2.3.5: The components of an optical dissolved oxygen sensor (YSI, 2009).

measure either the intensity or lifetime of the luminescence. Next, the sensor also emits a red light which is reflected by the dye. The photodiode is also used to measure the intensity or lifetime of this red light, which is used as a reference. The sensor then uses these two measurements to determine the concentration of dissolved oxygen (YSI, 2009).

Electrochemical dissolved oxygen sensors rely on the principles of amperometry, a technique in which an electrical current is measured and used to determine concentration (Regtien et al., 2004). They consist of an anode and cathode submerged in an electrolyte solution that is separated from the sample by a permeable membrane that allows dissolved oxygen to pass through. Dissolved oxygen diffuses through the membrane, is reduced at the cathode according to the following equation, and generates a current (YSI, 2009). This process is illustrated in Figure 2.3.6.

$$O_2 + 4e^- + 4H^+ \rightarrow 2H_2O \qquad (2.3.15)$$

The current that is produced by the reduction reaction can be related to the concentration of oxygen through Fick's first law of diffusion (Regtien et al., 2004).

$$i = nFAD\frac{\partial C}{\partial x} \qquad (2.3.16)$$

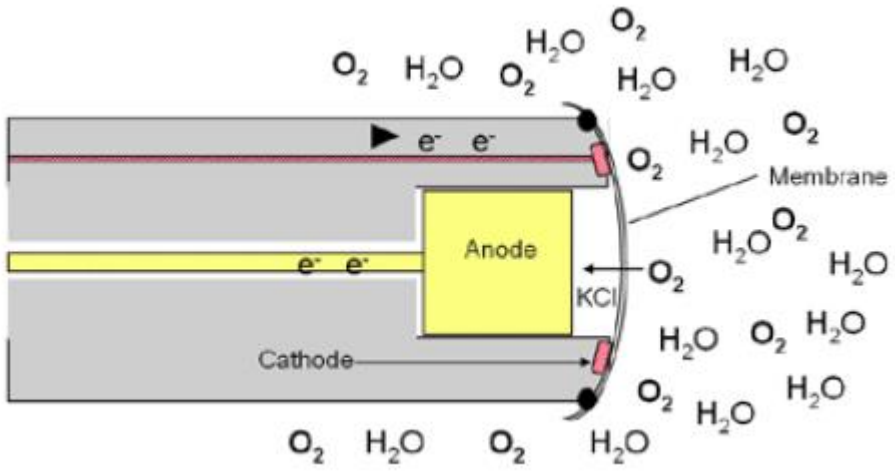


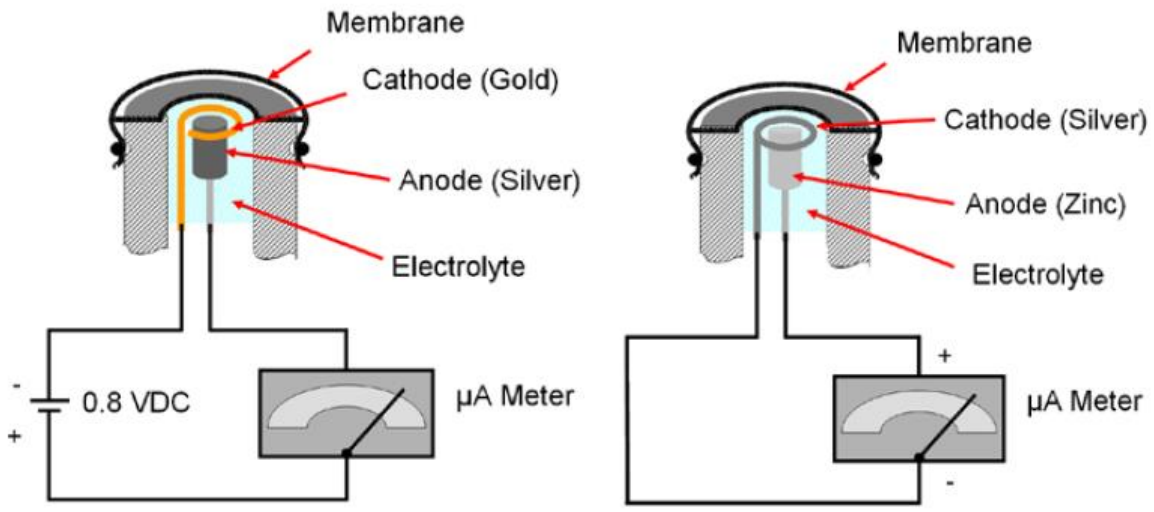
Figure 2.3.6: diagram of an electrochemical dissolved oxygen sensor (YSI, 2009).

Where i is the electric current generated by the reduction reaction, n is the number of electrons in the reduction equation ($n = 4$ for the reaction above), F is Faraday's constant, A is the electrode area, D is the diffusion coefficient, and $\partial C/\partial x$ is the concentration gradient. This equation can be simplified if a few assumptions are made. First is the assumption that the concentration gradient, $\partial C/\partial x$, is linear. Next is the assumption that the diffusion layer, δ_0, has a constant thickness. Last is the assumption that the oxygen is completely consumed at the surface of the electrode, so the concentration is equal to zero. With these three assumptions, the previous equation simplifies as follows (Regtien et al., 2004).

$$i = \frac{nFAD}{\delta_0} C \tag{2.3.17}$$

This equation shows that when the previous assumptions are met, the current generated by the reduction of oxygen will be directly proportional to the concentration. The value of the constant, $nFAD/\delta_0$, can be determined through calibration so that the concentration of dissolved oxygen can be calculated from measurements of current. Electrochemical dissolved oxygen probes are designed so that the assumptions needed to use these equations can be reasonably applied and, therefore, measured current can be used to determine the concentration of dissolved oxygen (Regtien et al., 2004).

There are two different types of electrochemical dissolved oxygen sensors: polarographic and galvanic sensors. The differences between these two types can be seen in Figure 2.3.7. The main difference between polarographic and galvanic sensors is that polarographic sensors require an applied voltage in order to polarize the electrodes, so reduction of oxygen can occur, while galvanic sensors do not require an applied voltage to polarize. This is because they typically use different metals for the cathodes and anodes. The metals shown Figure 2.3.7 are common metals that are used in these types of sensors, but other metals, such as lead instead of zinc, may be used



(a) (b)

Figure 2.3.7: Simple schematic showing the basic components of electrochemical dissolved oxygen sensors: (a) polarographic sensor and (b) galvanic sensor (YSI, 2009).

instead. Both types of sensors measure the change in current caused by oxygen reduction at the cathode and use this to determine the concentration of dissolved oxygen (YSI, 2009).

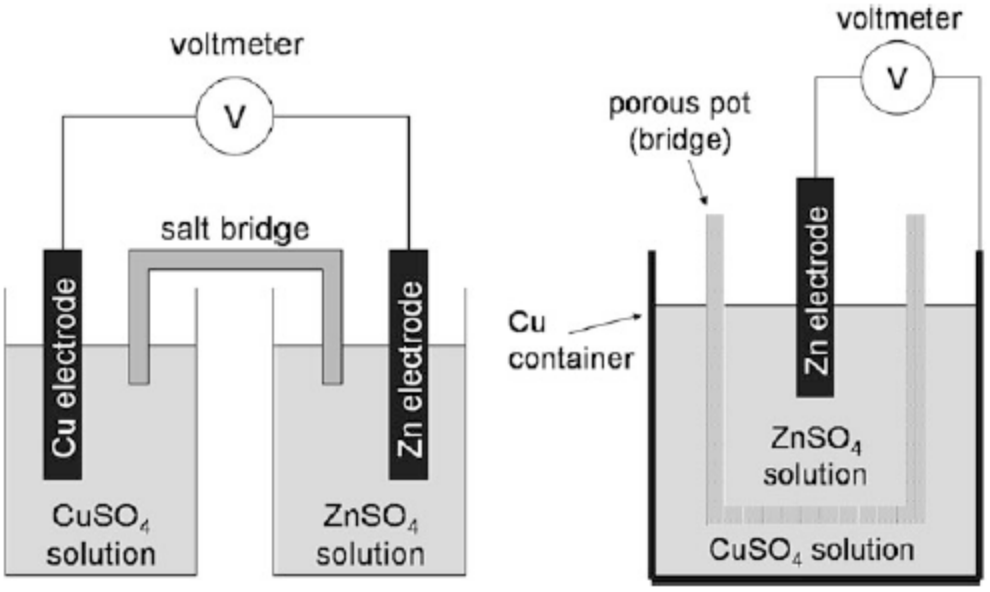
One concern with electrochemical dissolved oxygen sensors is that they consume oxygen as it is reduced. This means that, in a steady-state environment with no flow, the sensor will consume the dissolved oxygen in the sample and result in lower measured concentrations than the sample originally had. This is known as flow dependence and means that a small amount of flow is required when using electrochemical dissolved oxygen sensors to get accurate readings (YSI, 2009).

The dissolved oxygen probe from AtlasScientific is a galvanic probe which uses a silver cathode, zinc anode, and a polyethylene membrane (AtlasScientific, 2017b). Like the AtlasScientific conductivity probe, it can be interfaced with an EZO^{TM} circuit available from AtlasScientific to communicate with a microcontroller using either UART serial or I^{2}C

communication (AtlasScientific, 2017a). The probe has a range of 0 to 35 mg/L with a response time of approximately 0.3 mg/L per second. It can be fully submerged, withstand a maximum pressure of 690 kPa and a depth of 60 m, and functions over a temperature range of 1 to 50 ºC. It requires recalibration about once every year and replacement of the electrolyte solution and Teflon or HDPE membrane about every two years (AtlasScientific, 2017b). When combined with AtlasScientific's dissolved oxygen EZO™ circuit, measurements can be taken every second with an accuracy of +/- 5 mg/L. Measurements can be output in terms of both mg/L or percent saturation. In addition, the dissolved oxygen EZO™ circuit can be used to apply and store one or two-point calibration data and apply temperature, salinity, and pressure compensation, if needed. One disadvantage to using the dissolved oxygen EZO™ circuit is that, like the electrical conductivity EZO™ circuit, it is very sensitive to electrical noise and may require a voltage isolator (AtlasScientific, 2017a).

2.3.5 Ion Selective Electrodes: pH, Nitrate, and Chloride

Ion-Selective Electrodes are used to measure the concentration of a specific ion (Freiser, 1978; Cammann, 1979; Morf, 1981; Regtien et al., 2004; Ripka and Tipek, 2007; Yoon, 2016). They function based on the principles of an electrochemical cell, which consists of two electrodes submerged in separate electrolyte solutions connected through a salt bridge, semi-permeable membrane, or some other material which allows current to flow between the two electrodes (Freiser, 1978; Cammann, 1979; Regtien et al., 2004; Yoon, 2016). Two examples of electrochemical cells are shown in Figure 2.3.8. The combination of each electrode and electrolyte solution on either side of the bridge is known as a half-cell (Yoon, 2016). In each of the half-cells, a potential difference develops between the electrodes and the electrolyte solutions (Cammann, 1979; Morf, 1981). When these potential differences have different values, the



(a) (b)

Figure 2.3.8: Examples of electrochemical cells: (a) Galvanic cell – connected through a salt bridge and (b) Daniell cell – connected through a porous pot (Yoon, 2016).

potential difference between the two half-cells creates a current with a voltage that can be measured.

The potential difference that develops in a half-cell is dependent on different quantities depending on the type of electrode (Cammann, 1979). For example, for an electrode of the first kind, the potential difference is dependent on the activity of the metal ion the electrode is made of. An electrode of the second kind is coated in a thin layer of one of its slightly soluble anion salts and the potential difference is dependent on the activity of that anion salt in the electrolyte solution (Cammann, 1979; Ripka and Tipek, 2007). For ion-selective electrodes, a membrane which allows only a specific type of ion to pass through is placed between the electrode and the electrolyte. In this case, the potential difference develops across that membrane and is dependent on the activity of the particular ion allowed to cross the membrane (Freiser, 1978; Cammann,

1979; Regtien et al., 2004; Ripka and Tipek, 2007). For this reason, ion-selective electrodes can be used to determine the activity of a specific ion.

In order to measure the potential difference across the membrane, ion-selective electrodes require two reference electrodes, one internal and one external, on either side of the membrane (Morf, 1981; Ripka and Tipek, 2007; Zhang et al., 2008). Along with the internal reference electrode, an electrolyte is necessary in order to allow current to pass from the electrode, through the electrolyte to the membrane, and into the sample solution. The external reference electrode acts as a second half-cell, which completes the electrochemical cell and allows current to flow between the two electrodes and voltage to be measured (Ripka and Tipek, 2007). A complete ion-selective electrode is depicted in Figure 2.3.9.

Potential differences develop at each of the interfaces (between electrodes and electrolytes, between electrolyte and membrane, and between membrane and sample), however,

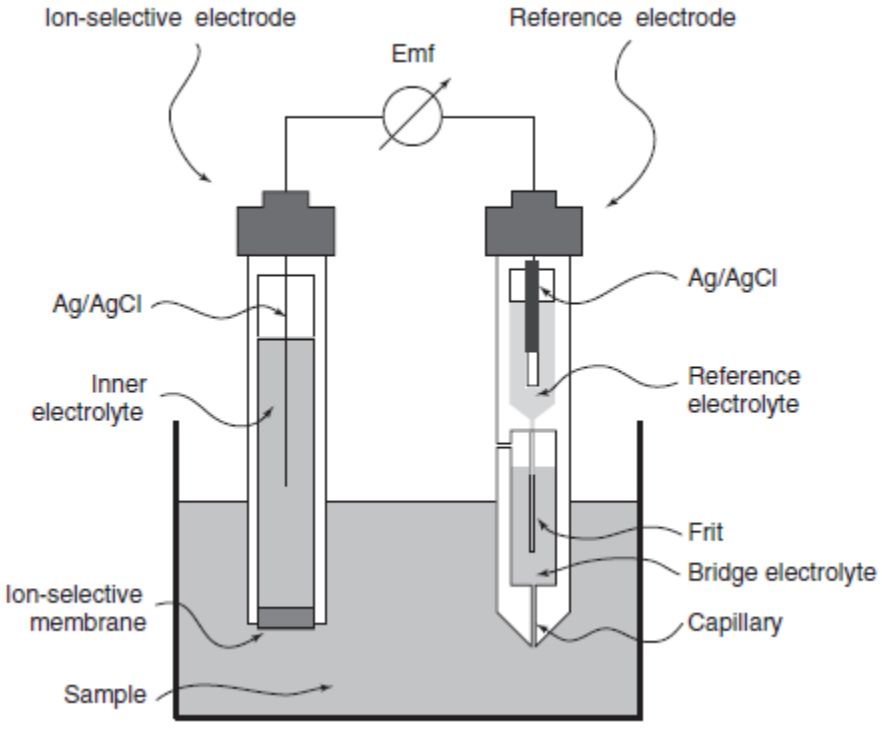


Figure 2.3.9: Ion-selective electrode (Zhang et al., 2008).

each of the potential differences can be considered constant except the potential difference that develops between the membrane and the sample (Morf, 1981; Regtien et al., 2004; Zhang et al., 2008). Therefore, through calibration, the effects of these other interfaces can be removed and the potential difference across the membrane-sample interface can be determined. This potential difference is related to the concentration of the particular ion able to travel through the membrane (which can be considered approximately equal to its activity) through the Nernst equation (Cammann, 1979; Morf, 1981; Regtien et al., 2004; Zhang et al., 2008; Yoon, 2016).

$$E = E° + \frac{RT}{nF}\ln[M^+] \qquad (2.3.18)$$

Where E is the electrical potential, $E°$ is the standard electrode potential (also called the reference or ground potential), $R = 8.3145\frac{J}{K*mol}$ is the universal gas constant, T is temperature in Kelvin, n is the number of electrons, $F = 96{,}487\frac{C}{mol}$ is Faraday's constant, and $[M^+]$ is the ionic concentration. Once the values of $E°$ and $\frac{RT}{nF}$ are determined through calibration, this equation can be used to calculate the concentration of the specific ion from the measured voltage.

Ion-selective electrodes can be used to measure several different ions. Different types of membranes are used in order to restrict the flow to different ions. There are three main types of membranes used by ion-selective electrodes, the first of which is a glass membrane (Morf, 1981; Yoon, 2016). This glass membrane is constructed as a very thin layer of glass with a specific composition designed to allow one specific type of ion to pass through (Yoon, 2016). The most common is the pH electrode, which measures the concentration of H^+ ions (Morf, 1981; Yoon, 2016). An example of this type of sensor is the pH probe from AtlasScientific (AtlasScientific, 2017f). Some ion-selective electrodes use liquid membranes instead. These are constructed as a plastic membrane, often PVC, which has absorbed a liquid ion-exchange material that restricts

the flow to one type of ion (Morf, 1981; Yoon, 2016). Liquid membranes can be used for ion-selective electrodes intended to measure the concentrations of both nitrate and chloride (Yoon, 2016). Vernier's nitrate ion-selective electrode is an example that uses a liquid, PVC membrane (Vernier, 2014). The final type of membrane is a solid-state membrane. These membranes are cut from a crystalline material that only allows a particular ion to pass through (Morf, 1981; Yoon, 2016). Solid-state membranes can be used to measure the concentration of chloride ions (Yoon, 2016).

In Figure 2.3.9, the ion-selective electrode is depicted with the two reference electrodes housed in separate bodies. However, in many modern sensors, both electrodes can be stored in the same body in what is called a combination electrode (Rundle, 2000). This does not change how the ion-selective electrode functions, but simply arranges it in a way that is more convenient for taking measurements. The pH probe from AtlasScientific as well as both the nitrate and chloride ion-selective electrodes from Vernier are combination electrodes.

The AtlasScientific pH probe uses a combination of silver wire and silver chloride as the sensing electrode, a glass membrane, and a KCl reference solution. It relates pH to measured voltage using the following equation, which is altered from the Nernst equation by relating the activity to pH (AtlasScientific, 2017f).

$$E = E^0 - \frac{2.303RT}{F} pH \qquad (2.3.19)$$

It is capable of measuring pHs ranging from 0 to 14 and has a response time of 95 percent of the total response in one second. Like the AtlasScientific dissolved oxygen probe, it can be fully submerged and is capable of withstanding a maximum pressure of 690 kPa and a depth of 60 m. The pH probe functions over a temperature range of 1 to 99 ºC. When used to measure weak acids or bases, the probe requires recalibration once a year for the first two years and

approximately every six months thereafter. When used to measure strong acids or bases, the probe should be recalibrated monthly or after every use. The probe is expected to last over 2.5 years and does not require much maintenance, however, it should be stored in a storage solution with a pH of about 3.8 and may require reconditioning in time. In addition, any coatings that may build up on the membrane should be cleaned by stirring and rinsing, with a chemical cleaning solution if necessary (AtlasScientific, 2017f).

Like the other probes offered by AtlasScientific, the pH probe can be combined with AtlasScientific's pH EZO™ circuit to use with a microcontroller with either UART serial or I^2C communication. When combined in this way, measurements can be taken every second with an accuracy of +/- 0.002. It can be used to apply and store one-, two-, or three-point calibration as well as apply temperature compensation. Like the other AtlasScientific EZO™ circuits, the pH EZO™ circuit is very susceptible to electrical noise and may require voltage isolation (AtlasScientific, 2017e).

The Vernier nitrate ion-selective electrode uses a PVC porous disk as a membrane and an Ag/AgCl electrode as the internal reference electrode. The PVC membrane is impermeable to water, but permeable to an ion exchanger that allows nitrate ions to pass through. It is capable of measuring concentrations between 1 and 10,000 mg/L with a precision of +/- 10 percent of the full scale. It functions over a temperature range of 0 to 40 °C and a pH range of 2 to 11. If any of the following ions are also present in the solution, they may interfere with the probe and cause inaccurate readings: ClO_4^-, I^-, ClO_3^-, CN^-, and BF_4^-. The PVC membrane should last at least one year before requiring replacement (Vernier, 2014).

The Vernier chloride ion-selective electrode uses a solid-state membrane or solid polymer porous disk as a membrane with an Ag/AgCl internal reference electrode. It can

measure concentrations ranging from 1 to 35,000 mg/L with a precision of +/- 10 percent of the full scale. It works within a temperature range of 0 to 80 °C and a pH range of 2 to 12. Like the nitrate ion-selective electrode, the presence of certain ions may interfere and cause inaccurate readings. These include CN^-, Br^-, I^-, OH^-, S^{-2}, and NH_3^-. Unlike the nitrate ion-selective electrode, however, the membrane is expected to last as long as the probe itself, though it may require polishing to remove the outer layer of the membrane as it becomes inactive with use (Vernier, 2017).

Both of Vernier's nitrate and chloride ion-selective electrodes are used in the same way. They output a voltage that can be related to concentration through the Nernst equation with the slope and intercept of the line relating the voltage to the natural log of concentration determined through a two-point calibration. When calibrating, the calibration standards should be chosen carefully so that all measured concentrations fall within the range of the two standards to get accurate readings. Before every use, the calibration of the probes should be checked, and recalibration may be required. Before use, the probes must be soaked in the high standard used for calibration for at least 30 minutes, but no longer than 24 hours. While in use, the probes should be submerged in at least 2.8 cm of water, not rest on the bottom of the container, have no bubbles trapped beneath them, and their white reference contacts should be completely submerged. In addition, the probes may cause or experience interference when used with other sensors, which must be taken into consideration. After use, the probes should be rinsed with distilled water and patted dry with paper towel. To maximize their lifetime, the probes should never be immersed in solution for longer than 24 hours and should be stored in a storage bottle with a sponge soaked with distilled water to keep the membranes and reference contacts moist.

When properly cared for, the probes are expected to last at least 5 years (Vernier, 2014; Vernier, 2017).

2.3.6 Turbidity

Turbidity is cloudiness caused by suspended sediments or other particles and is an indication of the quality of the water. Suspended particles can include clay, silt, sand, minerals, organic matter, plankton, and other microscopic organisms. When light travels through water with suspended particles, it is scattered and absorbed, and its intensity is reduced (Down and Lehr, 2005; Geddes et al., 2014). Turbidity sensors take advantage of this to determine the turbidity of a solution by measuring the intensity of light after it has passed through a portion of solution. The more suspended particles there are, the more the light will be scattered and absorbed; therefore, lower intensity indicates higher turbidity (Geddes et al., 2014).

There are two main approaches to measuring turbidity: taking and measuring the turbidity of samples and using a submersible turbidity sensor to measure the turbidity in place. The biggest disadvantage to sample-based turbidity sensors is that they cannot provide a continuous record of turbidity as submersible sensors can. In addition, once a sample is collected its turbidity will begin to change due to settling, dissolution, precipitation, flocculation, and biological production and consumption of the suspended particles. This means that samples must be either analyzed as quickly as possible or stored in specific conditions to minimize the effects of these processes. Therefore, for many applications, submersible sensors may be preferred. In both cases, turbidity is very hard to standardize, so using different turbidity sensors often results in different measured turbidities even when the same sample is used. The best approach is to choose one turbidity sensor and use that sensor for all measurements. Often, the same techniques

to measure turbidity can be used in both sample-based and submersible turbidity sensors (Down and Lehr, 2005).

One of the earliest techniques developed to measure turbidity is the EPA 180.1, or nephelometric method. This technique is the origin of the units used to describe turbidity, nephelometric turbidity units, or NTU. Sensors that use this method use a tungsten-filament lamp (TFL) as a light source and a photodiode to measure the intensity of light that has been scattered 90 degrees by the sample, as depicted in Figure 2.3.10 (Down and Lehr, 2005). While sensors that use this method are often durable and relatively cheap, they also require a lot of power and cannot be switched to reject ambient light. Therefore, this technique is not often used in submersible sensors. In addition, sensors that use this method cannot be used to measure samples with high turbidity (greater than 40 NTU) without using dilution (Down and Lehr, 2005).

The ISO 7027 method is similar to the EPA 180.1 method in that it uses a light source and detector oriented 90 degrees from one another as shown in Figure 2.3.11 (Down and Lehr, 2005). However, the ISO 7027 method uses an infrared emitting diode (IRED) or IR laser diode (LD) as a light source (rather than a TFL) which can be switched to reject ambient light. While

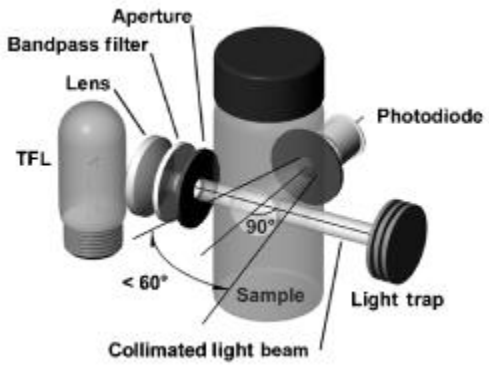


Figure 2.3.10: Schematic showing the EPA 180.1 method to measure turbidity (Down and Lehr, 2005).

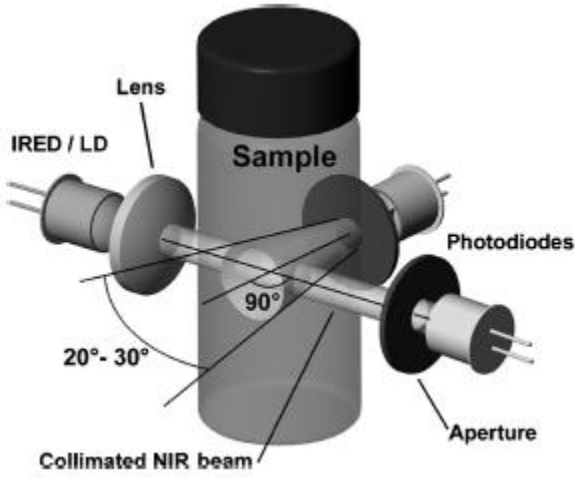


Figure 2.3.11: Schematic showing the ISO 7027 method to measure turbidity (Down and Lehr, 2005).

this method can be used in submersible sensors, like the EPA 180.1 method, it cannot be used to measure turbidities greater than 40 NTU without dilution (Down and Lehr, 2005).

Another method is the GLI-2 method, which uses two pairs of IREDs and photodiodes oriented 90 degrees from one another, as shown in Figure 2.3.12 (Down and Lehr, 2005). To determine the turbidity of the sample, two measurements are taken. First, one of the IREDs is turned on and both photodiodes measure the intensity of light, one directly across from the emitter and one oriented 90 degrees from it, to determine the ratio of transmitted to scattered light intensity. Then, this process is repeated using the second IRED as the source of light. A microcontroller then converts these ratios to a measured turbidity. This method greatly reduces errors relating to fluctuating IRED power, photodiode sensitivity, and uniform window fouling and can be used in submersible sensors (Down and Lehr, 2005).

The last technique used to measure turbidity is used by backscatter sensors. These sensors use an IRED as a light source which is surrounded by four photodiodes that detect light scattered more than 90 degrees (Down and Lehr, 2005). Figure 2.3.13 depicts this type of sensor.

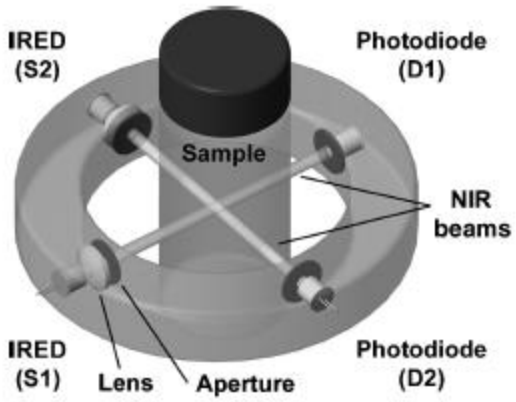


Figure 2.3.12: Schematic showing the GLI-2 method to measure turbidity (Down and Lehr, 2005).

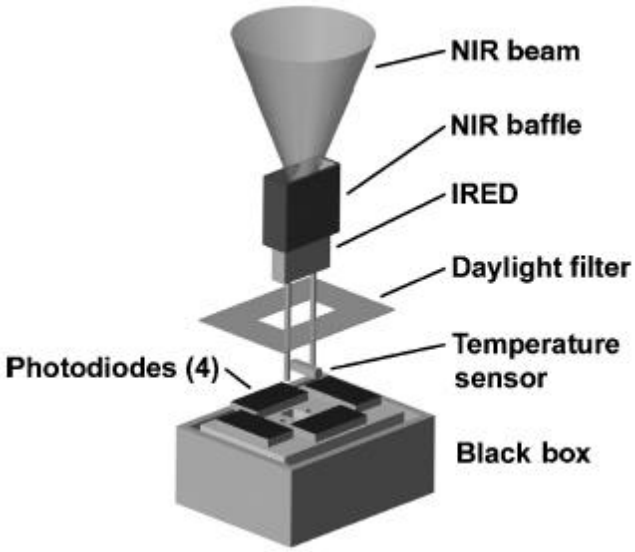


Figure 2.3.13: Schematic of backscatter sensors used to measure turbidity (Down and Lehr, 2005).

Backscatter sensors are very good for continuous measurements, require low power, and have a linear response up to 4000 NTU, however, they are not good at measuring low turbidities (less than 5 NTU) and are susceptible to both debris and color of suspended particles (Down and Lehr, 2005).

The gravity turbidity sensor offered by DFRobot can be used to measure turbidity and comes with an adaptor for use with a microcontroller. It requires a 5 V operating voltage, uses a maximum current of 40 mA, and has a less than 500 ms response time. It provides an output voltage between 0 and 4.5 V which can then be related to turbidity through the following plot in Figure 2.3.14. It functions over a temperature range of 5 to 90 °C and has a minimum insulation resistance of 100 MΩ (DFRobot, 2019).

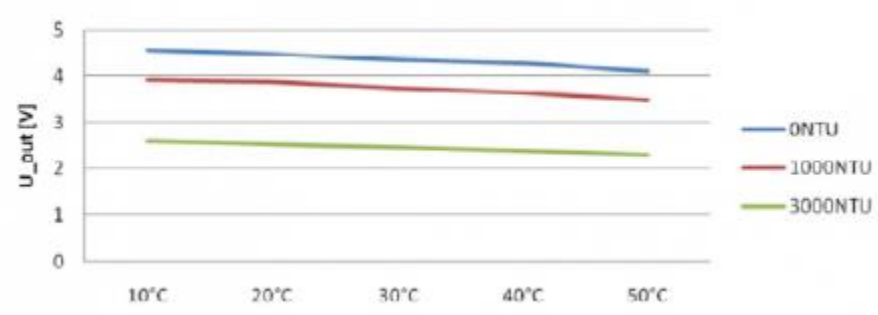


Figure 2.3.14: Relationship between turbidity and voltage output by the gravity turbidity sensor for various temperatures (DFRobot, 2019).

2.3.7 Accelerometer

Accelerometers are devices that indirectly measure the force exerted by a mass in order to determine the acceleration acting on the mass. The most common types of accelerometers include piezoelectric, piezoresistive, and capacitive accelerometers. To determine the force acting on a mass, piezoelectric and piezoresistive accelerometers use the strain or deformation of an elastic element caused by the mass, while capacitive accelerometers use the displacement. Force can then be related to the acceleration through the following equation (Ripka and Tipek, 2007).

$$F = -ma \tag{2.3.20}$$

Where F is the force exerted by the mass on the piezoelectric element, m is the mass, and a is the acceleration of the mass. Figure 2.3.15 shows simplified schematics of these types of accelerometers.

Piezoelectric accelerometers take advantage of the properties of piezoelectric materials, such as quartz or polycrystalline ceramics, which create an electrical signal that is proportional to the stress felt by the material (Wilson, 2005; Ripka and Tipek, 2007). In these devices, a mass is attached to the piezoelectric element so that when it experiences an acceleration, the mass applies stress to the piezoelectric element. The piezoelectric element then outputs an electric signal due to this stress (Wilson, 2005; Ripka and Tipek, 2007; Mori, 2017). Stress can be related to acceleration through the following equation.

$$\sigma = \frac{F}{A} = \frac{-ma}{A} \tag{2.3.21}$$

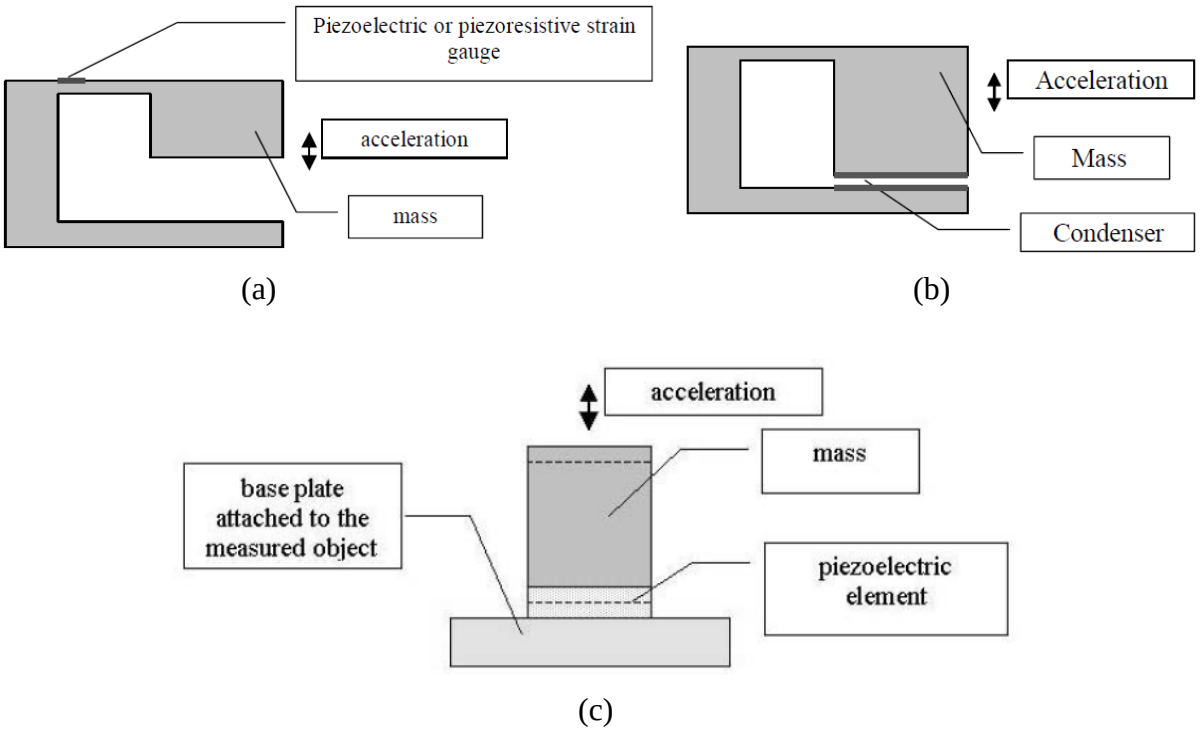


Figure 2.3.15: Simple schematics of types of accelerometers: (a) Piezoresistive Accelerometer, (b) Capacitive Accelerometer, and (c) Piezoelectric Accelerometer (Ripka and Tipek, 2007).

Where σ is the stress acting on the piezoelectric element and A is the area over which the force is acting. This means that, since the electric signal is proportional to the stress, it is also proportional to the acceleration and therefore can be used to determine the acceleration acting on the mass. The constant relating the measured electric signal to acceleration can be determined through calibration.

Piezoresistive accelerometers use a similar technique, however, they rely on strain gauges, which are made of piezoresistive rather than piezoelectric materials. Like piezoelectric materials, piezoresistive materials react when they experience stress, however, they react by changing resistance rather than generating an electric signal (Zhang and Wei, 2017). In piezoresistive accelerometers, the mass is attached to a small beam with strain gauges attached (Ripka and Tipek, 2007). When the mass experiences an acceleration, it causes the beam, along with the attached strain gauges, to deform and this deformation changes the resistance of the strain gauges (Wilson, 2005; Ripka and Tipek, 2007). The strain gauges are designed and arranged in such a way that the measured output, when altered by the resistances of the strain gauges, is proportional to the acceleration (Ripka and Tipek, 2007; Zhang and Wei, 2017).

Capacitive accelerometers use electrodes or capacitive plates to determine the acceleration. Two electrodes or capacitive plates are placed so that one is attached to the mass while the other is fixed in place. When the mass experiences an acceleration, the electrodes move relative to each other and as the distance between them changes, so does their capacitance (Ripka and Tipek, 2007). For example, the capacitance of two parallel plates can be calculated with the following equation (George et al., 2017).

$$C = \frac{\varepsilon_0 \varepsilon_r A}{d} \tag{2.3.22}$$

Where C is the capacitance, ε_0 is the dielectric permittivity of free space, ε_r is the relative dielectric permittivity, A is the cross-sectional area of the two plates, and d is the distance between the plates. Similar equations exist for different geometries and shapes of electrodes, however, since the geometry of the electrodes is known and constant, the correct equation can be determined to relate capacitance to distance (Ripka and Tipek, 2007). Then acceleration can be determined by measuring how the capacitance, and therefore distance between the electrodes, changes with time. The variation in distance between the electrodes is equivalent to the displacement of the mass and the acceleration of the mass is simply the second derivative of the displacement. In this way, the acceleration can be determined from the measured capacitance.

One of the benefits of piezoresistive and capacitive accelerometers is that they have the potential to be made into Micro-Electric Mechanical Systems, or MEMS (Wilson, 2005; Ripka and Tipek, 2007; Zhang and Wei, 2017). The basic principles used by MEMS accelerometers are the same as in conventional accelerometers, but they are built on a much smaller scale. The advantages to using MEMS accelerometers include low cost, very small size, and low power consumption. An example of capacitive MEMS accelerometers is the ADXL series by Analog Devices (Zhang and Wei, 2017).

The ADXL337 accelerometer from Analog Devices is a 3-axis, analog, capacitive MEMS accelerometer. It can convert accelerations within a range of +/- 3 g to an electrical signal with a sensitivity of about 300 mV/g and a nonlinearity of +/- 0.3 percent. It has bandwidths of 1600 Hz for the x and y axes and 550 Hz for the z axis. The noise density for the x and y axes is 175 μg/√Hz and 300 μg/√Hz for the z axis. It can withstand a shock of 10,000 g and functions within a temperature range of -55 to 125 °C (Analog Devices, 2010). The relationship between the output electrical signal and acceleration can be determined by

calibrating to find the gain and offset of each of the three axes, as shown in the equations below (SparkFun Electronics, 2019a).

$$O = 0.5\left(A_{+1g} + A_{-1g}\right) \tag{2.3.23}$$

$$G = 0.5\left(\frac{A_{+1g} - A_{-1g}}{1g}\right) \tag{2.3.24}$$

Where O is the offset, G is the gain, and A_{+1g} and A_{-1g} are the measured values when the axis is oriented in the vertical direction so that gravity is acting in the positive and negative directions. Once these values are determined, the acceleration can then be determined from the measured electrical signals by subtracting the offset and dividing by the gain (SparkFun Electronics, 2019a).

2.3.8 Electrical Resistivity Methods

Like electromagnetics, electrical resistivity methods are used to image the electrical resistivity distribution of the subsurface. However, while electromagnetic methods are used to determine the electrical conductivity of the subsurface, electrical resistivity methods are used to determine the electrical resistivity (Telford et al., 1990; Fitterman and Labson, 2005; Zonge et al., 2005). While these two methods are used to determine different material parameters, they really provide the same information because conductivity and resistivity are the inverse of one another.

$$\sigma = \frac{1}{\rho} \tag{2.3.25}$$

Where σ is electrical conductivity and ρ is electrical resistivity. The main difference between these two methods is in the way they obtain information. Unlike electromagnetics, which use coils to induce current in the subsurface, electrical resistivity methods rely on sending current through direct contact into the ground (Telford et al., 1990; Zonge et al., 2005).

Taking a measurement using electrical resistivity methods requires the use of four electrodes: two current electrodes and two potential electrodes. The current electrodes are used to send a current through the subsurface (Telford et al., 1990; Zonge et al., 2005). This current flows through the subsurface with an uneven distribution. The distribution of current is dependent on both the material composition of the subsurface and the spacing of the two electrodes. The current tends to concentrate in areas of low resistivity and at shallow depths, but the larger the spacing between the electrodes, the deeper the current penetrates into the subsurface. When the resistivity of the subsurface is completely uniform, the distribution of current is controlled solely by the separation between the electrodes. Figure 2.3.16 shows how the distribution of current follows these trends, with the tendency for current to concentrate at shallower depths and near materials with lower resistivity (Telford et al., 1990).

As the current flows through the subsurface, the electrical resistivity of the materials in the subsurface creates a resistance against this flow. Resistance and electrical resistivity can be related through the following equation for solids with uniform cross-sections (Telford et al., 1990).

$$R = \rho \frac{L}{A} \tag{2.3.26}$$

Where L is the length and A is the cross-sectional area of the material through which current flows. The corresponding potential differences vary spatially depending on the distribution of the current flow (for example, equipotential lines of the potential values in a homogeneous medium can be seen in Figure 2.3.16a) (Telford et al., 1990). The two potential electrodes can be used to measure the potential difference between two points and, if the current injected through the current electrodes is also recorded, the apparent resistivity of the subsurface can be determined from the following equation (Telford et al., 1990; Zonge et al., 2005).

$$\rho_a = \left(\frac{2\pi}{\frac{1}{r_1} - \frac{1}{r_2} - \frac{1}{r_3} + \frac{1}{r_4}} \right) \frac{\Delta V}{I} \tag{2.3.27}$$

Where ρ_a is the apparent resistivity, ΔV is the measured potential difference, I is the current, and the r values describe the geometric spacing of the electrodes, as shown in Figure 2.3.17. This type of measurement results in an apparent resistivity because it assesses the resistivity of a volume that is assumed to be electrically homogeneous. While this method cannot directly provide the actual resistivity of each point in the subsurface, useful information may still be obtained from how the apparent resistivity varies spatially by changing the location of and separation between electrodes (Telford et al., 1990; Zonge et al., 2005).

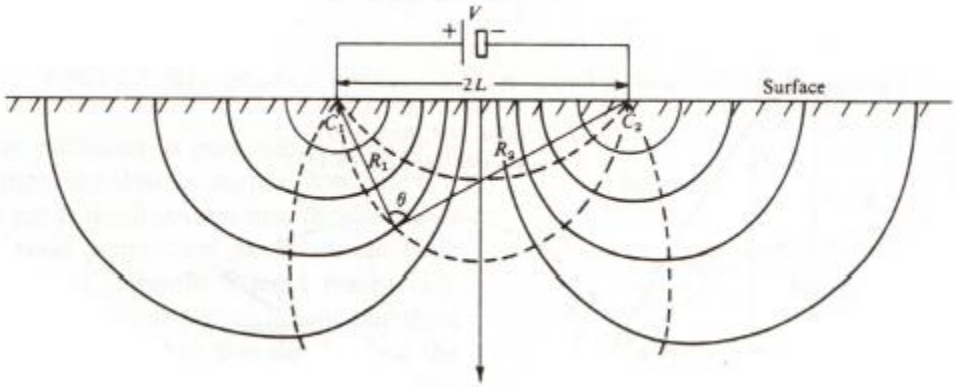
There are three main types of electrode configurations used to evaluate how the apparent resistivity varies spatially. These are the Wenner, Dipole-Dipole, and Schlumberger arrays, which are depicted in Figure 2.3.18 (Telford et al., 1990; Zonge et al., 2005). For the geometries of these arrays, the equation for apparent resistivity can be simplified using the a and n spacing described by Figure 2.3.18. These simplified forms of the equation are shown below where the equations correspond to the Wenner, Dipole-Dipole (Telford et al., 1990; Zonge et al., 2005), and Schlumberger arrays, respectively (Zonge et al., 2005).

$$\rho_a = 2\pi a \frac{\Delta V}{I} \tag{2.3.28}$$

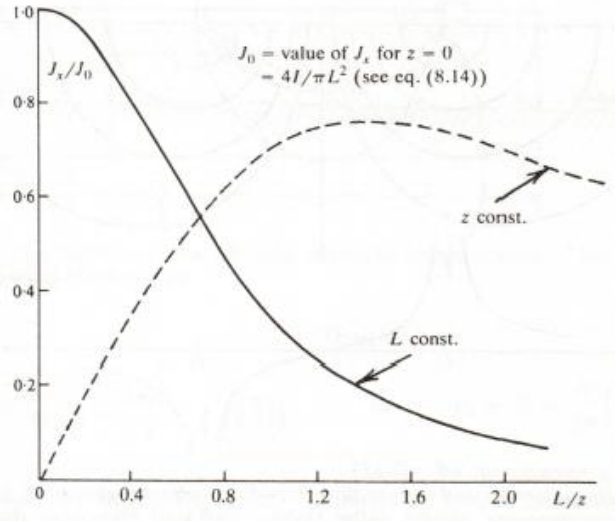
$$\rho_a = \pi a n (n+1)(n+2) \frac{\Delta V}{I} \tag{2.3.29}$$

$$\rho_a = \pi a n (n+1) \frac{\Delta V}{I} \tag{2.3.30}$$

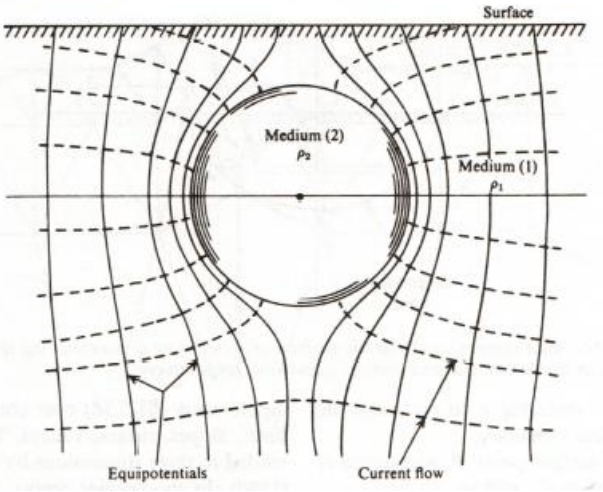
Where ρ_a is the apparent resistivity, a is the smallest electrode spacing, n is an integer that describes the spacing between current and potential electrodes for the Dipole-Dipole and Schlumberger arrays, ΔV is the potential difference, and I is the current.



(a)



(b)



(c)

Figure 2.3.16: Distribution of current flow: (a) in a homogenous medium, controlled by electrode separation (dashed lines indicate current flow lines while solid lines indicate constant potential values (equipotential lines)), (b) variation with depth (solid line shows decrease in current density with depth), and (c) influenced by material with low resistivity (Telford et al., 1990).

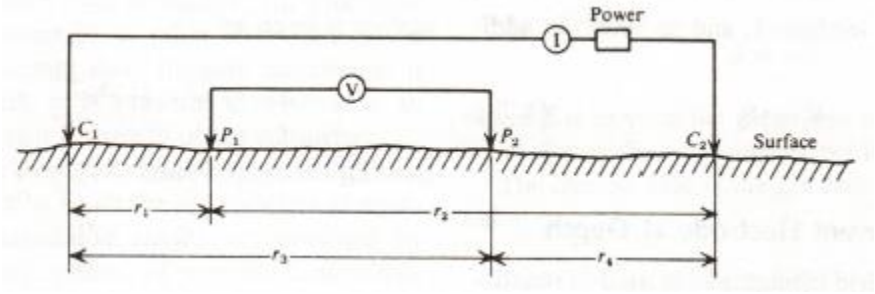
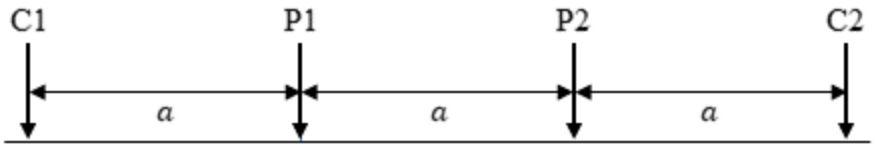
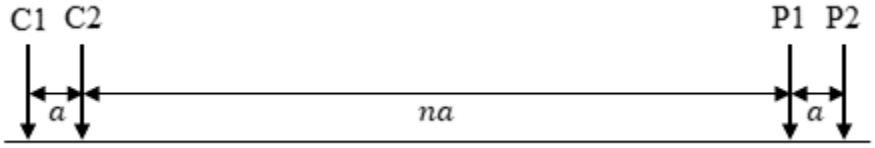


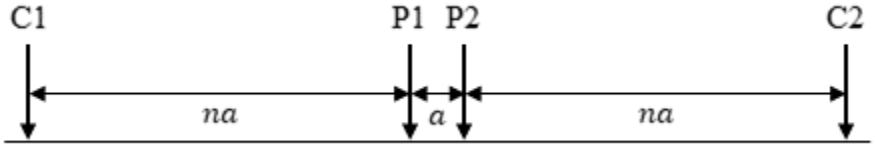
Figure 2.3.17: Geometric spacing of electrodes during a measurement of apparent resistivity (Telford et al., 1990).



(a) Wenner



(b) Dipole-Dipole



(c) Schlumberger

Figure 2.3.18: Most common electrode configurations (a) Wenner, (b) Dipole-Dipole, and (c) Schlumberger arrays.

In order to plot the data, a horizontal position and depth can be assigned to each apparent resistivity measurement to create a cross section called a pseudosection. These pseudosections are not real cross sections of the electrical resistivity distribution of a medium, however, since each measurement, in reality, corresponds to a volume rather than a single point and the values do not correspond to the true resistivity (Zonge et al., 2005). To plot a point in the pseudosection for any given electrode configuration, the horizontal location of the point is considered to be directly in the center of all four electrodes at a depth that is related to the spacing between the electrodes. The depths for the three arrays shown in Figure 2.3.18 are described by the following

equations with the equations corresponding to the Wenner, Dipole-Dipole, and Schlumberger arrays, respectively (Sharma, 2002).

$$d = \frac{a}{2} \tag{2.3.31}$$

$$d = \frac{a}{2}(1+n) \tag{2.3.32}$$

$$d = \frac{2na + a}{6} \tag{2.3.33}$$

Using the locations of each measurement in the pseudosection, electrode arrays can be used to measure how the apparent resistivity varies spatially by adjusting the position of the electrodes. For the Wenner array, to evaluate how the apparent resistivity varies horizontally, measurements are taken by moving all four electrodes to different positions while maintaining a constant spacing, a. For the Schlumberger array, the current electrodes are kept in the same location and the potential electrodes are moved between them. In the dipole-dipole array, the current and potential electrodes are moved, and measurements are taken at several different locations. For all three arrays, the relative electrode spacings can be increased or decreased to measure how apparent resistivity varies with depth (Telford et al., 1990; Zonge et al., 2005).

There are several different methods for interpreting electrical resistivity data. One method involves matching the data to expected trends. As each of these arrays cross different horizontal or vertical features in the subsurface (such as layering or anomalies) the measured apparent resistivity values vary in distinct ways. For example, a Schlumberger array passing over a spherical anomaly (when the current electrodes are located far from the anomaly and their position is fixed) will produce a curve like the one shown in Figure 2.3.19. Similar curves exist for many other features that may be present in the subsurface and can be used to determine what type of structures may be present. Measured apparent resistivity values can then be matched to

dimensionless master curves to determine the geometry and resistivities of the materials that make up those structures (Telford et al., 1990; Zonge et al., 2005). Another more modern method is to use computer software to perform an inversion of the data. This software attempts to create a cross section that would match the electrical resistivity data when traversed with the same electrode positions and spacings. One disadvantage to this technique is that the results are non-unique and the true cross section is impossible to determine (Zonge et al., 2005).

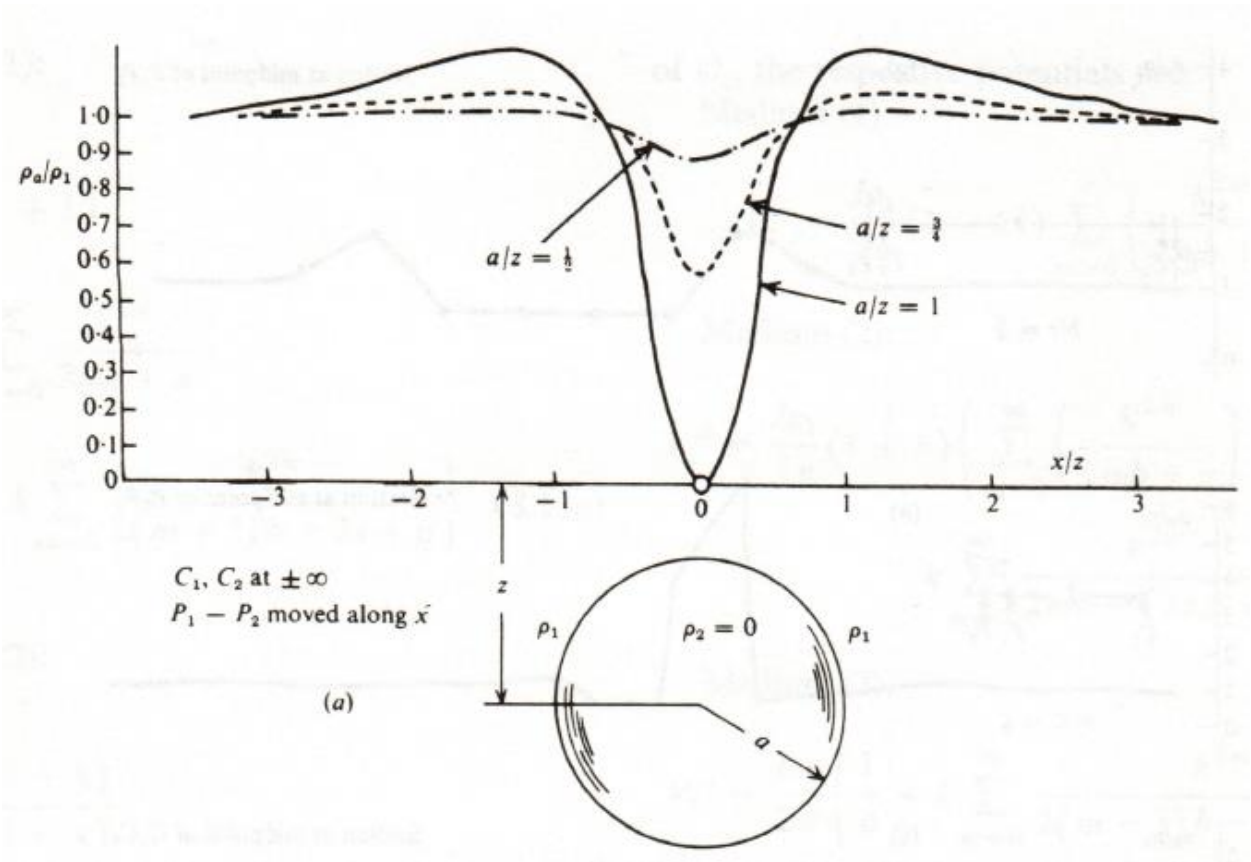


Figure 2.3.19: Typical change in apparent resistivity caused by traversing across a spherical anomaly with a Schlumberger array with constant location of current electrodes (Telford et al., 1990).

3 Digitization of an Electromagnetic Sensor

One method of evaluating the properties of the subsurface is to use electromagnetic sensors. These sensors rely on induction to produce and measure electromagnetic waves, which are influenced by the conductivity of nearby materials. Because of this, electromagnetic sensors can be used to identify changes in rock or sediment type, groundwater chemistry, the location of contaminants, and other factors that influence conductivity. One of the biggest advantages to electromagnetic sensors is that, since they rely on induction, they do not have to be in direct contact with the ground, leading to rapid and easy data collection. In this chapter, the details of the design and development of a data logging system for a Geonics Limited EM-31 electromagnetic sensor are discussed.

3.1 System Setup and Wiring

The Geonics Limited EM-31 is an electromagnetic instrument which relies on the basic electromagnetic methods described in Chapter 22.3.1 of this thesis to provide measurements of electrical conductivity of near surface soils and rocks. We have an older EM-31 model that uses an analog dial gauge to show electrical conductivity readings, which must then be recorded manually. However, our EM-31 also has a recorder connector which can be used by an external system to digitally record electrical conductivity readings. The control panel of the EM-31 which contains the analog dial gauge, recorder connector, and other adjustable settings is shown in Figure 3.1.1. Our goal for this project was to use this recorder connector to develop a microcontroller-based system that could digitally and continuously record electrical conductivity readings from the EM-31 and pair them with GPS coordinates. To achieve this, we needed to include a GPS sensor to provide the location data, a microSD card for data storage, and a microcontroller to pull data from the EM-31 and control all the components.

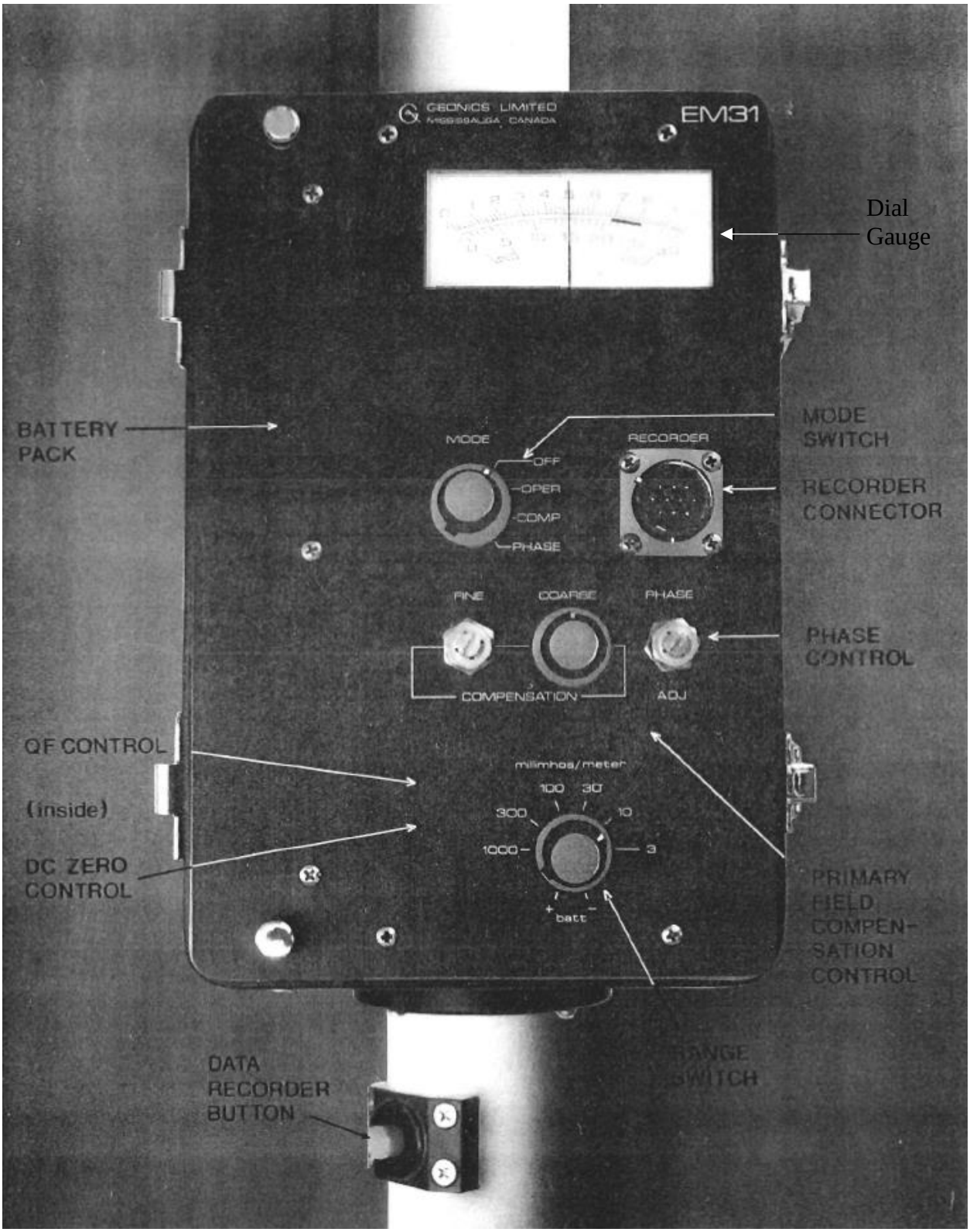


Figure 3.1.1: Control panel of the EM-31 (Geonics Limited, 1991).

In order to pull data from the EM-31 instrument, we needed to identify what information is provided from the recorder connector and how to use that information to determine the electrical conductivity readings. The recorder connector includes 10 pins (Figure 3.1.1). The schematic shown in Figure 3.1.2 describes the output of each of the pins in the recorder connector. To capture conductivity readings, the EM-31 sensor uses a combination of digital signals and analog output. The first step in determining the electrical conductivity is to measure the voltage across pins A and B (that is, the analog output). This voltage can then be converted to electrical conductivity using the following equation.

$$\sigma = V_{A-B} * \frac{Range}{0.5} \qquad (3.1.1)$$

To determine the range used by this equation, the state of pins D, E, and F are used (that is, the digital signals). These pins have two possible voltage states (a low state at which they output 0V and a high state at which they output 4.2V) and unique combinations of the states of each pin are

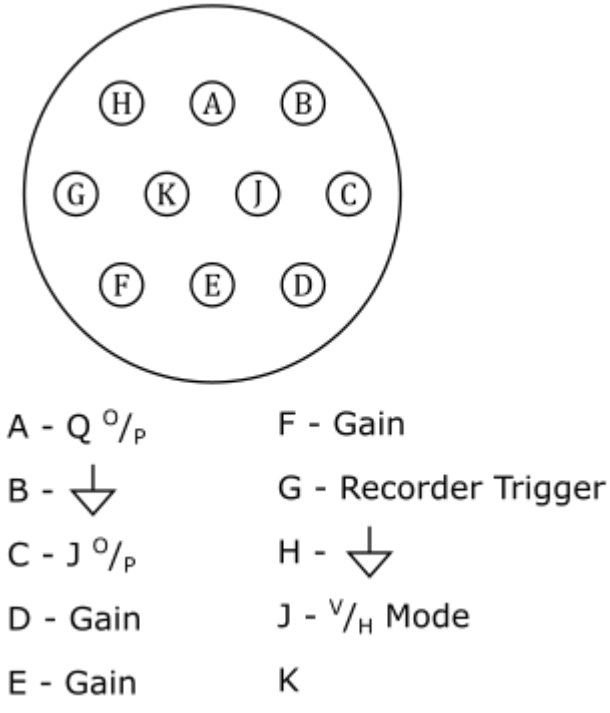


Figure 3.1.2: EM-31 recorder connections.

used to determine the range. The state combinations that correspond to the possible ranges of the EM-31 sensor are described in Table 3.1.1 below. Therefore, by measuring the voltage across pins A and B and determining the state of pins D, E, and F, the electrical conductivity can be determined. It is important to note that the voltage measured across pins A and B only correlates to the electrical conductivity when the appropriate range is selected. If the needle on the analog dial gauge does not fall within the limits of the dial gauge, the measured voltage will not correspond to the electrical conductivity. It is important for the user to monitor the analog dial gauge and adjust the range so that the needle always falls within the limits of the gauge.

In addition to determining the electrical conductivity, the recorder connector can also be used to determine when the data recorder button (see Figure 3.1.1) is pressed. This can be used to identify data that corresponds to specific locations, the time of any significant changes (for example, switching the coils between horizontal and vertical orientations), or any other reason it might be important to identify a specific time or location. Through experimentation, we discovered that pressing the button did not directly cause a change in the recorder trigger pin (pin G), however, when pin G is hooked up to a digital pin, pressing the button indirectly causes the voltage across pins A and B to briefly spike. This behavior can be used to identify when the button is pressed.

Table 3.1.1: State of the three gain pins of the EM-31 for different ranges.

Range	D	E	F
3	HIGH	LOW	LOW
10	LOW	HIGH	LOW
30	HIGH	HIGH	LOW
100	LOW	LOW	HIGH
300	HIGH	LOW	HIGH
1000	LOW	HIGH	HIGH

For our GPS we selected SparkFun's GPS logger shield. This shield provides the geographical location of the measurements and is also equipped with a microSD card slot so that data can be stored to a microSD card. For our microcontroller, we selected the Arduino Uno microcontroller, which is the simplest and most basic microcontroller offered by Arduino. Since the microcontroller system only needs to be able to perform simple functions (measure a voltage, read GPS data, and log data to a microSD card), the selection of these components is not a major concern and the simple components selected should be sufficient. We also included three LEDs which can be used to indicate that the microcontroller system has power, that data are being logged, and whether the button was successfully detected, all of which act as indicators that the microcontroller system is working properly. Figure 3.1.3 below shows these connections and the wiring connections between them. Figure 3.1.4 shows images of the completed microcontroller system and how it is attached and connected to the EM-31 control panel.

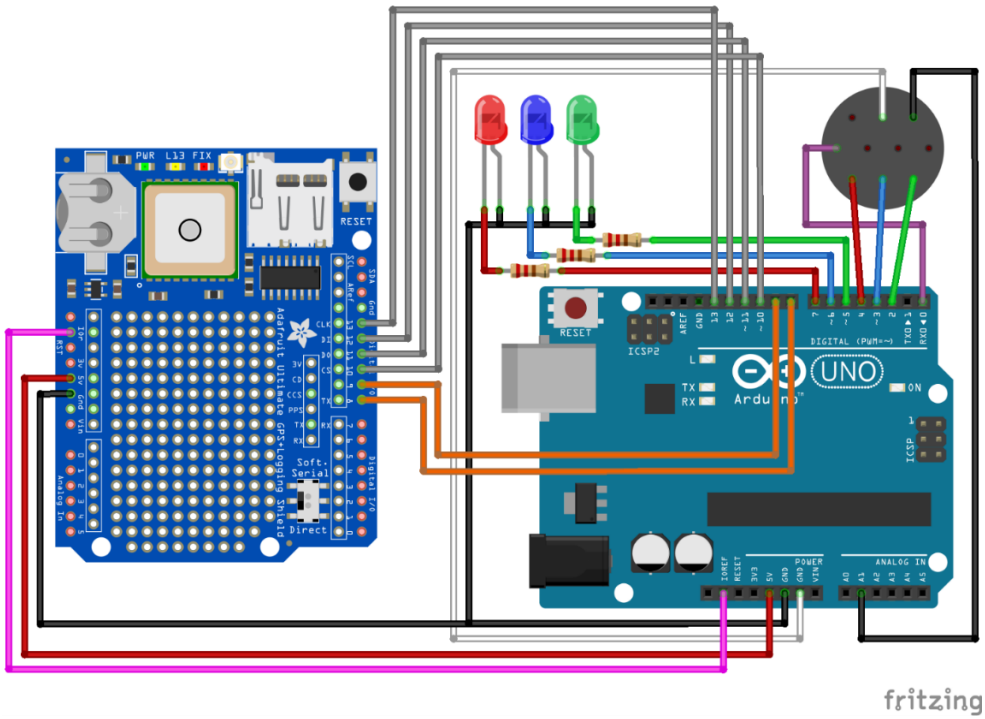
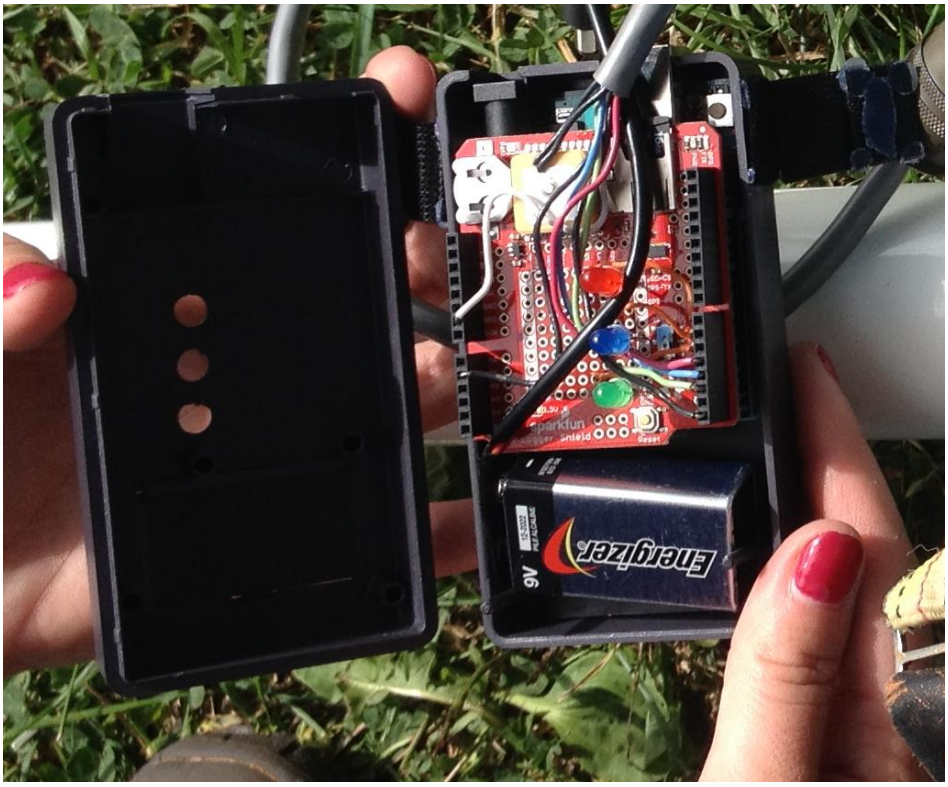


Figure 3.1.3: Diagram of wiring connections for EM-31 microcontroller system (made with Fritzing using parts from Adafruit, 2019 and other unknown sources).



(a)

(b)

Figure 3.1.4: Microcontroller datalogging system built for the EM-31: (a) the contents of the enclosure for the microcontroller system and (b) how the microcontroller system is attached and connected to the EM-31.

3.2 Code Design

The code for the EM-31 instrument needed to perform automatic data logging while simultaneously continuously checking if the data recorder button had been pressed. In both cases, the code needs to read the voltage and range from the EM-31 instrument, convert the voltage to an electrical conductivity, obtain the GPS location and time from the GPS shield, and log this data to the microSD card. The final code for the EM-31 instrument along with a detailed description can be found in the appendix.

3.3 Results

3.3.1 Initial Testing

To ensure the microcontroller system was working properly, we conducted a test to check that the microcontroller system was recording electrical conductivities that matched the readings on the analog dial gauge. This was done by collecting data at Camp Randall Memorial Park (Madison, WI) while occasionally recording the electrical conductivity from the analog dial gauge. Each time a measurement was recorded from the analog dial gauge, the data recorder button was pressed so the EM-31 microcontroller system would indicate which measurements the manual readings should be compared to. The results of this test are shown in Figure 3.3.1. The results are also shown in Figure 3.3.2 plotted spatially in Google Earth using the recorded latitude and longitude from the GPS. The readings from the analog dial gauge are plotted using the latitudes and longitudes that correspond to the matching measurements from the EM-31 microcontroller system when the button was pressed.

The conductivities recorded by the EM-31 microcontroller system and from the analog dial gauge are very similar, though the EM-31 microcontroller system recorded slightly higher values of conductivity for nearly all measurements. There is one conductivity measurement

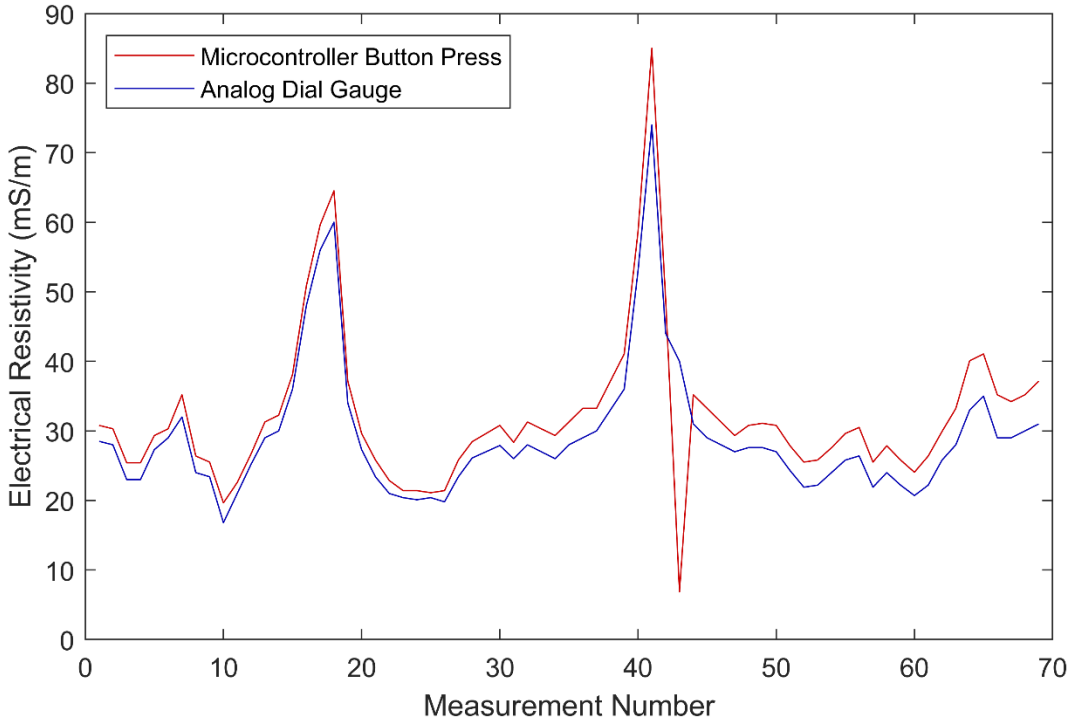
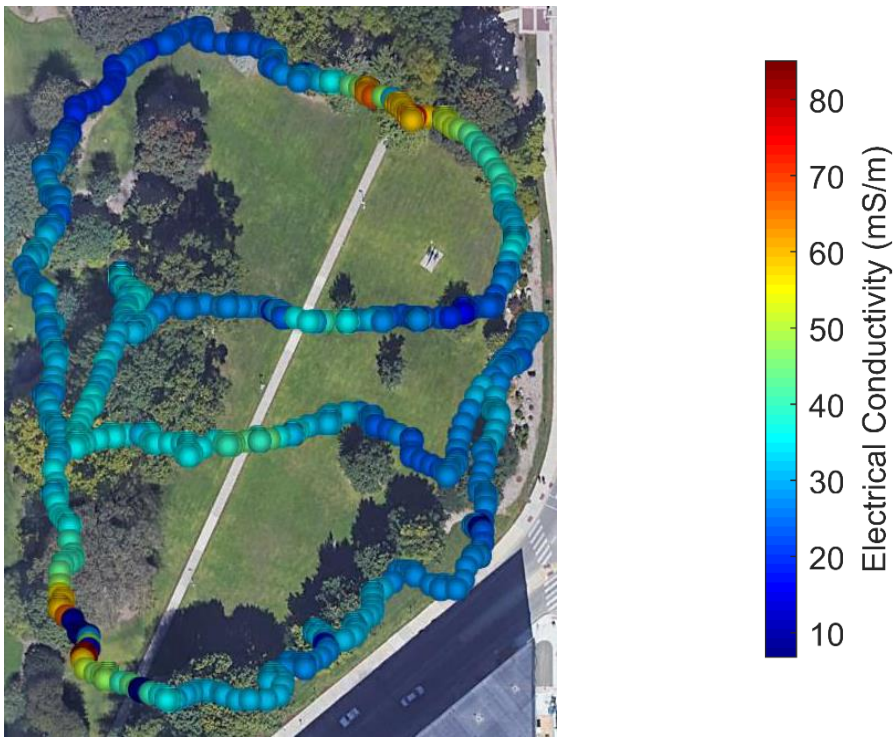
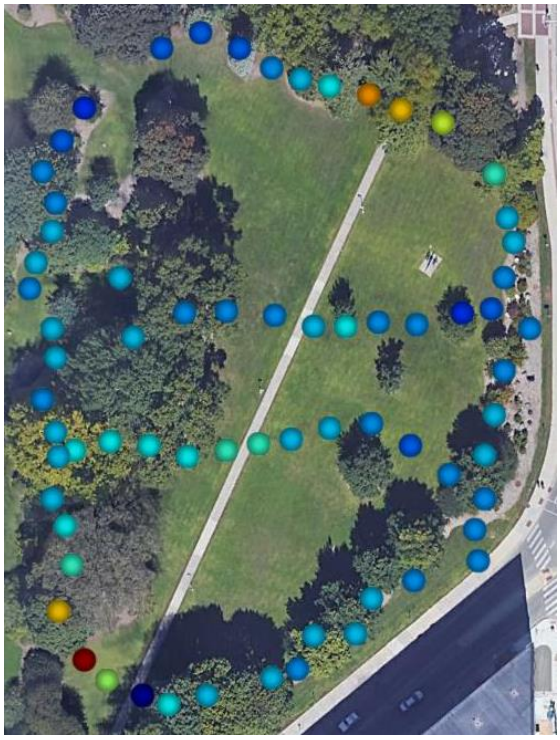


Figure 3.3.1: Comparison of electrical conductivities recorded by the EM-31 microcontroller system and recorded manually from analog dial gauge.

where the EM-31 microcontroller system recorded a lower value than the analog dial gauge reading, however, this value also corresponds with a sudden, steep spike suggesting it may be erroneous. It is also worth noting that the accuracy of the analog dial gauge readings is dependent on the person performing the readings and may be subject to user error. This is especially true since the analog dial gauge tends to fluctuate and sometimes gets stuck. These results show that the microcontroller system is consistently recording electrical conductivity values that seem to be reasonably accurate.



(a)



(b)

(c)

Figure 3.3.2: Results of comparison between microcontroller and analog dial gauge plotted in google earth: (a) all data recorded by the EM-31 microcontroller system, (b) data recorded by the EM-31 microcontroller system upon button press, and (c) data recorded manually from analog dial gauge (plots created using OpenEarthTools open source MatLab software).

3.3.2 Geophysics Field Trip

The EM-31 microcontroller system was also tested during the Fall 2017 geophysics field trip, which took place just north of Crystal Lake near Minocqua, WI. At this site is a large clearing located directly east of a restroom facility. The area serves as a drainage field for two parallel septic mounds running the length of the clearing. Data were collected by walking around the site in a large grid consisting of several lines running approximately parallel and perpendicular to the edges of the clearing. This grid was traversed with the EM-31 sensor oriented in two different orientations: with the coil dipoles aligned in the horizontal direction and with the coil dipoles aligned in the vertical direction. The expectation was that the electrical conductivity would be elevated near the restroom facility where sewage is discharged to the septic mounds. As the sewage is transported along the septic mounds and degraded, we expected the electrical conductivity to decrease.

The data for both the horizontal and vertical dipole orientations showed several large spikes, which likely are the result of bad data points. Therefore, any data points that differed from the mean by more than 20% of the mean were removed. Figure 3.3.3 shows the original electrical conductivity data and the data that was used for analysis with the large spikes removed. The results for the despiked data are shown plotted in Google Earth in Figure 3.3.4 and Figure 3.3.5. As expected, the electrical conductivity is generally high near the restroom facilities and decreases away from the facilities. In addition, the electrical conductivities for the vertical dipole orientation are lower than for the horizontal dipole orientation. This difference is most pronounced in the elevated electrical conductivities near the restroom facilities. Since measurements with the dipoles oriented vertically typically sense an area with a greater depth,

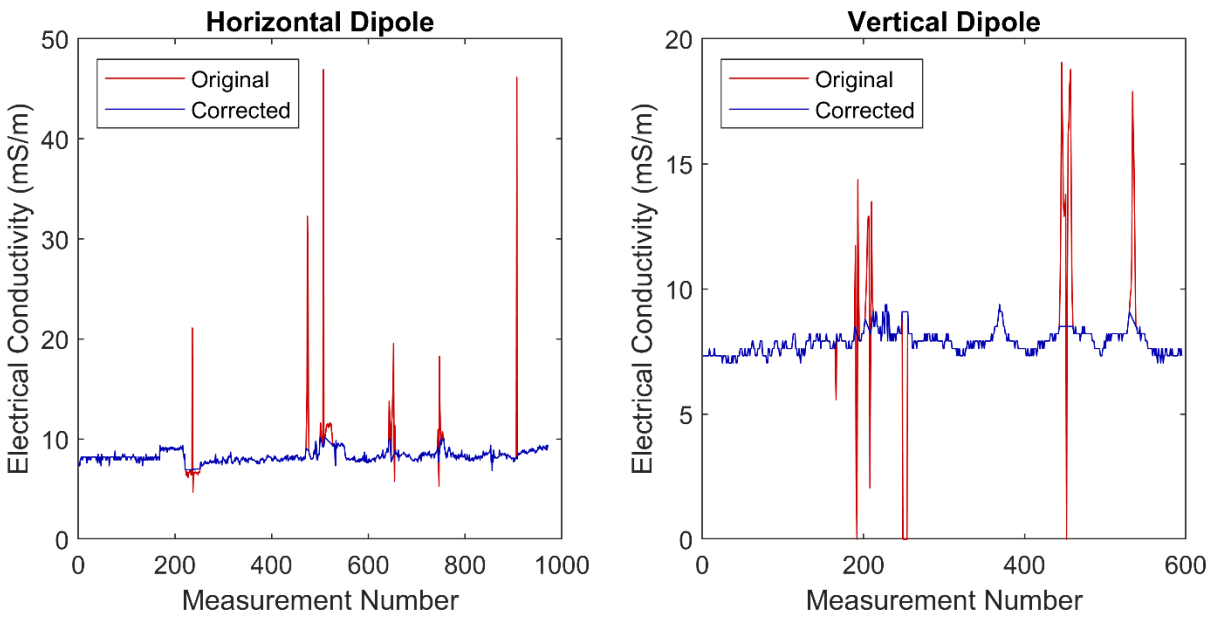


Figure 3.3.3: Electrical conductivity from the geophysics field trip showing data before and after spikes were removed.

this suggests that the electrical conductivity is highest near the surface where the septic mound is located, as would be expected.

3.3.3 Haskell Lake

The EM-31 microcontroller system was also used by Carolyn Streiff and David J. Hart with the Wisconsin Geological and Natural History Survey (Streiff and Hart, 2017) as part of a geophysical survey performed at Haskell Lake in Vilas County. During this survey, data were recorded with the instrument in both a vertical dipole and horizontal dipole orientation while walking around the site. The recorded electrical conductivities are shown mapped in Figure 3.3.6 and Figure 3.3.7. From the vertical dipole data, the WGNHS was able to identify the location of an area with elevated electrical conductivities north of the paved area that may warrant additional investigation. In addition, they identified an area of high conductivity between the hotel and drain field. From the horizontal dipole data, the WGNHS was able to detect that the electrical

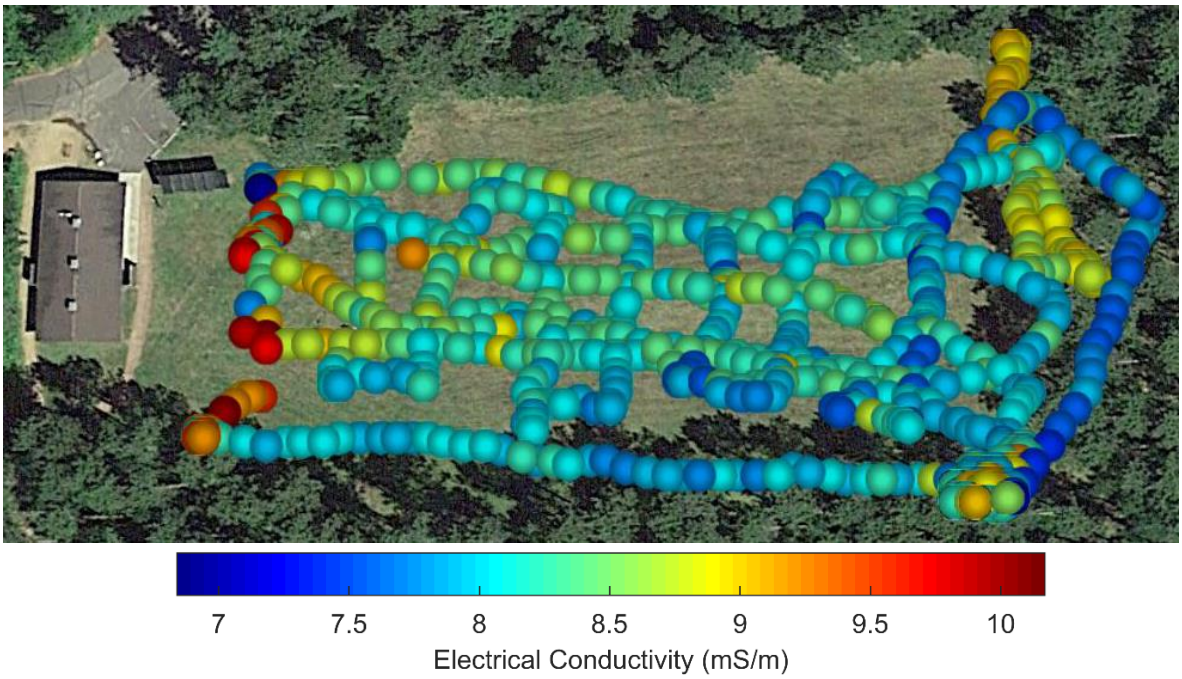


Figure 3.3.4: Horizontal dipole electrical conductivity data plotted spatially in Google Earth (plots created using OpenEarthTools open source MatLab software).

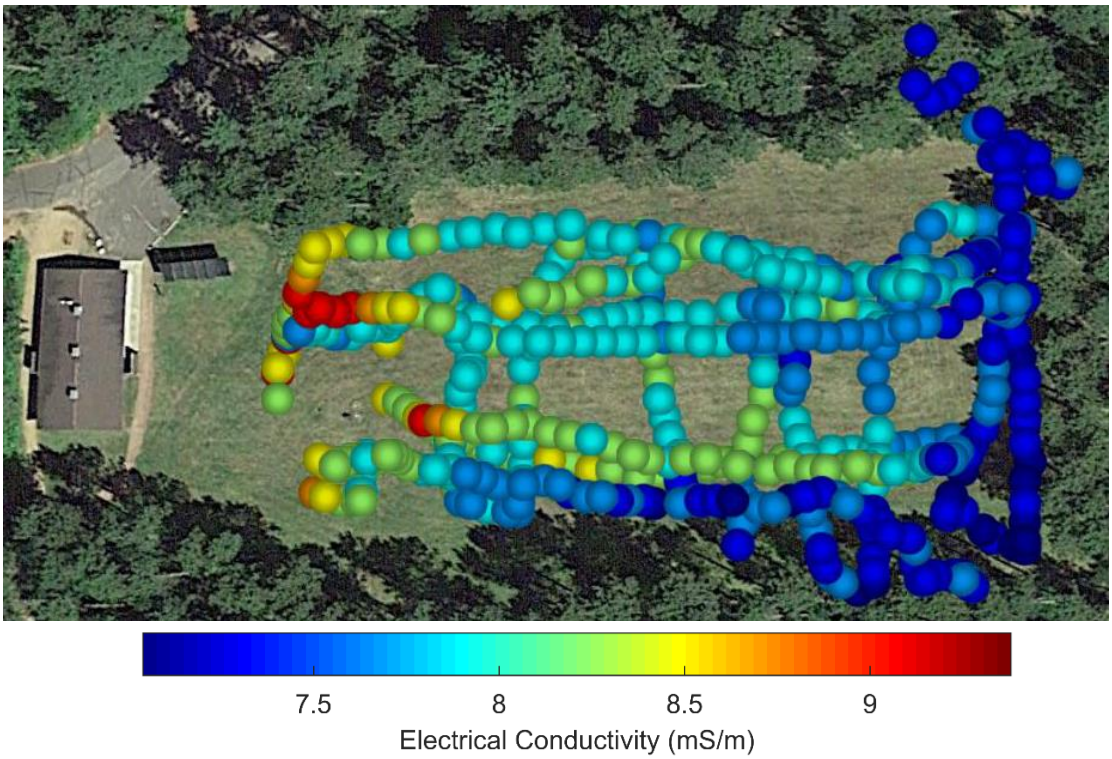


Figure 3.3.5: Vertical dipole electrical conductivity data plotted spatially in Google Earth (plots created using OpenEarthTools open source MatLab software).

conductivity was high at the northern edge of the pavement area, which they believe may be due to placement of road salt.

In addition to using the EM-31 microcontroller system to map the electrical conductivity at the site, the WGNHS also performed three electrical resistivity imaging (ERI) survey lines during their geophysical investigation. They found that in general, the electrical conductivities recorded by the EM-31 microcontroller system matched well with the data collected from the ERI surveys. This comparison was done in terms of general values and trends rather than a comparison of specific data collected at the same location.

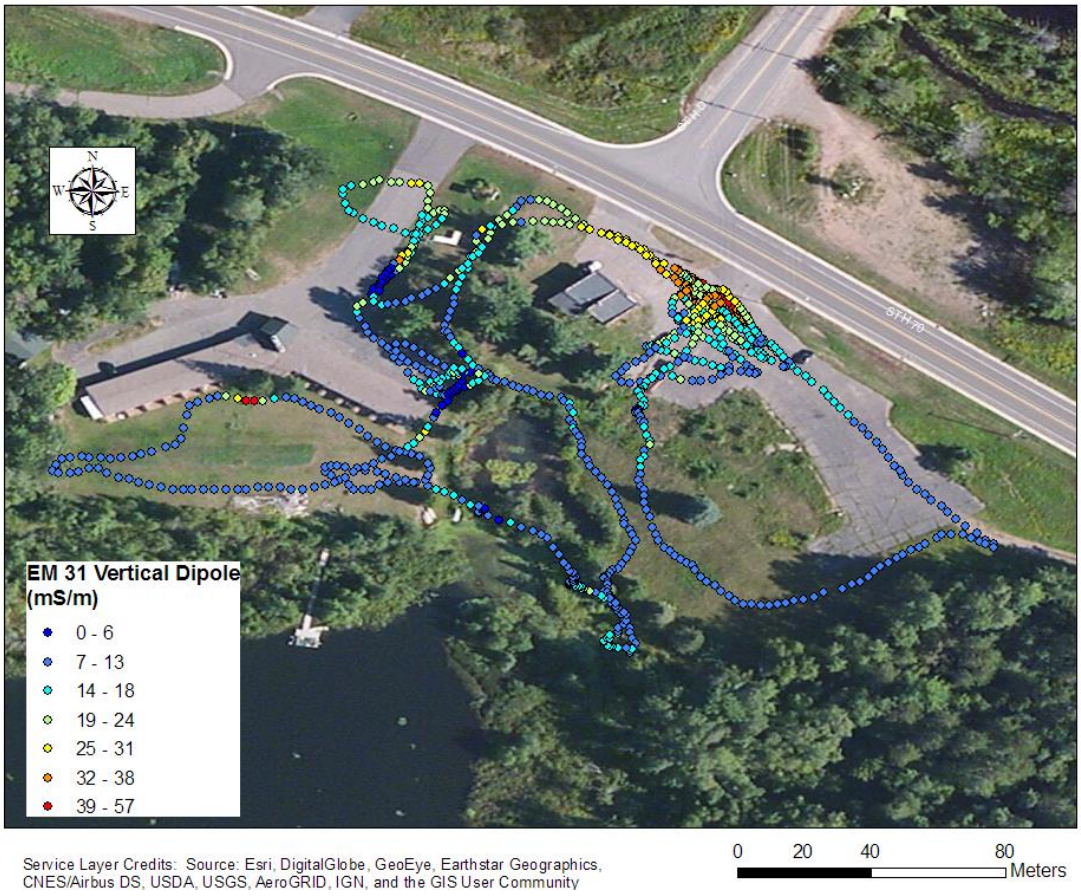


Figure 3.3.6: Vertical dipole EM-31 data (Streiff and Hart, 2017).

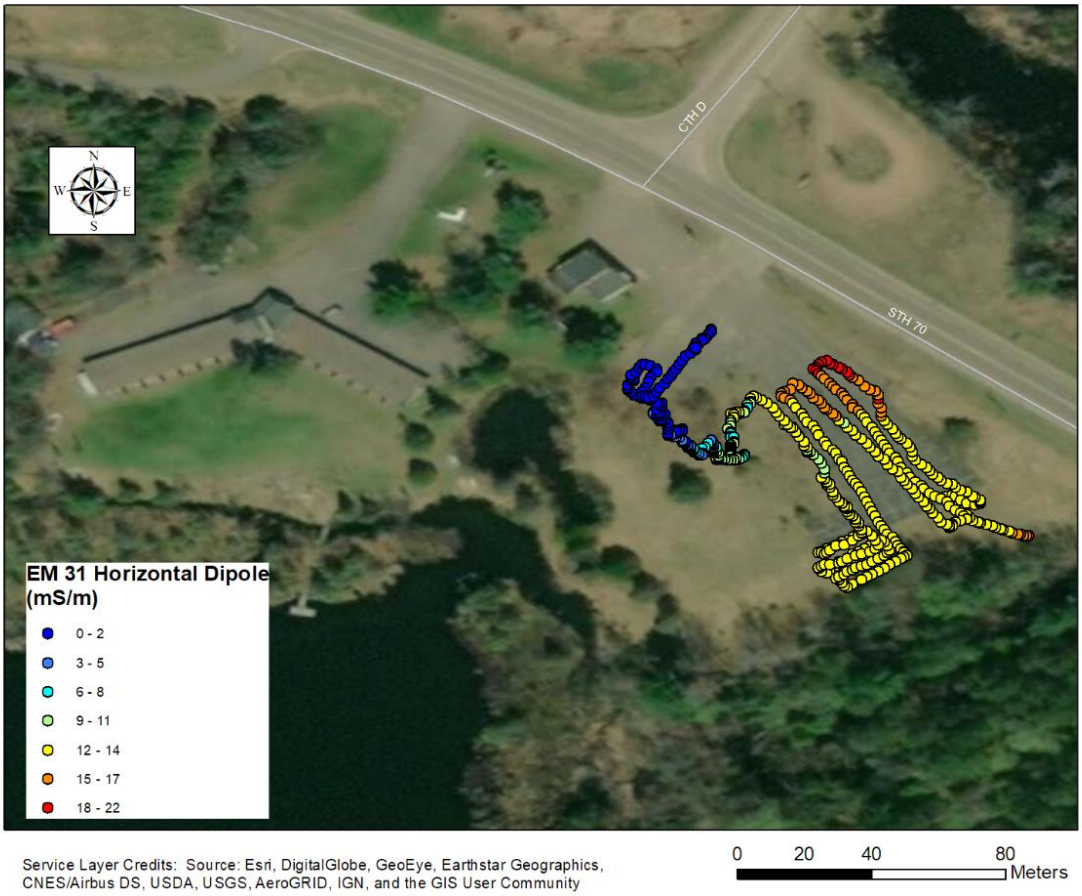


Figure 3.3.7: Horizontal dipole EM-31 data (Streiff and Hart, 2017).

4 Development of Multi-Sensor Array for Water Quality Monitoring

Many different properties and parameters influence the water quality of streams and lakes and are therefore of interest when performing environmental assessments. There are a wide variety of techniques that can be used to measure these properties, discussed in more detail in Chapter 2.3, however, very few systems exist that can measure multiple properties at once. In addition, systems that are capable of measuring multiple properties typically only measure a few. Presented in this section is the development of a system that can measure and record a large variety of water quality properties and parameters.

4.1 System Setup and Wiring

We wanted our water quality monitoring system to be able to measure and record the electrical conductivity, pH, dissolved oxygen, nitrate concentration, chloride concentration, temperature, and turbidity while simultaneously recording the real time and GPS location of each measurement. To achieve this, several different components were required to measure the various properties, provide the real time and GPS position, and store the data. In this section, the various components that were included in the microcontroller system and how they are connected to achieve this goal are discussed.

Probes were selected to measure each of the water quality properties of interest. During our selection we focused on probes that were relatively inexpensive and could easily be interfaced with a microcontroller. In addition, we had to consider the pin connections that potential probes would require to communicate with a microcontroller and the available pin connections of potential microcontrollers. Due to the large number of probes, the available pin connections of the Arduino Mega 2560 microcontroller and Teensy 3.6 microcontroller were considered while selecting probes, while the Arduino Uno microcontroller was not considered

since it is only capable of one serial communication and that serial communication is used to communicate with a computer. The available pin connections for the Arduino Mega 2560 microcontroller and Teensy 3.6 microcontroller are described in Table 4.1.1. Ultimately, we decided to use the Arduino Mega 2560 microcontroller, as is discussed later in this section.

For electrical conductivity, pH, and dissolved oxygen we selected probes offered by AtlasScientific. These probes can easily be interfaced with a microcontroller using EZO™ circuits also offered by AtlasScientific. These circuits can communicate with a microcontroller using either UART serial or I^2C communication and are capable of providing measurements once every second, applying and storing calibration data, and applying temperature compensation, if necessary. In addition, the dissolved oxygen EZO™ circuit is also capable of applying salinity and pressure compensation, if necessary. All three EZO™ circuits are susceptible to electrical noise and the electrical conductivity EZO™ circuit also acts as a source of electrical noise. Therefore, a voltage isolator available from AtlasScientific was chosen to be used along with the pH and dissolved oxygen EZO™ circuits to protect them from electrical noise caused by the electrical conductivity EZO™ circuit.

For nitrate and chloride concentrations, we selected ion-selective electrodes offered by Vernier. For these probes, Vernier offers an analog protoboard adapter that allows these probes to be interfaced with a microcontroller. These probes simply output a voltage that can be read by

Table 4.1.1: Available pin connections of the Arduino Mega 2560 and Teensy 3.6 microcontrollers.

	Arduino Mega 2560	**Teensy 3.6**
Analog Pins	16	25
Digital Pins	54	62
Hardware UART Serial Communication	4	6
SPI Communication	1	3
I^2C Communication	1	4

the microcontroller using an analog pin and then converted to a concentration. However, they require a 5 V power supply, making them incompatible with the Teensy 3.6 microcontroller unless an external power source is used to provide additional voltage. This was one of the leading factors that led us to choose the Arduino Mega 2560 microcontroller for our microcontroller system.

Several temperature sensors were considered for this microcontroller system. Initially we selected a type K thermocouple that could be interfaced with a microcontroller using Analog Devices' K type thermocouple amplifier (AD8495). We selected a thermocouple because of its rapid response time, linear response, and simplicity to use. When used with the thermocouple amplifier, the thermocouple outputs a voltage that can be read and converted to a temperature by the microcontroller. However, we found that when the thermocouple amplifier was used alone and relied on the Arduino Mega 2560 microcontroller's ADC, we were unable to obtain the required resolution needed for field studies. The thermocouple amplifier alters the output of a type K thermocouple so that a change of 1 °C corresponds to a change of approximately 0.005 V. The Arduino Mega 2560 microcontroller's ADC has 10-bits of resolution spread over a 5 V voltage range, therefore, you can calculate the smallest detectable change in voltage to be approximately 0.005 V, as shown in the equation below.

$$\frac{5\ V}{2^{10}} = \frac{5\ V}{1024} \approx 0.005\ V \qquad (4.1.1)$$

This means that the ultimate resolution, limited by the Arduino Mega 2560 microcontroller's ADC, is only about 1 °C. Since we are interested in changes in temperature much smaller than this, we needed to reconsider our choice of temperature sensor.

To improve the resolution of our temperature sensor, we decided to add an external ADC to the thermocouples. In addition, we selected a type J thermocouple, which is rated to have a

better precision than type K thermocouples, as one of the temperature sensors included in our microcontroller system. We continued to use our initial type K thermocouple as the other for comparison. For use with both thermocouples, we selected an ADS1115 16-bit ADC shield created by Adafruit to increase their resolution. For the type J thermocouple, we used the same thermocouple amplifier as for the type K thermocouple. Even though this thermocouple amplifier is designed for use with type K thermocouples, we determined that it could also work for type J thermocouples since the response of both thermocouples is very similar. Therefore, the thermocouple amplifier will still convert the output of the type J thermocouple to a readable value and the differences between the two thermocouple types can be accounted for in calibration. The ADS1115 16-bit ADC shield uses I^2C communication to communicate with a microcontroller. We also considered two additional temperature sensors: a thermistor provided by Vernier and a digital thermometer (DS18B20). We knew that the DS18B20 thermometer works well, however, the thermocouple with the ADS1115 16-bit ADC shield showed a large improvement in precision of temperature measurements. In addition, the DS18B20 thermometer had a compatibility issue between the 1-Wire communication required for the DS18B20 and the software serial communication used by the GPS shield. Ultimately, we decided to favor the faster response times of the thermocouples.

To measure turbidity, we selected the gravity analog turbidity sensor offered by DFRobot which comes with an adapter for use with a microcontroller. The adapter sends an analog signal from the turbidity sensor to an analog pin of the microcontroller. The microcontroller can then record the voltage output of the sensor which can be correlated to the turbidity. This is a very inexpensive turbidity sensor that can only provide a rough approximation of the turbidity. For more sophisticated measurements, a higher quality, higher cost sensor could be selected.

We considered two different GPS modules for our microcontroller system, both of which came on shields that also provided access to a microSD card slot. These are SparkFun's GPS logger shield and the Ultimate GPS logger shield offered by Adafruit. Both GPS shields are almost identical in function, in that they use the same number and type of connections to communicate with the microcontroller, however, the Ultimate GPS logger shield provides a connection that allows an external antenna to be hooked up to the GPS shield while SparkFun's GPS logger shield does not. In addition, SparkFun's GPS logger shield had difficulty acquiring GPS signals when surrounded by the wires and circuitry connecting the components of the microcontroller system. Therefore, we selected the Ultimate GPS logger shield for our microcontroller system. The Ultimate GPS logger shield provided us with GPS location for all our measurements, the real time for each measurement, and a means of storing the data on a microSD card.

As mentioned previously, we chose to use the Arduino Mega 2560 microcontroller for our microcontroller system. There were three reasons we made this decision. First, the Vernier probes (nitrate and chloride) required a 5 V voltage supply. The Arduino Mega 2560 microcontroller supplies 5 V to external components while the Teensy 3.6 microcontroller supplies 3.3 V. Second, the Arduino Mega 2560 microcontroller has sufficient pin connections available to accommodate all of our selected probes, therefore, the additional pin connections of the Teensy 3.6 microcontroller were not necessary. Finally, rapid data collection was not a major concern for this project, so we preferred using the Arduino Mega 2560 microcontroller and not having to use an external power supply over the increased processing speed of the Teensy 3.6 microcontroller. A summary of the pin connections used to communicate between the Arduino Mega 2560 microcontroller and all the chosen components of the microcontroller system can be

found in Table 4.1.2. In addition to these communication connections, each of the components also had to be connected to the power and ground. A schematic showing the wiring necessary to connect all the components of the microcontroller system is shown in Figure 4.1.1. When wiring the components together, wires were twisted or braided together whenever possible to reduce the effects of electrical noise. Additional details and specifications of the microcontroller system components can be found in Chapter 2.

Table 4.1.2: Summary of communications and pin connections used on the Arduino Mega 2560 microcontroller for all components included in Multi-Sensor Array.

Component	Communication Type	Pin Type	Pin
Electrical Conductivity	HW UART Serial	TX	18
		RX	19
pH	HW UART Serial	TX	16
		RX	17
Dissolved Oxygen	HW UART Serial	TX	14
		RX	15
Chloride Concentration	Analog		A0
Nitrate Concentration	Analog		A1
Analog to Digital Converter	I^2C	SDA	20
		SCL	21
Type J Thermocouple	Analog		A0 (ADC)
Type K Thermocouple	Analog		A1 (ADC)
Turbidity	Analog		A3
GPS	Software Serial	TX	11
		RX	10
microSD	SPI	MISO	50
		MOSI	51
		SCK	52
		SS	53
Power LED	Digital		2
GPS Data Logged LED	Digital		3
Data Logged LED	Digital		4

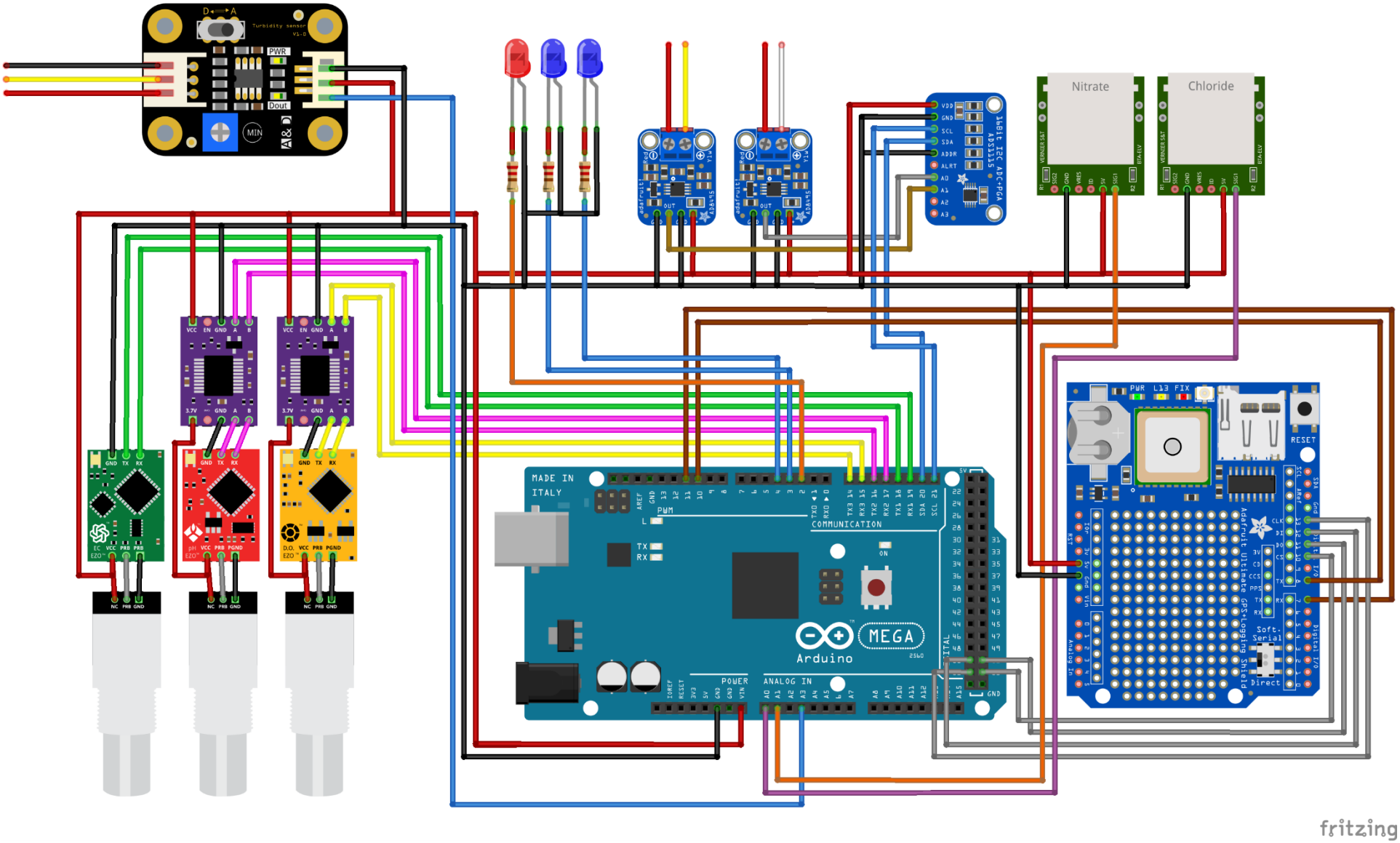


Figure 4.1.1: Diagram of wiring connections for Multi-Sensor Array (made with Fritzing using parts from Adafruit, 2019 and other unknown sources).

To protect the electrical components and wiring, we needed an enclosure that could contain and protect the microcontroller system while still providing access to the plugs and adapters necessary to connect the probes and sensors to the microcontroller system. We initially used a generic plastic box with a top that could be screwed in place with six screws. However, we decided to move the microcontroller system to a case that had easier internal access. While moving the microcontroller system, we also looked for a more water-resistant case and elevated the wiring by placing components on top of a foam block to mitigate the potential for water damage to the microcontroller system. We did this because of some issues we had with the components on the bottom of the previous enclosure being exposed to water that found its way into the box. Based on this, we selected a Pelican case, which have proven to be durable and very water-resistant.

When creating our enclosure, we wanted to ensure that the probes and sensors were not permanently connected to the enclosure so that they could be separated and left in a flow through cell (discussed in the following section) for storage. This meant that each of the probes needed to have some kind of connector and matching plug that we could mount into the side of the enclosure. For the AtlasScientific probes and Vernier probes, this was not an issue, since each probe came with a connector attached on the end. The AtlasScientific probes came with BNC connecters and plugs that could easily be connected to their respective EZO^{TM} circuits and the Vernier probes came with BTA connectors that plug directly into Vernier's analog protoboard adapters. The type J thermocouple came with a miniature thermocouple connector, so we could easily use the matching plug to connect it to the microcontroller system. Our type K thermocouple, however, did not come with a connector. Since we planned on using the type J thermocouple inside the flow through cell and the type K thermocouple outside, we decided that

the type K thermocouple could be directly connected to the microcontroller system. Since the type K thermocouple does not go into the flow through cell, we could store it with the enclosure with a permanent connection. The turbidity sensor came with a three-pin connector that plugs directly into the sensor's adapter, however, the cable was far too short for our purposes. Therefore, we extended the cable by cutting the wires connecting the sensor to the three-pin connector, soldering wires to the cable to increase the length, and attaching our own plug and connector to the two separated ends. We used a six-pin plug and connector, then used two of the extra pins to connect the metal tubing of the flow through cell, discussed in the following section, to the microcontroller system's ground.

To mount the plugs in the side of the enclosure, holes were drilled or cut into the enclosure to fit the sizes of the various plugs. In addition, holes were drilled or cut to allow the USB Type B and power connectors on the Arduino Mega 2560 microcontroller to be accessible from the outside of the enclosure while closed. Also, three holes for LEDs were drilled to include a visual indication of when the microcontroller system has power and when GPS and sensor data are successfully logged while the enclosure is closed and a computer is not connected. Lastly, an additional hole was drilled to allow an external antenna to be connected to the Ultimate GPS logger shield. The various plugs, connectors, and LEDs were held in place by either a nut if they were threaded (BNC plugs, six-pin connector for turbidity, and external GPS antenna) or using either hot glue or silicone glue. A picture showing the completed enclosure with the microcontroller system wiring in place can be found in Figure 4.1.2.

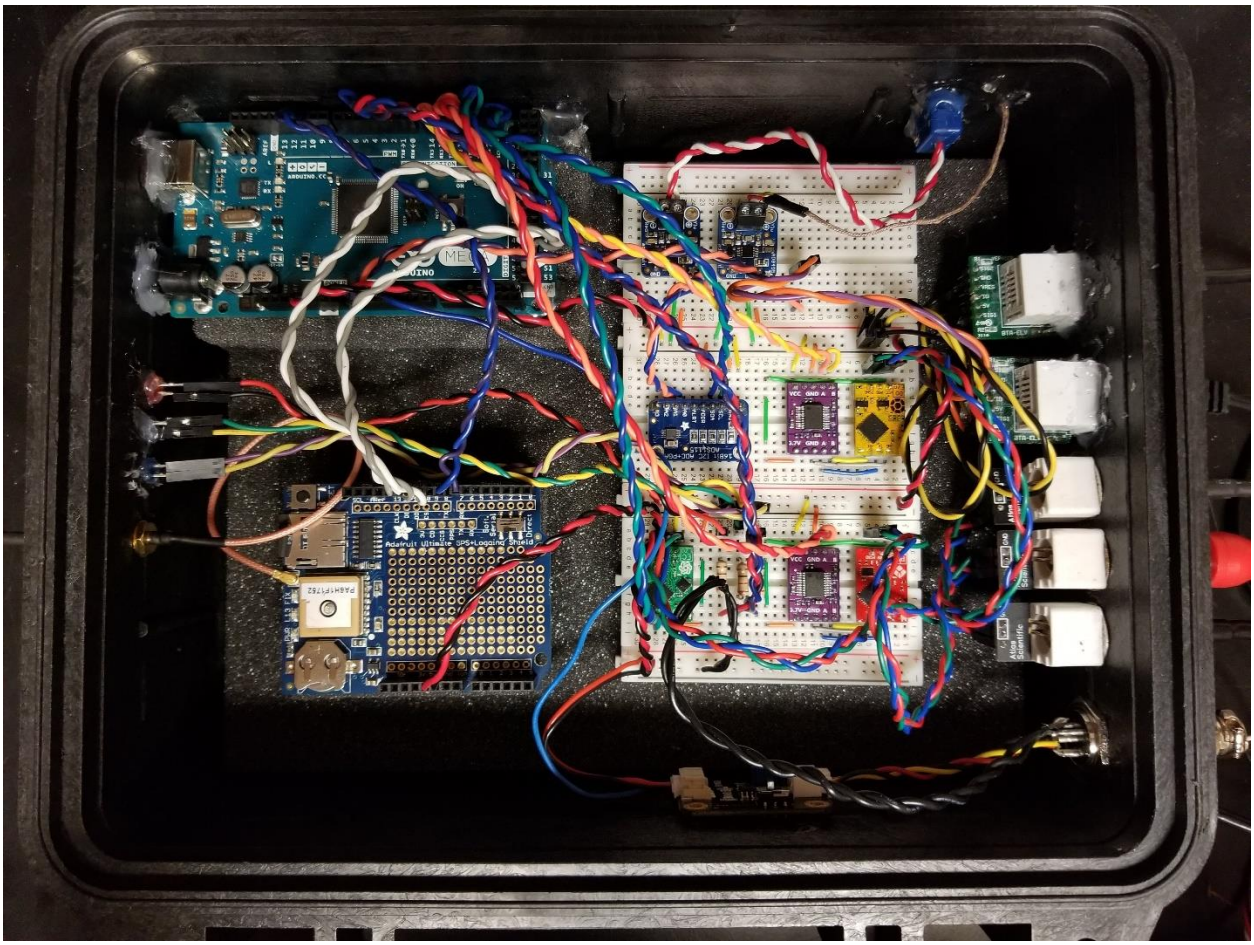


Figure 4.1.2: Picture of completed, final enclosure and wiring.

4.2 Flow Through Cell Design

We considered two options for how to employ the sensors in the field. First, we considered attaching the sensors to some apparatus that can be attached to the side of a canoe. For this method to work, the sensors would have to be attached in such a way that an impact with any obstacles would not damage the probes. In addition to large obstacles that may damage the probes, we worried that vegetation might get tangled up with the probes and make this method impractical. Therefore, we decided to go with our second option, which was to create a flow through cell that could fit all the probes and pump water from the body of water of interest through the cell. This option allows us to keep the probes relatively protected and limits the

components that must be attached outside of the canoe where they may become entangled in vegetation.

While designing a flow through cell, we needed to take into consideration the potential for interference between different probes. Specifically, we were concerned that the electrical conductivity probe and the ion selective electrodes for nitrate and chloride may cause interference with one another. To test if there was interference between these probes, we tested them in pairs to see if adding and removing one probe from a beaker of water would cause the readings of the other probe to vary significantly. We found that each probe did, in fact, noticeably change the readings of the other probes, therefore, when designing our flow through cell we needed some method of isolating these three probes.

To isolate these probes, we designed the flow through cell to include three separate chambers. This way, each of the probes that caused interference could be inserted into a different chamber and therefore separated from one another. The chambers can then be connected to each other using rubber tubing to allow water to flow from chamber to chamber through the entire cell. In order to ensure the chambers continued to be electrically isolated, we included a section of small, metal pipe along the tubing between chambers that was connected to the microcontroller system's ground. This way, electrical current should not be able to travel between chambers and cause interference and only one pump is required to pump water through all three chambers.

To minimize the time it takes water to flow through the cell, and therefore the delay between measurements and geographical locations, we designed the cell so it would be just large enough to fit the probes with the caps or bottles needed for storage and the depth was sufficient for readings to be accurate. This way, the probes could be left inside the flow through cell for

long-term storage and we could minimize any potential damage that may occur by inserting and removing the probes frequently. The final, inner dimensions of each of the chambers was 33.33 mm by 90 mm with a depth of 80 mm.

A diagram depicting the final design of the flow through cell is shown in Figure 4.2.1. The cell consists of three separate chambers with removable lids that allow access to the bottoms of the probes. The lids are held shut using hand screws that can be tightened or removed in the field without the use of a tool. The lids are sealed with a rubber gasket that runs around the entire perimeter of each chamber. The probes are inserted into the chamber through openings in the lids. To ensure the chambers remain sealed, each of the openings contains an O-ring which provides a seal around the probes. Plastic hose barbs were used for the inlets and outlets so that plastic tubing could be attached. The inlets were placed near the bottom of the chamber while the outlets were placed near the top to promote flow across the entirety of the chambers and to try to minimize areas in the chamber that may remain stagnant and may not be replaced with fresh, incoming water. A peristaltic pump is used to pump water from the stream or lake of interest into and through the flow through cell. The pump was located before the first inlet and the outflow from the final outlet can be directed by plastic tubing to discharge back into the stream or lake.

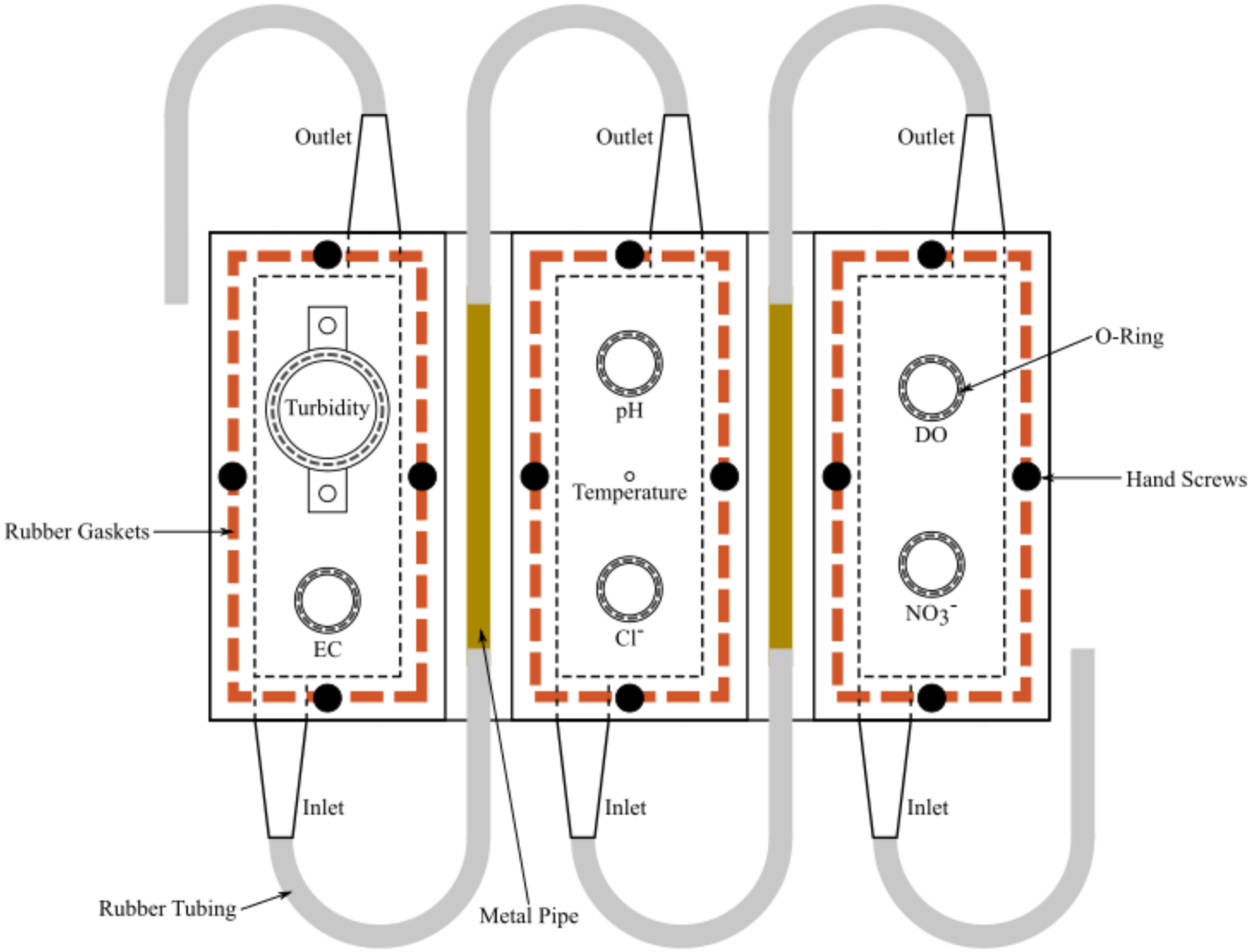


Figure 4.2.1: Depiction of flow through cell design (top view).

4.3 Code Design

There were two functions that the code for our microcontroller system needed to perform: calibration and data logging. While we initially included both of these functions in the same code, to try to simplify the code we ultimately split the code into two separate codes, one for calibration and one for data logging.

4.3.1 Calibration

Calibration was required for the electrical conductivity, pH, dissolved oxygen, nitrate, and chloride probes as well as for the thermocouples. The turbidity sensor does not require calibration; only the output voltage of the turbidity sensor would be logged because of the qualitative nature of the sensor and the difficulty in coding that would be required to relate the measured voltage to turbidity. After data logging in the field, the recorded output voltage of the turbidity sensor can be roughly related to turbidity using the recorded temperature and Figure 2.3.14. For the sensors requiring calibration, there were two ways that the calibration needed to be coded. First, for the nitrate probe, chloride probe, type J thermocouple, and type K thermocouple the code needed to take care of applying and storing the calibration parameters. For the AtlasScientific probes (electrical conductivity, pH, and dissolved oxygen), however, the EZO™ circuits that came with the probes can apply and store the calibration parameters, so the coding needed for these probes only had to be able to send the appropriate commands to these circuits.

To achieve both of these types of calibrations, the code needed to accept serial commands sent to the computer. These commands were then used to tell the code whether to start a calibration sequence for the nitrate probe, chloride probe, type J thermocouple, or type K thermocouple or if commands should be sent to the electrical conductivity, pH, or dissolved

oxygen probes. A summary of the commands built into the code to complete these actions and the most useful commands for the AtlasScientific probes can be found in Table 4.3.1.

If the code detects a command to start a calibration sequence for nitrate, chloride, type J thermocouple, or type K thermocouple it will call a function that performs the calibration for the appropriate sensor. These functions are nearly identical but use different variable and file names to store and apply the calibration parameters and the equations used to calculate and apply the calibration parameters are different for the ion-selective electrodes and the thermocouples. In each case, the code walks the user through the steps necessary to develop a linear relationship between the output voltage and the parameter or natural log of the parameter of interest. The microcontroller system must store four different values: the lower and higher parameter values being calibrated (i.e. the known standards) and the output voltages corresponding to each. These values are stored in files on the microSD card so they can be accessed by the microcontroller to calculate the calibration parameters. The final calibration code along with a detailed description can be found in the appendix.

4.3.2 Data Logging

The data logging code needed to be able to collect data from each of the sensors, convert raw data into the parameters we were interested in (for example, converting raw voltage to a temperature value), store the data on the microSD card, and print the data to the serial monitor so the user can monitor the parameters using a computer while data logging occurs, if desired. The final data logging code along with a detailed description can be found in the appendix.

Table 4.3.1: Summary of commands.

Type		Command	Action
Built into Code		EC	Switch selected AtlasScientific probe to electrical conductivity probe.
		pH	Switch selected AtlasScientific probe to pH probe.
		DO	Switch selected AtlasScientific probe to dissolved oxygen probe.
		n cal	Begin the nitrate calibration sequence.
		c cal	Begin the chloride calibration sequence.
		tJ cal	Begin the type J thermocouple calibration sequence.
		tK cal	Begin the type K thermocouple calibration sequence
AtlasScientific	EC	cal,dry	Dry calibration (must do first).
		cal,low,n	Calibrate low value (n = conductivity µS/cm, ex. 12880).
		cal,high,n	Calibrate high value (n = conductivity µS/cm, ex. 80000).
	pH	cal,mid,x.xx	Calibrate mid value (performed first, should always be 7, x.xx is pH, ex. 7.00).
		cal,low,x.xx	Calibrate low value (performed second, x.xx is pH, ex. 4.00).
		cal,high,xx.xx	Calibrate high value (performed last, xx.xx is pH, ex. 10.00).
	DO	cal	Dry calibration (must do first).
		cal,0	Zero DO calibration (only need for < 1 mg/L).
		O,%,0	Disable percent saturation.
		O,%,1	Enable percent saturation.
		O,mg,0	Disable concentration (mg/L).
		O,mg,1	Enable concentration (mg/L).
		O,?	Check which parameters are enabled.
	Shared	c,0	Enable continuous readings.
		c,1	Disable continuous readings.
		c,?	Check if continuous mode is enabled (?c,0 = continuous mode enabled, ?c,1 = continuous mode disabled).
		t,xx.x	Input temperature compensation.
		t,?	Check current temperature compensation.
		cal,clear	Clear calibration data.
		cal,?	Check calibration (?cal,0 = not calibrated, ?cal,1 = single point calibration, ?cal,2 = two point calibration, ?cal,3 = three point calibration).
		response,1	Enable response.
		response,0	Disable response.

4.4 Field Testing

The microcontroller system was used by Catherine Christenson and David J. Hart for a variety of stream and lake surveys. The following results are from a stream survey performed on June 27, 2018 at the Mukwonago River (WI) between a spring discharge pool and Lulu Lake (WI). The microcontroller system was used to record data while canoeing the river in the upstream direction. The water quality properties recorded during this survey are shown mapped in Figure 4.4.1 through Figure 4.4.6.

From these results, they were able to identify the groundwater fed portion of the stream (located on the west side of the figures near the spring pool). This groundwater fed portion of the stream showed lower temperatures, higher fluid conductivities, and higher dissolved oxygen contents than the rest of the stream. Their interpretation that higher fluid conductivity and dissolved oxygen values are indicative of groundwater entering the stream are supported by theories that the groundwater flow in the area is driven by fracture flow. They also found a slight increase in nitrate and chloride levels in and near the spring pool, which Christenson and Hart (personal communication) hypothesized may be the result of the agricultural areas surrounding the stream. Finally, they were able to identify changes in some of the water quality parameters where the tributary enters the Mukwonago River (WI) and in Lulu Lake (WI), which is expected when entering into or flow is introduced from different bodies of water.

Turbidity data were not included because a filter was used at the inlet to prevent debris or sediment from clogging the flow through cell or damaging the probes. However, during other surveys, in the rare instances where a filter was not used, there were trends in the measured voltage that could potentially be used to identify trends in turbidity.

While the nitrate and chloride data from this survey are reasonable and correlate well with the context of the site, overall at many of the sites they observed issues with the nitrate and chloride values being unrealistic. They suspect that variations in temperature along the stream, which can influence the readings from the nitrate and chloride probes, may be contributing to these unrealistic values. Because the nitrate and chloride probes are intended for laboratory use, they are designed to be calibrated and take measurements at one, constant temperature. Therefore, the significant changes in temperature that can occur in field conditions are likely causing the nitrate and chloride values to be inaccurate. In addition, they suspect that the probes themselves are not well suited for field use and may be contributing to unrealistic values.

At many of the sites, Christenson and Hart (personal communication) traversed the same path with the microcontroller system to obtain duplicate measurements. One example of this is shown in Figure 4.4.7. This figure shows the fluid conductivity that was recorded at Plainfield Lake (WI) on September 20, 2018. This figure shows a good comparison between the values of fluid conductivity recorded during both loops around the circumference of the lake. The results of many of the stream and lake surveys showed good repeatability in the measurements, a good indication that the microcontroller system works reliably.

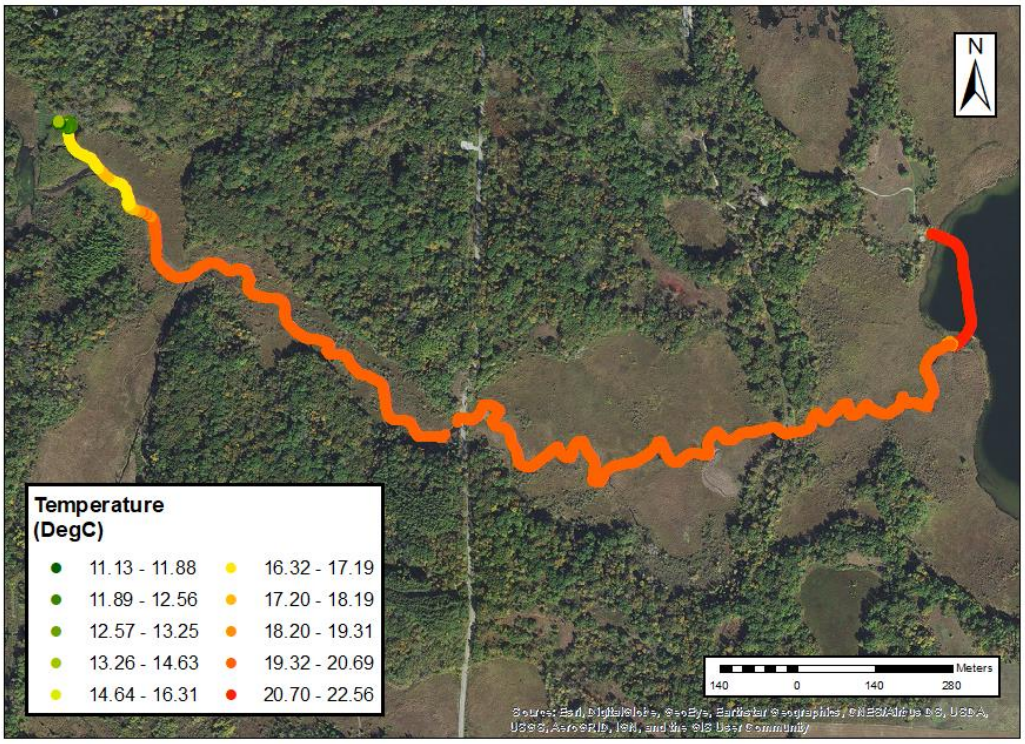


Figure 4.4.1: Temperature data from the Mukwonago River (Christenson and Hart, personal communication).

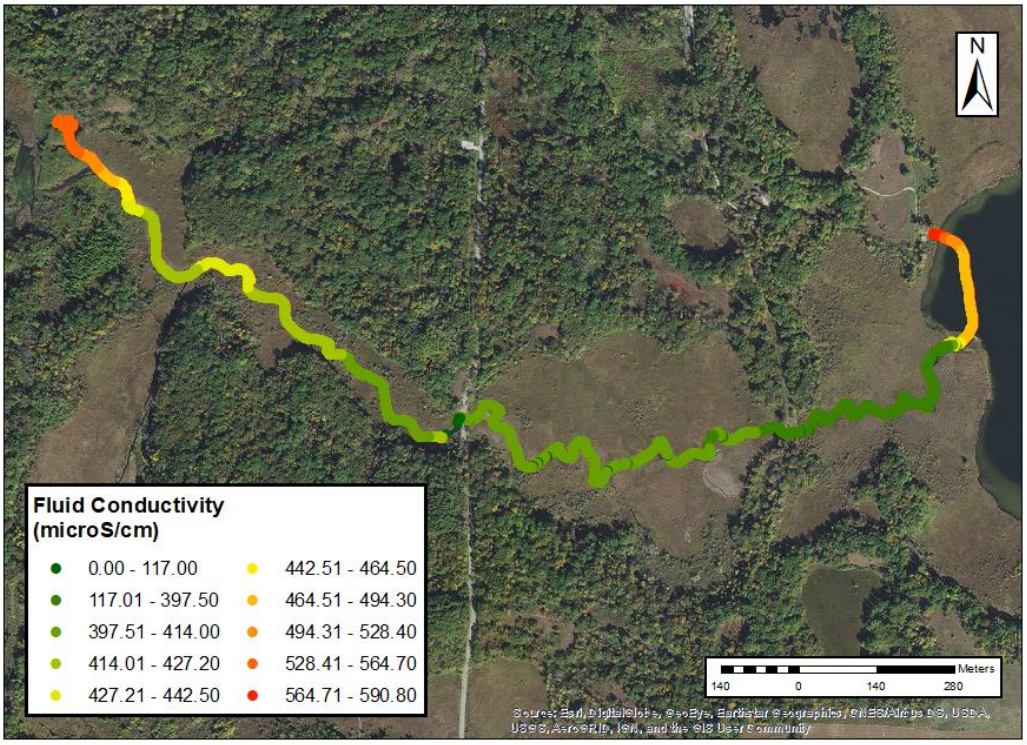


Figure 4.4.2: Fluid conductivity data from the Mukwonago River (Christenson and Hart, personal communication).

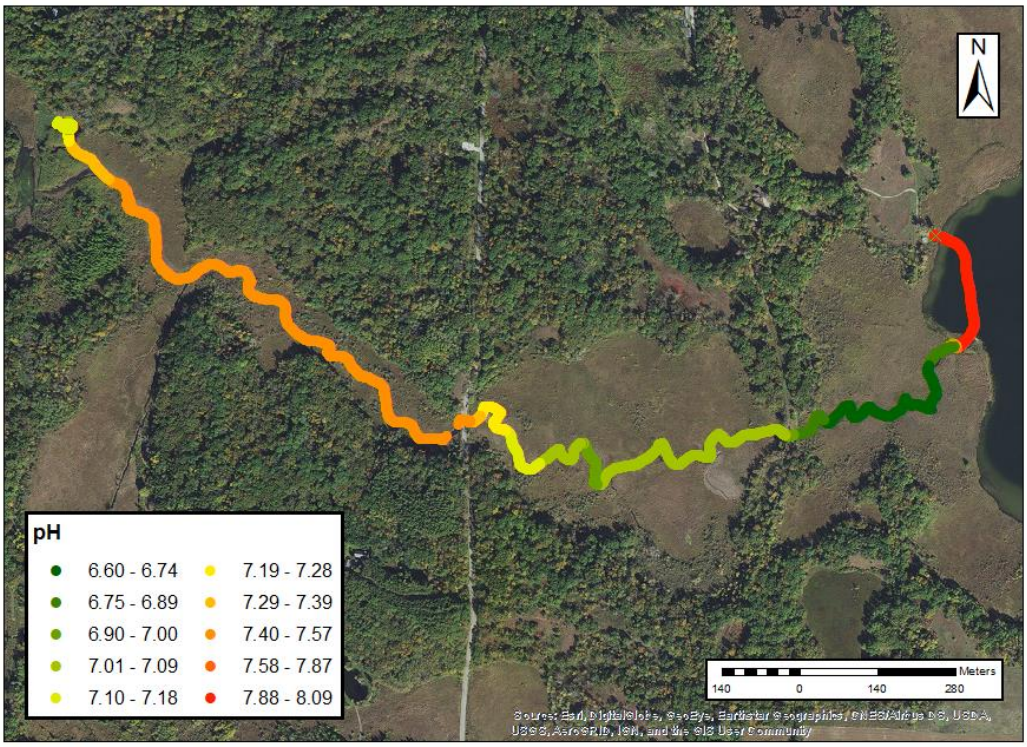


Figure 4.4.3: pH data from the Mukwonago River (Christenson and Hart, personal communication).

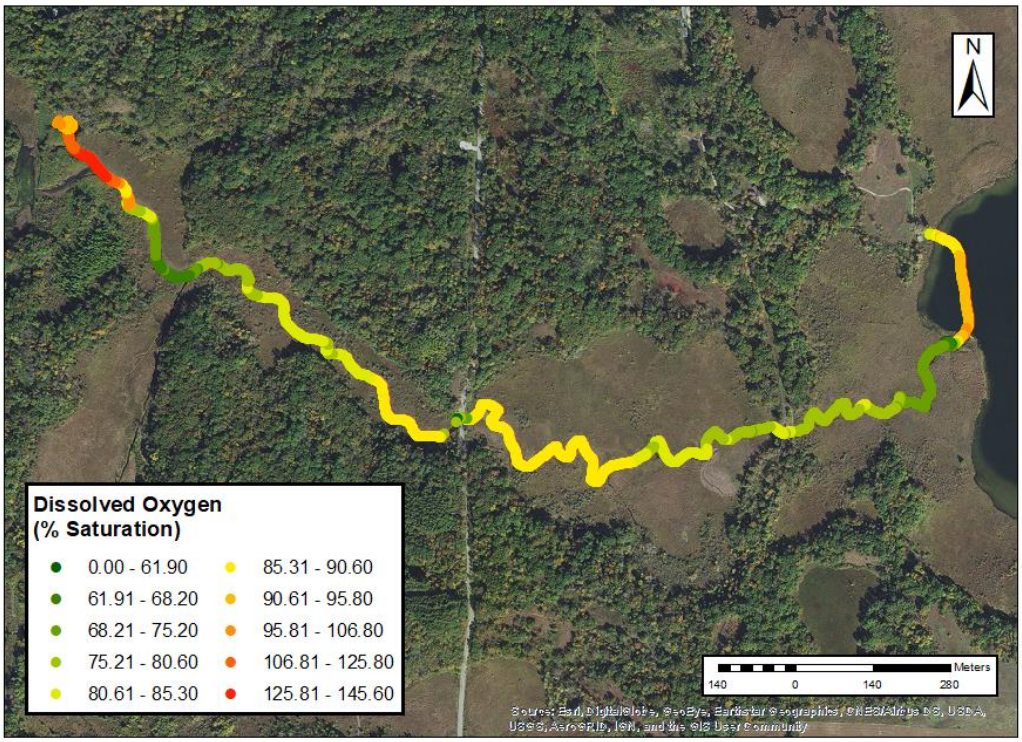


Figure 4.4.4: Dissolved oxygen data from the Mukwonago River (Christenson and Hart, personal communication).

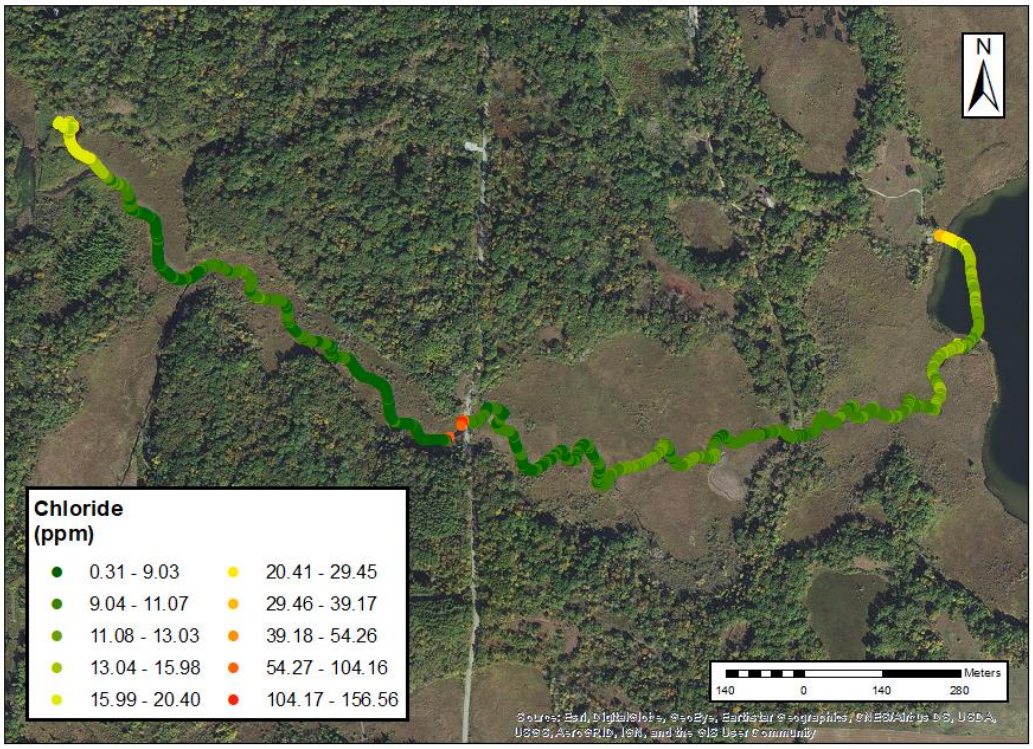


Figure 4.4.5: Chloride data from the Mukwonago River (Christenson and Hart, personal communication).

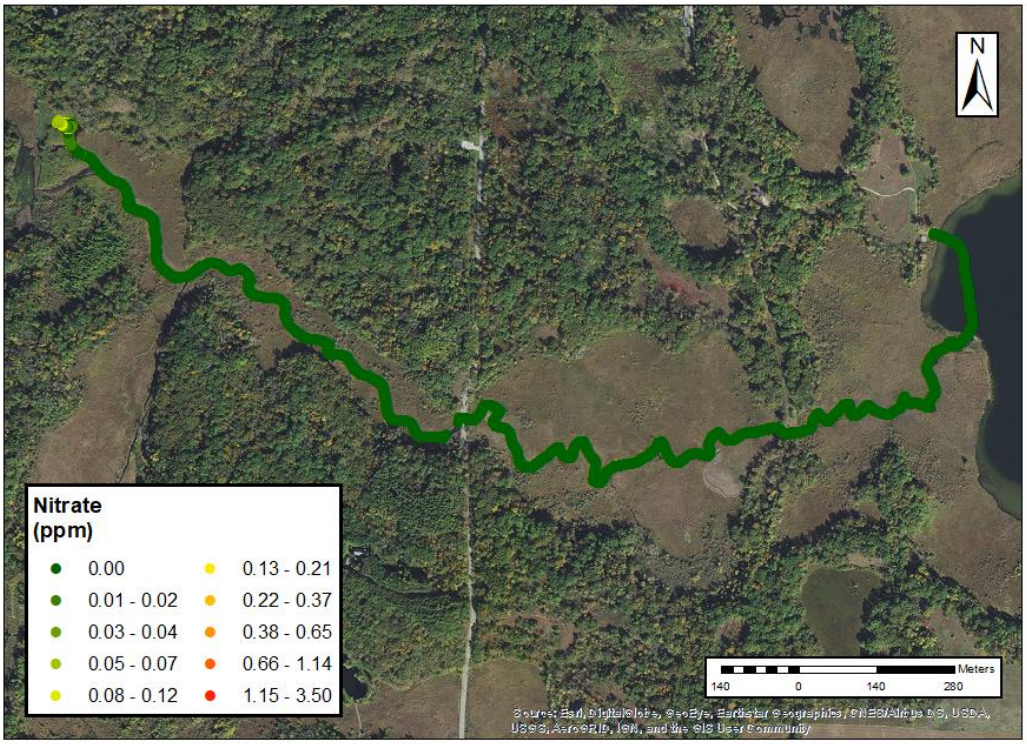


Figure 4.4.6: Nitrate data from the Mukwonago River (Christenson and Hart, personal communication).

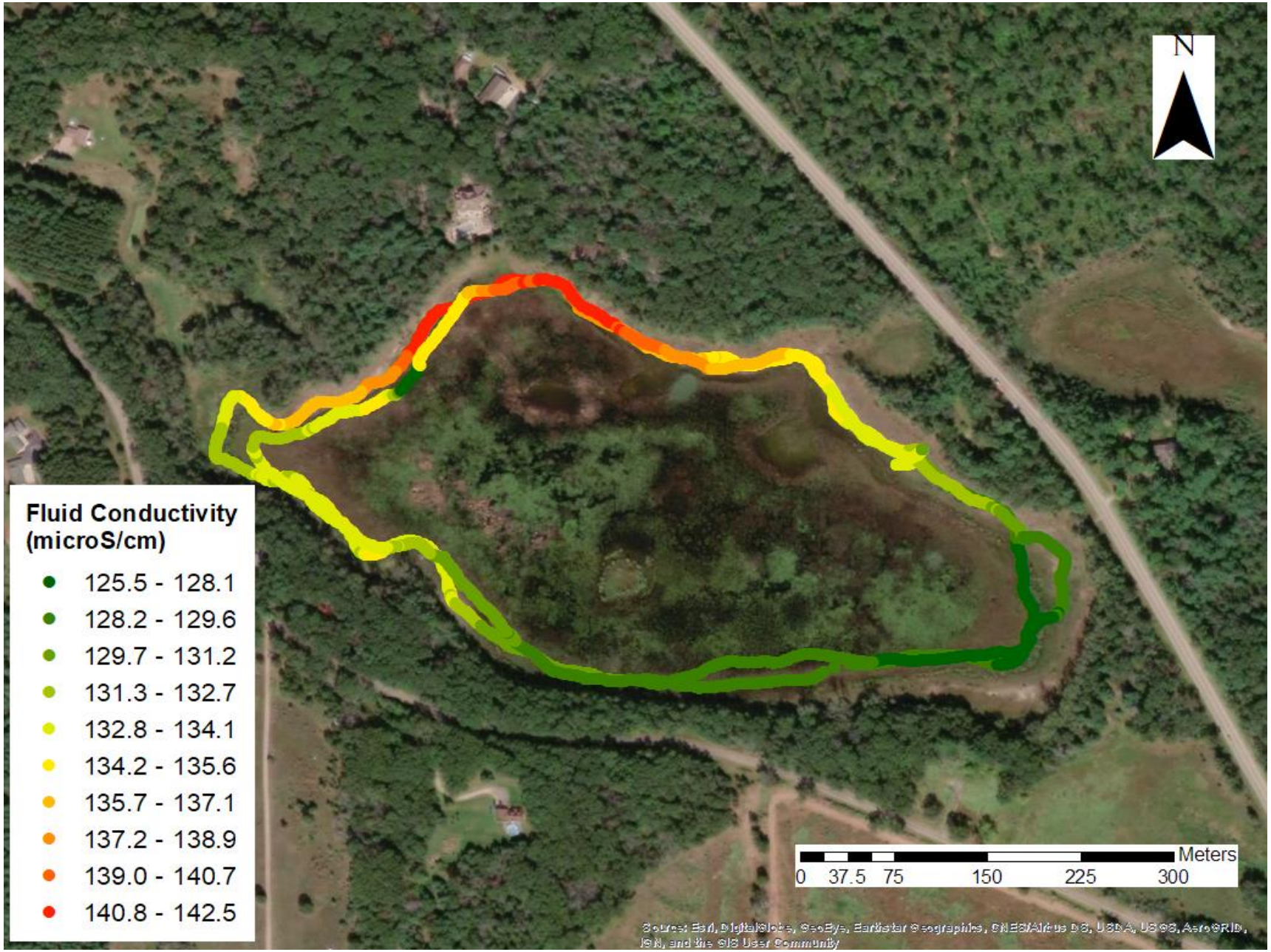


Figure 4.4.7: Fluid conductivity data from Plainfield Lake (Christenson and Hart, personal communication).

5 Portable, Self-Contained Seismic Sensor Array

Seismic sensors are one of the most versatile and widely used instruments for geophysical applications. One of the most common applications is to use an array of seismic sensors to perform near-surface imaging of the subsurface. Another example in which seismic sensors are used is for vibration monitoring to ensure that vibration levels do not exceed acceptable values and cause structural damage. Seismic sensors can also be used to measure the ambient noise at a site and calculate the ratio of noise in the horizontal and vertical directions (HVSR), a technique which is used to determine the thickness of sediment layers or perform seismic zoning. In this section, we describe the design of a seismic sensor that can be used for any of these applications.

5.1 System Setup and Wiring

For this sensor, our goal was to create a device that was portable, easy to install and setup, self-contained, and could record 3D seismic signals using geophones in real time and locate them with GPS coordinates. To do this, we needed the following components: a GPS to provide real time and location, three geophones to detect seismic signals, an ADC to convert the raw, seismic signals to digital values, a microSD card to store data, and a microcontroller to control the system. Presented in this section is a discussion of how each of these components was selected.

5.1.1 Seismic Sensor

A digital accelerometer was first considered since it was simple to use and did not require converting an analog signal to a digital signal. However, testing of a digital accelerometer setup revealed that the resolution was not sufficient to capture vibration signals. To address this issue, we instead considered using an analog accelerometer (ADXL 337) or geophones in conjunction

with an ADC. Ultimately, we selected geophones because we already had a large quantity we could use and the quality of our geophones was much better than the quality of inexpensive 3D accelerometers.

For the geophones, we included three small circuits (similar to the circuit shown in Figure 5.1.5) containing three resistors that alter the voltage output by the geophones so it falls within the range that can be measured by the microcontroller. In addition, we added a first order RC filter to filter out signals above the Nyquist frequency of the sampling rate (125 Hz). An ADC is required to convert the signal output by the geophone to digital values that can be recorded by the microcontroller.

5.1.2 Analog to Digital Converter

Several different ADCs were considered, including the Arduino Uno microcontroller's built-in 10-bit ADC, the Teensy 3.6 microcontroller's built-in 13-bit ADC, the ADS1015 12-bit and ADS1115 16-bit ADC shields from Adafruit, and the ADS1220 24-bit ADC shield from Protocentral. When considering which ADC to use, both the resolution and sampling rate that could be achieved were considered. Since the sampling rate also depends on the processing speed of the microcontroller, the performance of the external ADCs when used with both the Arduino Uno microcontroller and the Teensy 3.6 microcontroller were considered. The goal was to have sufficient resolution to capture low amplitude vibrational signals while still maintaining a sufficiently fast sampling rate to capture seismic signals of interest.

ADC selection occurred in three phases. During the first phase, selection was mainly based on simplicity and ease of use, with the main goal of simply creating a system that was functional. Once this was achieved, further improvement could be done if necessary. During this phase, only the Arduino Uno microcontroller was used for simplicity, and we considered the

Arduino Uno microcontroller's built-in 10-bit ADC and the two ADC shields from Adafruit. Since we wanted to maximize resolution, we decided to try the ADS1115 16-bit ADC shield because it had much higher resolution than the Arduino Uno microcontroller itself and functioned identically to the ADS1015 12-bit ADC shield, so the higher resolution was preferred. However, testing of the microcontroller system using the ADS1115 16-bit ADC shield revealed some major problems with the ADC shield. The biggest problem was how unreliable the shield was. For some reason we have been unable to identify, some of the ADS1115 16-bit ADC shields caused the seismic sensors to stop working after a short period of time, after which they would no longer work even after being reset. Individual testing of the several different components used in the seismic sensors showed that whatever the problem was, the source was the ADS1115 16-bit ADC shield. In addition, some of the seismic sensors that appeared to work recorded very poor-quality data that was unusable. Another more minor issue that occurred was that reducing the conversion delay to increase the sampling rate caused some of the values to be recorded as the wrong axis. While this was not as large of a concern as the reliability issue, it still required addressing. Examples of some of the data we recorded are shown in Figure 5.1.1.

However, once all the bad data were filtered out, the results looked promising as shown in Figure 5.1.2. This figure shows the acceleration recorded by the y-axis accelerometer from Sensor 1 (Figure 5.1.1) as a compactor drives back and forth on the road nearby. The data clearly shows an increase in acceleration as the compactor nears the sensor and a decrease as it moves away. This suggested that we could capture seismic signals with the resolution of ADCs coupled with a microcontroller, however, our microcontroller system needed improvement. This began the second phase of ADC selection. During this phase, our goal had shifted to maximizing the resolution and sampling rates we could reasonably achieve.

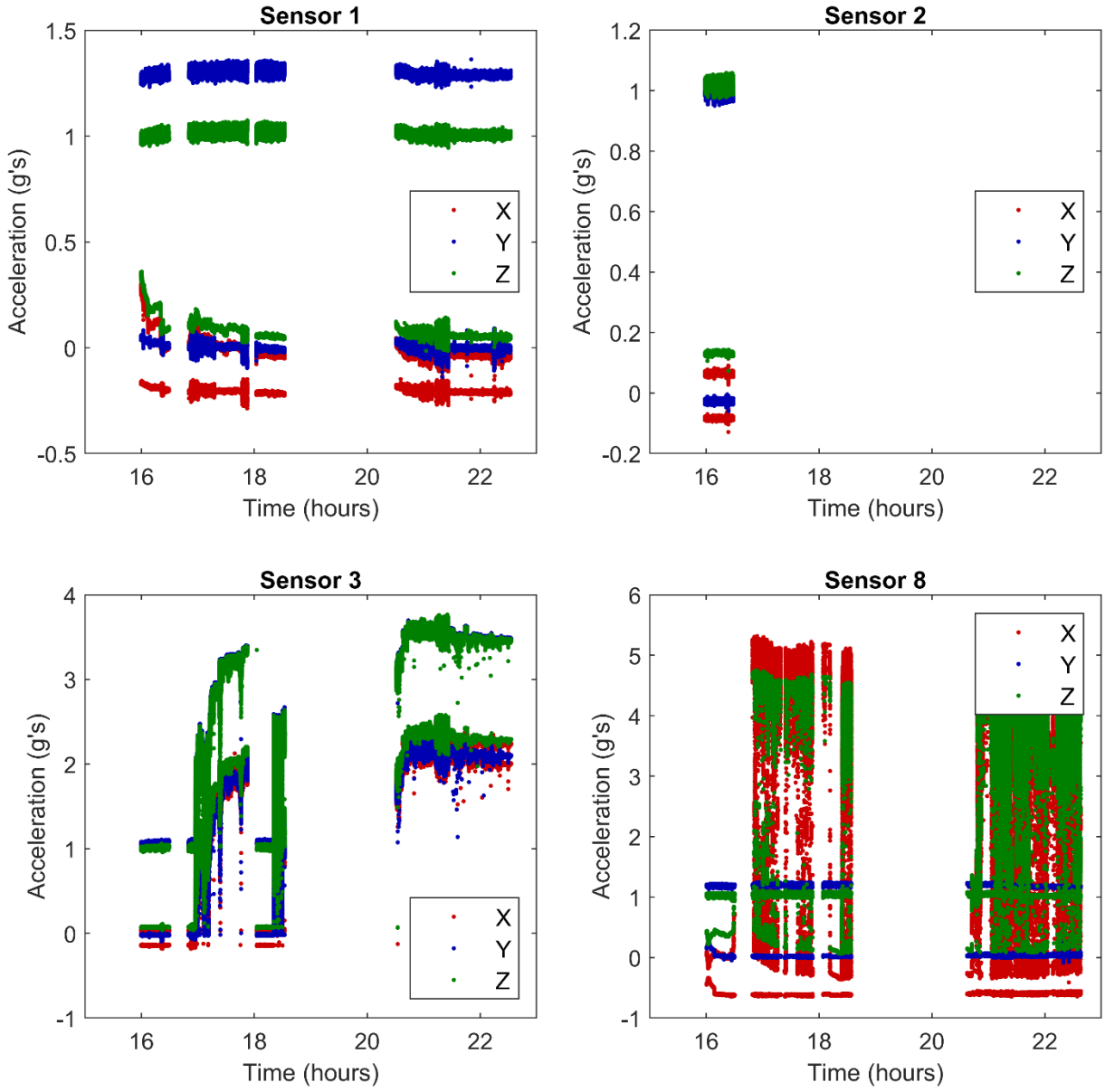


Figure 5.1.1: Quality of raw data recorded with preliminary microcontroller system: horizontal (x and y axes) and vertical (z axis) acceleration recorded by different sensors (Sensor 1 – usable data, Sensor 2 – microcontroller system stopped working, Sensor 3 – some poor quality data and some usable data, Sensor 8 – unusable data). Note, in all cases there are values that are recorded as the wrong axis.

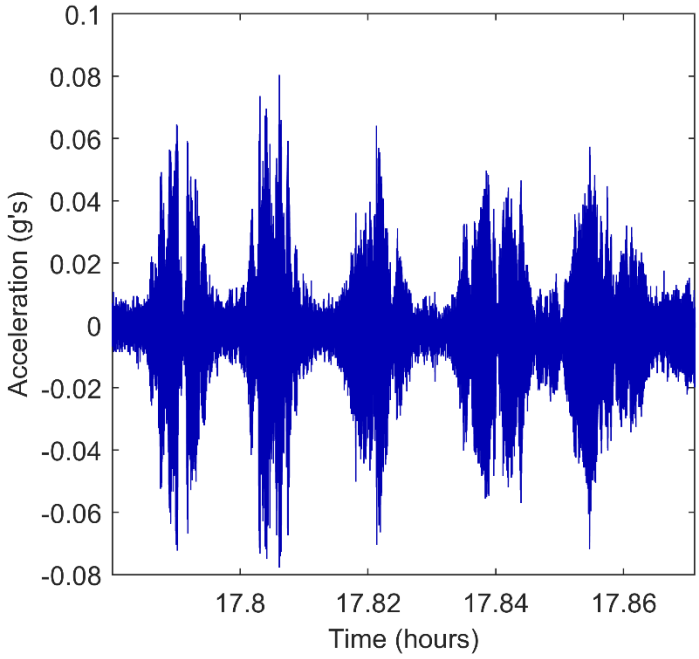
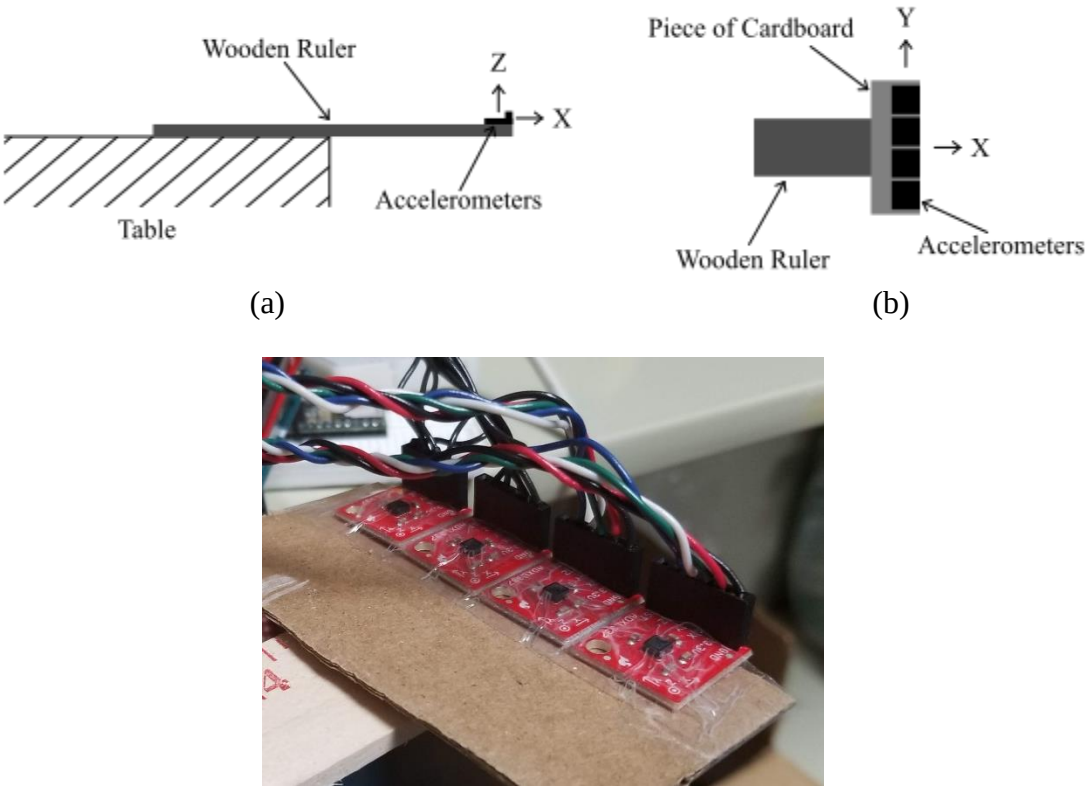


Figure 5.1.2: Quality of usable data recorded with preliminary microcontroller system: acceleration as compactor drives back and forth. The acceleration visibly increases as the compactor passes near the sensor.

In the second phase, we began by comparing the average sampling interval between data points and the quality of the recorded signal for several different combinations of microcontroller and ADC shields. To test this, accelerometers were attached to the end of a ruler acting as a cantilever beam. The end of the ruler was displaced and released to generate a signal that could be recorded and compared. Figure 5.1.3 shows the orientation and attachment of the accelerometers for this setup. A summary of the combinations of microcontroller and ADC shields that were compared and their results are shown in Table 5.1.1 and Figure 5.1.4. Based on these results, Adafruit's ADS1115 16-bit ADC shield is preferable to Protocentral's ADS1220 24-bit ADC shield because of its higher sampling rate. Since 16-bits of resolution should be sufficient for our purposes, the extra resolution of the ADS1220 24-bit ADC shield is not worth the lower sampling rate. In addition, all combinations containing the Teensy 3.6 microcontroller



(a) (b)

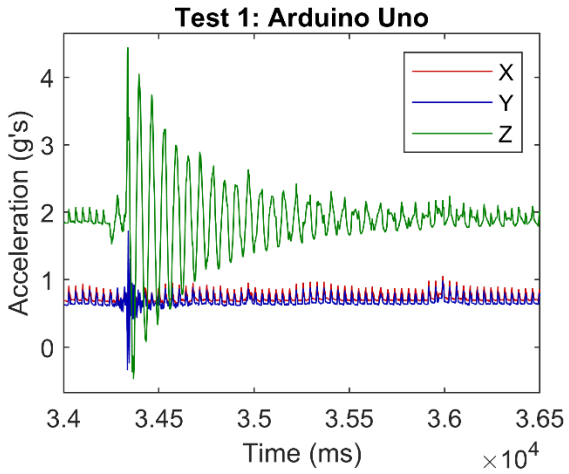
(c)

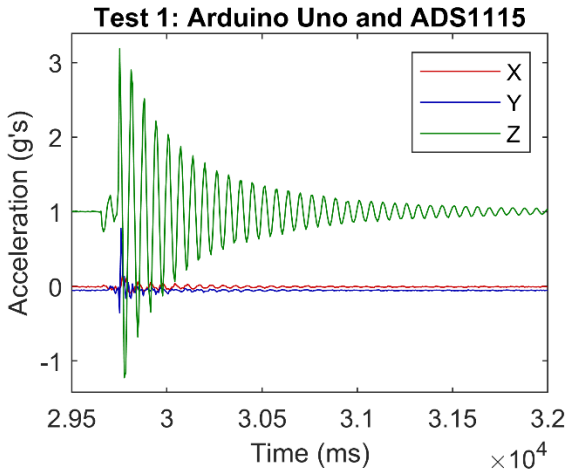
Figure 5.1.3: Testing setup used to compare performance of different microcontroller and ADC shield combinations (test described in Table 5.1.1): (a) side view, (b) top view, and (c) image of attachment of accelerometers.

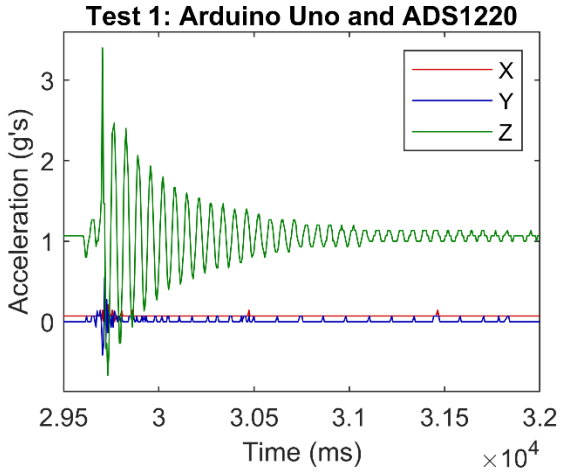
performed better than the corresponding combinations with the Arduino Uno microcontroller, likely due to the Teensy 3.6 microcontroller's higher processing speed; therefore, all combinations including the Arduino Uno microcontroller were eliminated. Using storage arrays greatly reduced the irregularities in sampling interval, therefore, using the Teensy 3.6 microcontroller or the Teensy 3.6 microcontroller and Adafruit's ADS1115 16-bit ADC shield with storage arrays were selected as the best combinations. To determine which of these two options was better, further testing was performed to evaluate if the Teensy 3.6 microcontroller's 13-bit resolution would be sufficient.

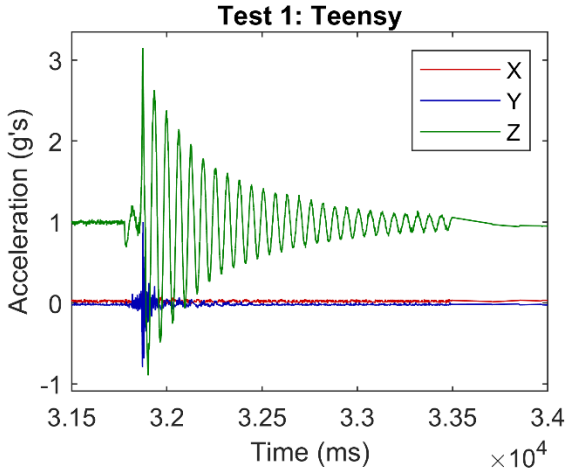
Table 5.1.1: Comparison of performance of different microcontroller and ADC shield combinations.

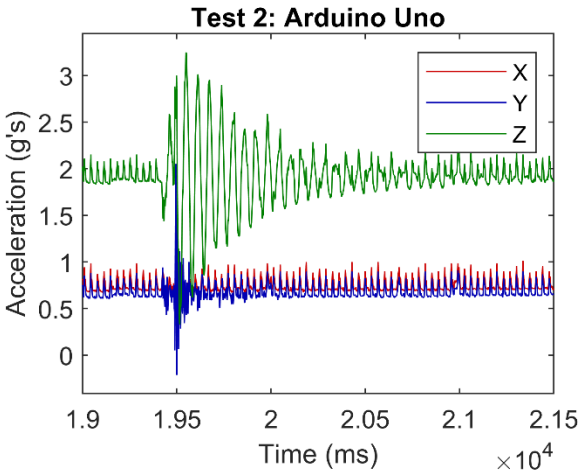
Test	Microcontroller	ADC	Resolution	Average Sampling Interval (ms)	Standard Deviation of Sampling Interval (ms)	File Size (KB)	Notes
1	Arduino Uno	None	10-bit	1.3	0.98	822	Using millis(), no storage array
	Arduino Uno	ADS1115	16-bit	7.2	1.05	202	Using millis(), no storage array
	Arduino Uno	ADS1220	24-bit	5.9	1.08	248	Using millis(), only 2 decimal points recorded, no storage array
	Teensy 3.6	None	13-bit	0.1	0.98	11,355	Using millis(), multiple data points with same time, no storage array
2	Arduino Uno	None	10-bit	1.5	0.84	862	Using micros(), no storage array
	Arduino Uno	ADS1115	16-bit	7.4	0.95	222	Using micros(), no storage array
	Arduino Uno	ADS1220	24-bit	9.5	1.28	322	Using micros(), 10 decimal points recorded, no storage array
	Teensy 3.6	None	13-bit	0.1	1.00	12,194	Using micros(), no storage array
3	Teensy 3.6	None	13-bit	0.1	1.00	12,204	Using micros(), no storage array
	Teensy 3.6	ADS1115	16-bit	3.6	$7.18*10^{-4}$	451	Using micros(), with storage array and interrupt (to indicate data conversion is complete)
	Teensy 3.6	None	13-bit	2.6	$4.70*10^{-5}$	571	Using micros(), with storage array (record data for 60 sec, 2.5 ms sampling interval)
	Teensy 3.6	None	13-bit	0.05	$3.58*10^{-4}$	14,872	Using micros(), with storage array (record data for 1.25 sec, as fast as possible)

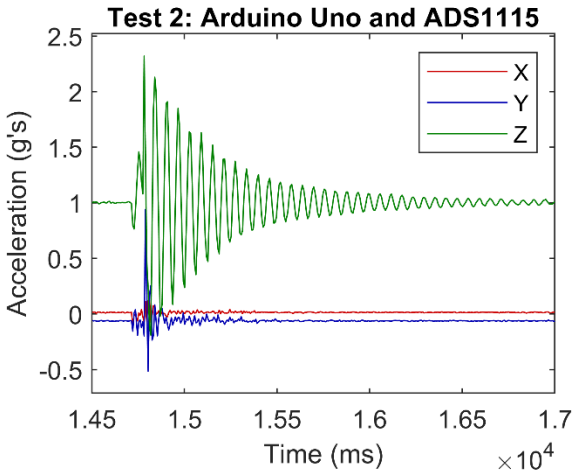

Test 1: Arduino Uno
Acceleration (g's)
Time (ms)
×10^4
X
Y
Z


Test 1: Arduino Uno and ADS1115
Acceleration (g's)
Time (ms)
×10^4
X
Y
Z


Test 1: Arduino Uno and ADS1220
Acceleration (g's)
Time (ms)
×10^4
X
Y
Z


Test 1: Teensy
Acceleration (g's)
Time (ms)
×10^4
X
Y
Z


Test 2: Arduino Uno
Acceleration (g's)
Time (ms)
×10^4
X
Y
Z


Test 2: Arduino Uno and ADS1115
Acceleration (g's)
Time (ms)
×10^4
X
Y
Z

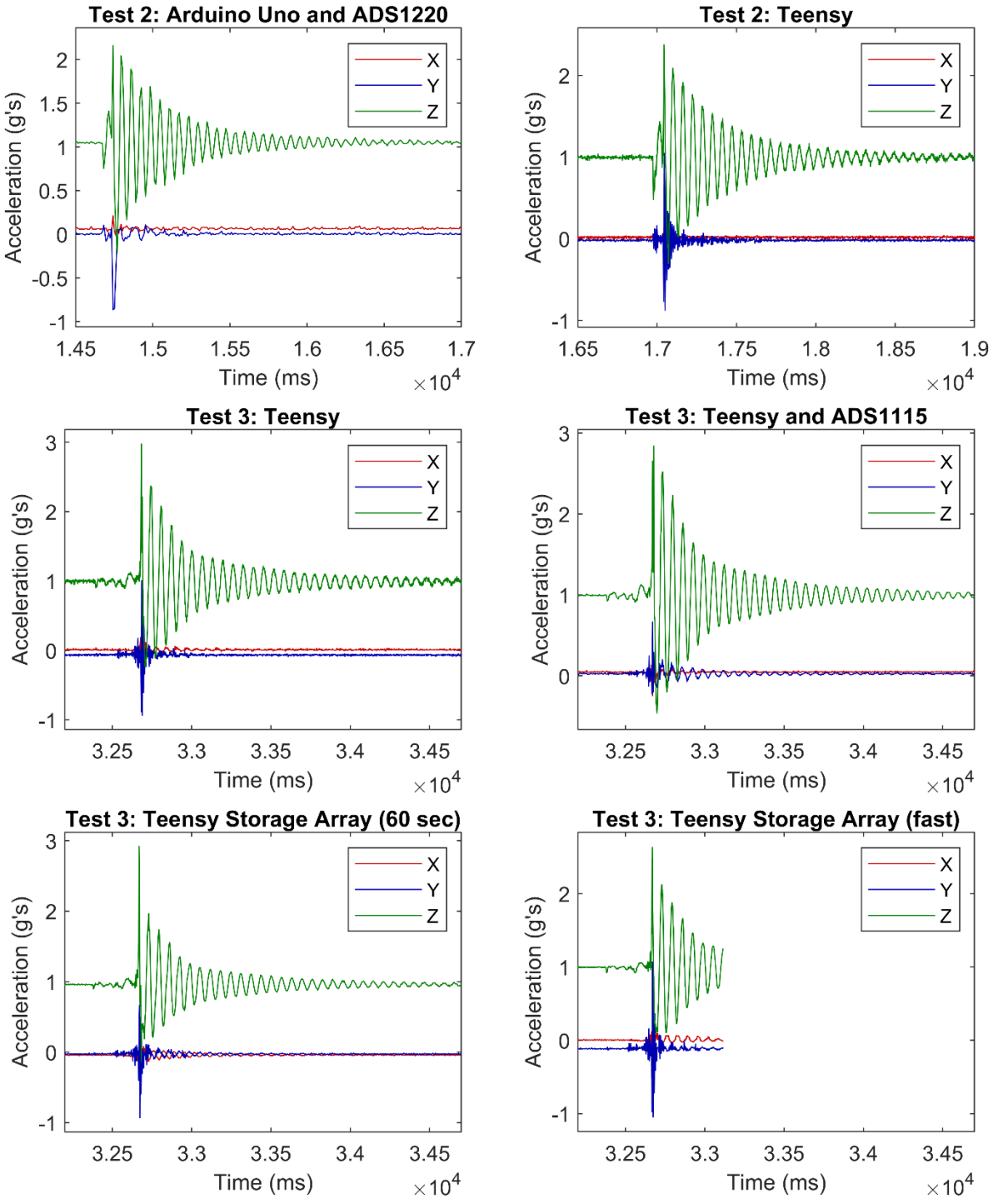


Figure 5.1.4: Comparison of performance of different microcontroller and ADC shield combinations for test conditions described in Table 5.1.1.

To determine if the Teensy 3.6 microcontroller's 13-bit resolution would be sufficient to capture low amplitude signals, we recorded the output signal of a geophone with both the Teensy 3.6 microcontroller and an oscilloscope to compare the results. To do this, a small circuit containing three different resistors and a battery was necessary to alter the voltage output range of the geophone so that only positive voltages are provided to the Teensy 3.6 microcontroller, which is incapable of measuring negative voltage. The circuit used to make this alteration is depicted in Figure 5.1.5. This circuit alters a voltage range of -3 to 3 V (which should fully include the range of the geophone's output during testing using low amplitude signals – e.g. knocking on table) to approximately 0 to 3 V, which lies completely in the voltage range the Teensy 3.6 microcontroller can measure. Using this circuit to alter the geophone's signal before it reached the Teensy 3.6 microcontroller, four different tests using different sources of vibrations were performed and the signals recorded with the oscilloscope and the Teensy 3.6 microcontroller were compared. Figure 5.1.6 shows a sketch of each of the testing setups depicting the sources of vibrations used. The details of these tests are described in Table 5.1.2 and the results are shown in Figure 5.1.7.

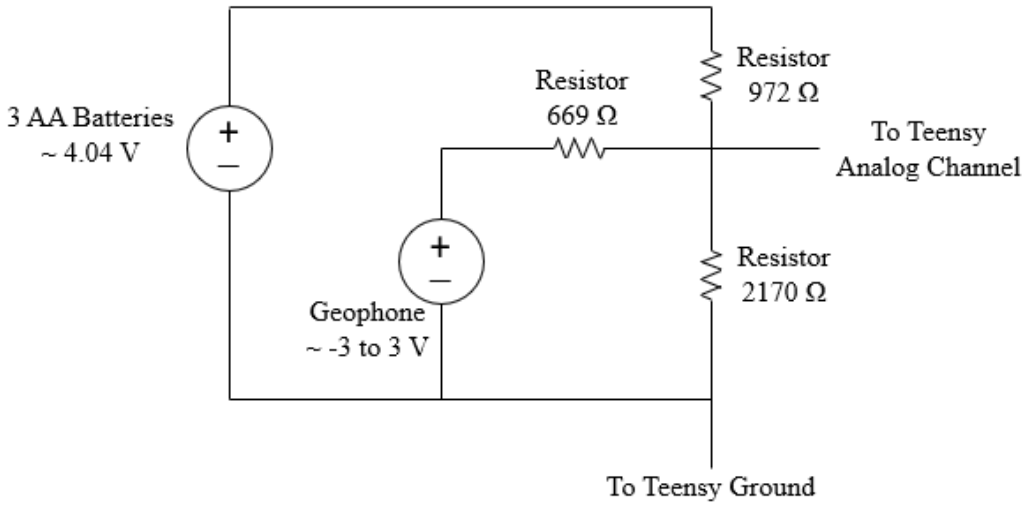


Figure 5.1.5: Circuit used to alter geophone output to only positive voltages.

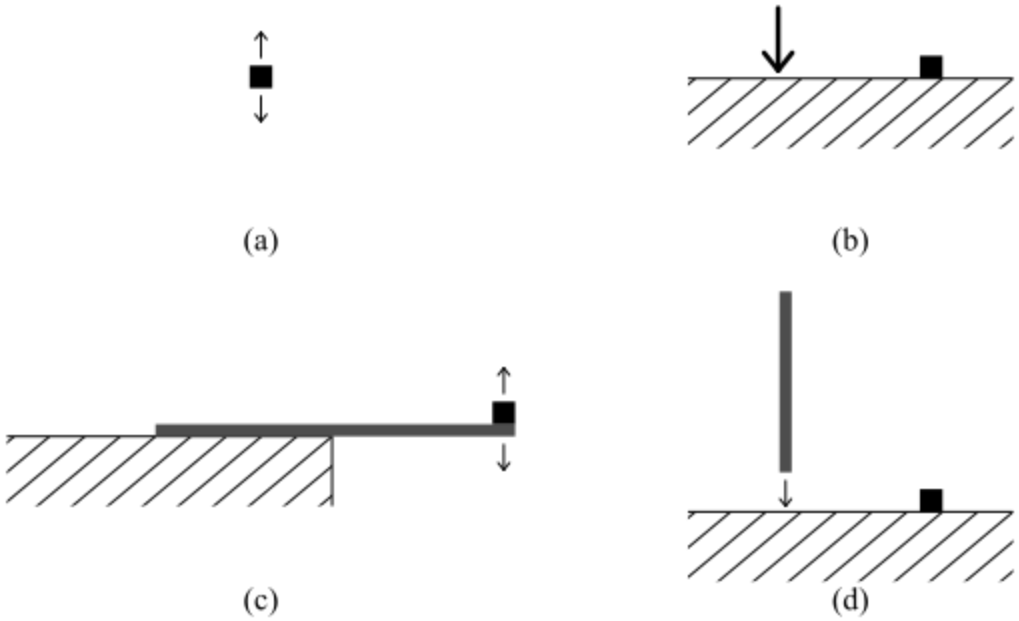


Figure 5.1.6: Testing setups used to evaluate resolution of Teensy 3.6 microcontroller: (a) Test 1 – geophone lightly shaken by hand, (b) Test 2 – knock on table with hand next to geophone, (c) Test 3 – geophone attached to end of wooden ruler that is displaced and released, and (d) Test 4 – end of wooden ruler dropped onto table next to geophone (geophone represented by black box, wooden ruler represented by gray rectangle).

Table 5.1.2: Comparison of geophone signal recorded with Oscilloscope and Teensy 3.6 microcontroller.

Test	Data Recording Device	Average Sampling Interval (ms)	Duration of Recorded Signal (sec)	Description
1	Oscilloscope	10	12	Lightly shaking geophone by hand
	Teensy 3.6	2.5	60	
2	Oscilloscope	0.2	0.24	Knock on table next to geophone
	Teensy 3.6	2.5	20	
3	Oscilloscope	0.2	0.24	Geophone attached to end of ruler that is displaced and allowed to oscillate
	Teensy 3.6	0.87	20	
4	Oscilloscope	0.2	0.24	End of ruler oriented vertically dropped onto table next to geophone
	Teensy 3.6	0.87	20	

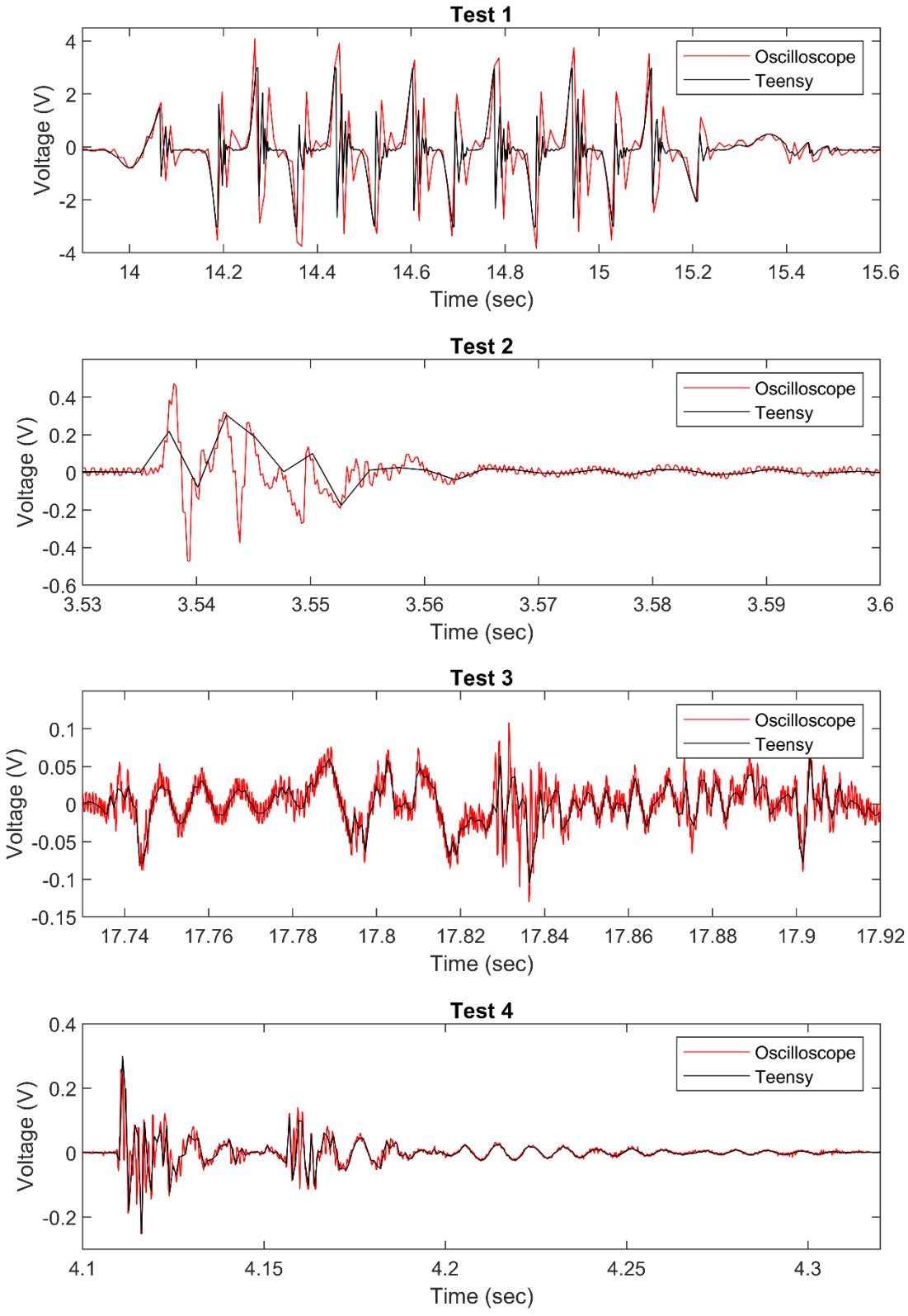


Figure 5.1.7: Comparison of geophone signals recorded with Oscilloscope and Teensy 3.6 microcontroller for testing setups shown in Figure 5.1.6.

In the first test, the data from the oscilloscope and Teensy 3.6 microcontroller show a good match, however, the oscilloscope's lower sampling rate may be causing some aliasing of the signal and the amplitude of the signal falls outside the -3 to 3 V range allowed by the circuit described above. This resulted in some of the peaks in the signal being cut off by the Teensy 3.6 microcontroller. Therefore, the following three tests used a higher sampling rate for the oscilloscope and lower amplitude signals that would not exceed the -3 to 3 V range. In addition, the duration of the recorded signal for both the oscilloscope and Teensy 3.6 microcontroller was decreased to remove much of the excess data recorded before and after the signal. In the second test, the oscilloscope's higher sampling rate allowed it to capture the signal quite well, however, the Teensy 3.6 microcontroller's sampling rate was insufficiently fast to capture the signal and significant aliasing occurred. Therefore, the Teensy 3.6 microcontroller's sampling rate was increased to the highest sampling rate it could accommodate in a storage array for a duration of 20 sec for the remaining two tests. Tests 3 and 4 both show a very good match between the oscilloscope and the Teensy 3.6 microcontroller. Due to the Teensy 3.6 microcontroller's lower sampling rate, much of the signal's high frequency noise is lost, but the lower frequency signal itself is captured well by the Teensy 3.6 microcontroller. This suggests that the Teensy 3.6 microcontroller's 13-bit resolution should be sufficient to capture low amplitude signals. In addition, the higher sampling rates the Teensy 3.6 microcontroller alone is able to achieve compared to the Teensy 3.6 microcontroller with an ADS1115 16-bit ADC shield is a large advantage that can allow us to modify and optimize the sampling rate and duration of recorded signals to be able to sufficiently capture signals of different frequencies. Therefore, we initially chose the Teensy 3.6 microcontroller's built-in ADC for our microcontroller system.

However, once we had completed a well-functioning microcontroller system, we decided to limit the sampling rate of our microcontroller system to 250 Hz, as discussed later, to minimize the impact of gaps in the data as storage arrays are printed to the microSD card. In addition, while working on another project we discovered that the provided library for the ADS1220 24-bit ADC shield from Protocentral has many artificially built-in delays that were unnecessarily slowing down the maximum sampling rate we could achieve. When the library is not used and the unnecessary delays are removed, the ADS1220 24-bit ADC shield is capable of a maximum sampling rate of 618 Hz (average sample interval of 1.62 ms), which is well above our selected sampling rate. Therefore, our third phase of ADC selection began as we reconsidered whether to use the Teensy 3.6 microcontroller's built-in 13-bit ADC or Protocentral's ADS1220 24-bit ADC shield. The downside to using the ADS1220 24-bit ADC shield is that the variables used to store data must be 32-bit integers, instead of the 16-bit integers that are sufficient when using the Teensy 3.6 microcontroller's 13-bit ADC. As discussed later, this influences the maximum number of measurements that can be stored before data must be written to the microSD card and therefore the maximum length of data that can be recorded before a gap occurs as data are saved. However, using the ADS1220 24-bit ADC shield with a sampling rate of 250 Hz results in a maximum possible file length of 53.6 sec, which we determined to be sufficiently long. Therefore, we determined that the improvement in resolution to be worth the decrease in file length and ultimately selected the ADS1220 24-bit ADC shield from Protocentral for our microcontroller system.

5.1.3 GPS

In order to be able to record seismic signals in real time and record the location of the seismic signals, we needed to include a GPS in the microcontroller system. Initially, we

considered using SparkFun's GPS logger shield, however, due to issues with inaccuracy we switched to Adafruit's Ultimate GPS shield. We had found Adafruit's Ultimate GPS to be more reliable for our water quality microcontroller system, so we decided to use it for our seismic microcontroller system as well and no longer had inaccuracy issues. Instead of using the backup battery coin slot that can be attached to the back of the Ultimate GPS shield, we decided to use the Teensy 3.6 microcontroller as the power supply for the backup battery. This will allow the Ultimate GPS shield to maintain any settings that were altered at the beginning of the code when turned off but does not require an additional battery. Using the Teensy 3.6 microcontroller as the power supply for the circuit used to alter the geophone output, we were then able to only require one external battery source used to power the entire microcontroller system.

5.1.4 Microcontroller

Both the Arduino Uno microcontroller and the Teensy 3.6 microcontroller were considered for this microcontroller system. The Arduino Uno microcontroller was initially considered because of its ease of use and simplicity; however, its slower processing speed was a large disadvantage and ultimately was limiting the sampling rates we could achieve. Therefore, the Teensy 3.6 microcontroller was ultimately chosen because of its much higher processing speed. The large improvement in sampling rates that are possible with the Teensy 3.6 microcontroller's higher processing speed greatly outweighed the relatively small reduction in simplicity and ease of use associated with using a more advanced microcontroller, especially since the Teensy 3.6 microcontroller can be programmed almost identically to the Arduino Uno microcontroller using the Teensyduino add-on to the Arduino software. In addition, the Teensy 3.6 microcontroller comes with a built-in microSD card slot allowing us to record the data without the added complexity that comes from hooking up and communicating with an external

microSD card slot. A schematic of the components included in the microcontroller system and the wiring connections between them is shown in Figure 5.1.8.

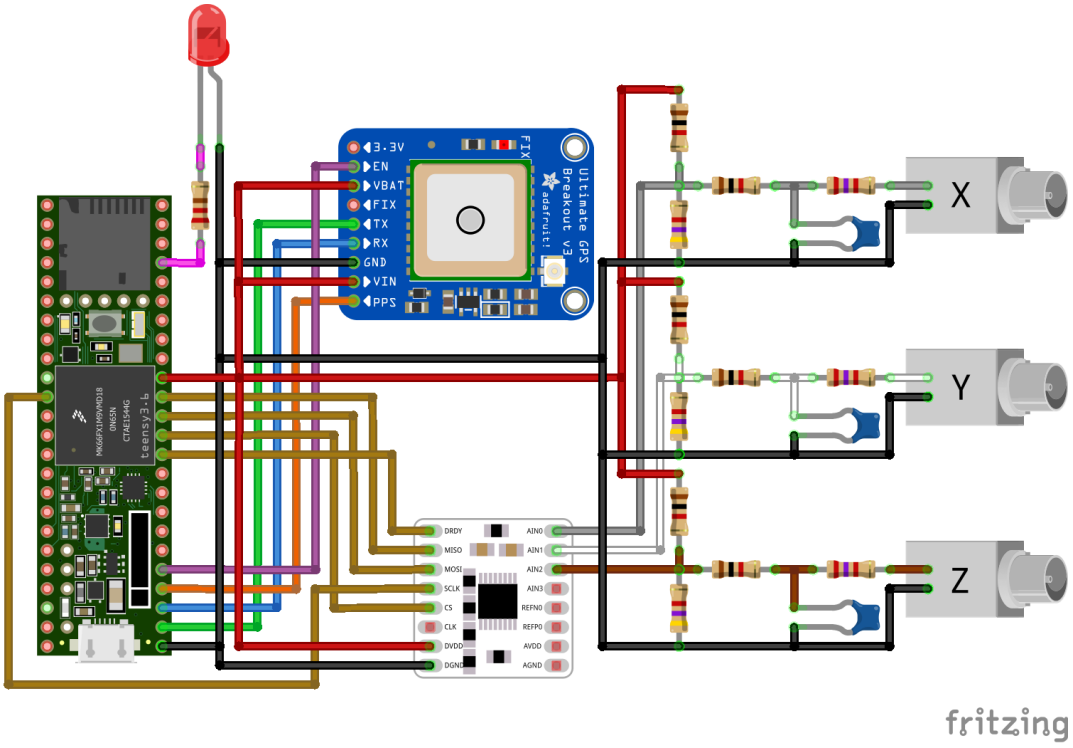


Figure 5.1.8: Diagram of wiring connections for seismic sensor (made with Fritzing using parts from Adafruit, 2019, Skymoo, 2019, and other unknown sources).

5.2 Code Design

There are several different functions that the code must be able to perform. First, since the GPS has a maximum update rate of 10 Hz, which is far too slow of a sampling rate to capture seismic signals, the code must be able to record data from the Ultimate GPS shield and the Teensy 3.6 microcontroller's internal reference time in order to establish a relationship between the two. This relationship will allow seismic data to be recorded using the Teensy 3.6 microcontroller's internal reference time (which updates approximately every microsecond) and correlated to real time using the trend between GPS time and the Teensy 3.6 microcontroller's

internal time. Next, the code must be able to record seismic data from the geophones at a high sampling rate. Finally, the code needs to incorporate some method that will provide an indication that the microcontroller system is working.

5.2.1 GPS Data and Timing

In order for seismic signals recorded by different microcontroller systems to be compared, each measurement in the recorded seismic signal needed to be accurately correlated to real time. In addition, for seismic signals to be meaningfully compared between microcontroller systems, our goal was to achieve timing accuracy with a precision of +/- 1 ms.

Initially, the GPS time was simply recorded at the start of each measurement, and the delay between when the GPS updated the time and when the microcontroller system recorded the time was neglected. However, this resulted in very poor timing accuracy and required much improvement. To improve our timing accuracy, we began to use the GPS's pulse per second (PPS) signal to accurately pair the GPS time to the microcontroller's internal reference time. Every time the GPS updates the time it sends a PPS signal; by using an interrupt to record the microcontroller's internal reference time this occurs at, much more accurate timing can be achieved.

In addition, to be able to accurately determine when GPS data were received from the GPS shield, we were unable to use an existing library to communicate with the GPS shield and parse the data. When using an existing library, you can very easily pull the most recent data from the GPS shield, but there is no simple way to accurately determine when the new data were first detected and therefore you cannot accurately pair the GPS data to the correct PPS signal. To get very precise timing, we used an interrupt to record the time when the PPS was detected, then also recorded the time that new GPS data were received. Together, both of these times can be used to

match the GPS data to the PPS that occurred shortly before the GPS data were received (we found the delay between the two typically falls between 175 to 700 ms).

Since we were not using an existing library to parse the GPS data, the code needed to include a function that could read the raw NMEA strings output by the GPS shield and parse them into separate variables for each of the parameters of interest. In addition, several indicators needed to be checked to verify that the NMEA strings contained valid data. We were interested in two NMEA strings in particular: the RMC string which contains date and time information, and the GGA string which contains location information. These two strings follow the format outlined in Table 5.2.1 which provides an example for the two sentences and highlights the parameters pulled from each sentence. While both sentences include the time, latitude, and longitude information, in order to get the date, altitude, and number of satellites, both sentences are used. Since you only need to pull the time, latitude, and longitude from one sentence, we chose to use the time from the RMC sentence and the latitude and longitude from the GGA sentence, so that the RMC sentence was used for all timing data and the GGA sentence was used for all location data. However, both sentences contain the exact same data for these shared parameters, so either one can be used interchangeably.

In order to parse the data, the code uses the locations of the commas to identify which characters correspond to which parameters. For example, in the RMC sentence, the time will always fall between the first and second commas. For this method to work, the code must first check which sentence is being parsed. This can be identified by checking the fourth through sixth characters which are always either RMC or GGA and identify the sentence format. In addition, once the sentence type is identified and the parameters from Table 5.2.1 are pulled from between the appropriate commas, the code checks parameters that indicate if the GPS data are valid

Table 5.2.1: NMEA sentence format example (taken from SparkFun Logger Shield Datasheet).

RMC		
$GPRMC,064951.000,A,2307.1256,N,12016.4438,E,0.03,165.48,260406,,,A*55		
Parameter	**Example**	**Description**
Date	260406	ddmmyy
Time	064951.000	hhmmss.sss
Status	A	A is valid, V is invalid (first A)
Mode	A	A is autonomous mode, D is differential mode, E is estimated mode, N is invalid (second A)
GGA		
$GPGGA,064951.000,2307.1256,N,12016.4438,E,1,8,0.95,39.9,M,17.8,M,,*65		
Parameter	**Example**	**Description**
Latitude	2307.1256	ddmm.mmmm
Latitude Direction	N	N or S
Longitude	12016.4438	dddmm.mmmm
Longitude Direction	E	E or W
Altitude	39.9	Altitude above/below mean sea level in meters
Number of Satellites	8	
Fix	1	0 is invalid, 1 is a GPS fix, 2 is a Differential GPS fix

before saving the data. For the RMC sentence this includes two parameters, the status and mode. The code checks that the status is A and the mode is either A, D, or E which are all considered to be valid. For the GGA sentence the code checks that the fix is either 1 or 2, which are both valid.

Because the update rate for the GPS data is relatively slow, a trend between the GPS time and the Teensy 3.6 microcontroller's internal reference time is used to allow seismic data to be recorded at a much faster rate. This must be done carefully to ensure the desired 1 ms precision is maintained. While the Teensy 3.6 microcontroller's internal reference time is fairly accurate and the trend between the two times fits extremely well to a linear trend, there are small variations and fluctuations in the Teensy 3.6 microcontroller's internal reference time that could result in a loss of precision if neglected, especially over long periods of time.

To determine how frequently the trend needed to be reestablished, we recorded the GPS time and the Teensy 3.6 microcontroller's internal reference time for several hours to evaluate

how the trend varies over time. A linear relationship between the GPS times and the Teensy 3.6 microcontroller's internal reference times was determined using different amounts of data to evaluate the best method of relating the two times. Using these linear relationships, the real time was calculated using the Teensy 3.6 microcontroller's internal reference time and the difference between this calculated time and the GPS time was evaluated. Figure 5.2.1 shows how the difference between calculated real time and recorded GPS time varies when the following GPS data are used to calculate the linear relationship: 2 min of data at the start only, 4 min of data at the start only, and 2 min of data every hour (shown for the first two hours). This figure shows that when the trend is only calculated at the beginning, the timing will exceed an error of 1 ms before the first hour ends. In contrast, when the trend is established using data from the beginning and end of each hour, the timing maintains an accuracy of 1 ms for the duration of the entire hour.

Therefore, to adjust the trend over time to very accurately define the relationship between the two times, we decided to collect approximately 2.5 minutes of timing data (GPS time and Teensy 3.6 microcontroller's internal reference time) every hour. To define the trend relating the Teensy 3.6 microcontroller's internal reference time to real time, the 2.5 minutes of data collected before and after the hour interval are used to establish a linear relationship between the Teensy 3.6 microcontroller's internal reference time and the GPS time. The linear relationship is then used to convert the Teensy 3.6 microcontroller's internal reference time recorded along with the seismic data during the hour intervals to real time. This method was verified using three different microcontroller systems containing a Teensy 3.6 microcontroller and an Ultimate GPS shield to separately record the same sinusoidal signal output by a signal generator. The results showed that the desired 1 ms precision was met as shown in Figure 5.2.2.

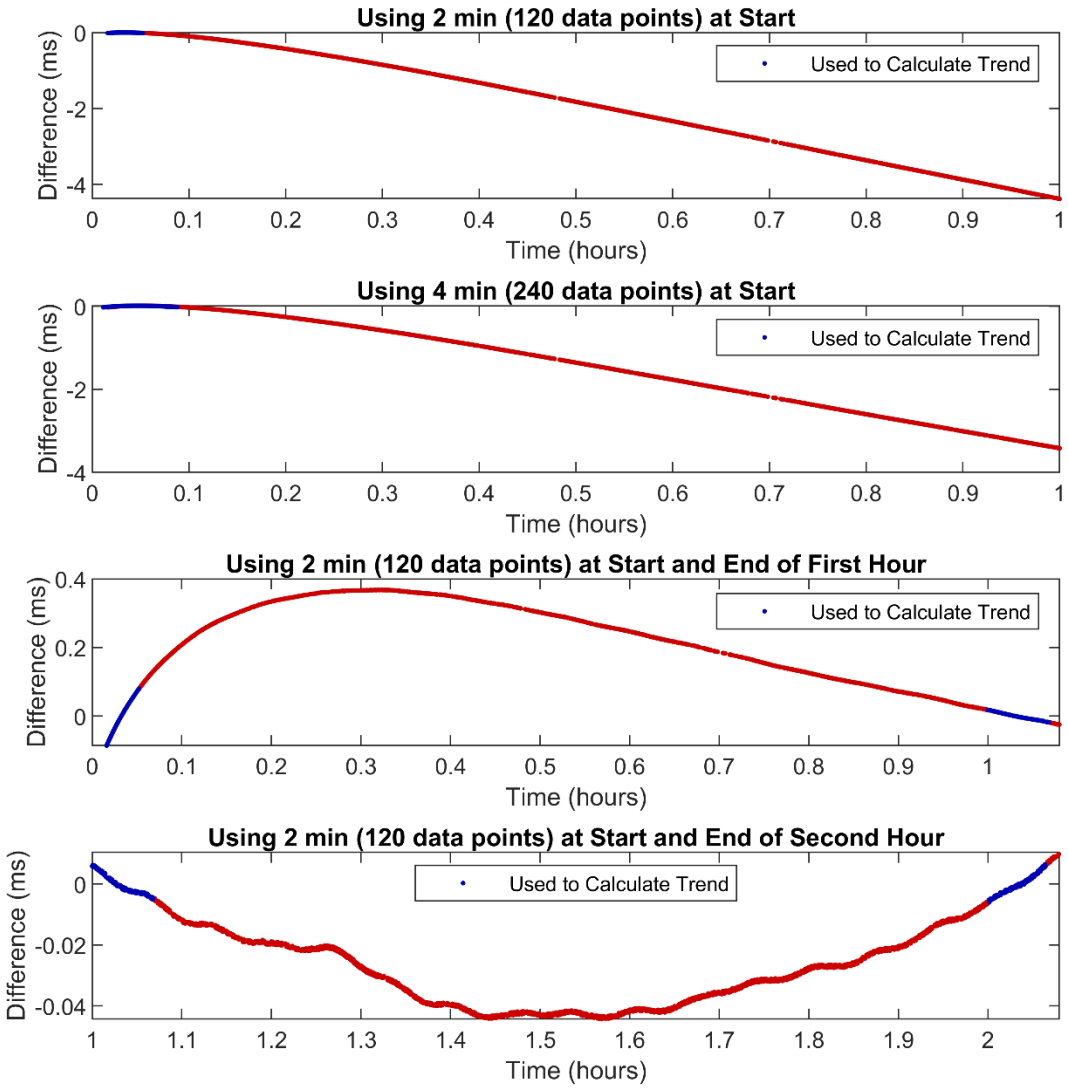


Figure 5.2.1: Quality of linear relationship between Teensy 3.6 microcontroller's internal reference time and GPS time for different amounts of data used to calculate trend.

The GPS location tends to fluctuate, therefore, to get a more precise location, GPS location data are also recorded each time the GPS time is recorded. Since the microcontroller system is not expected to be moved during data collection, all the GPS location data can be averaged to get a more accurate representation of the location.

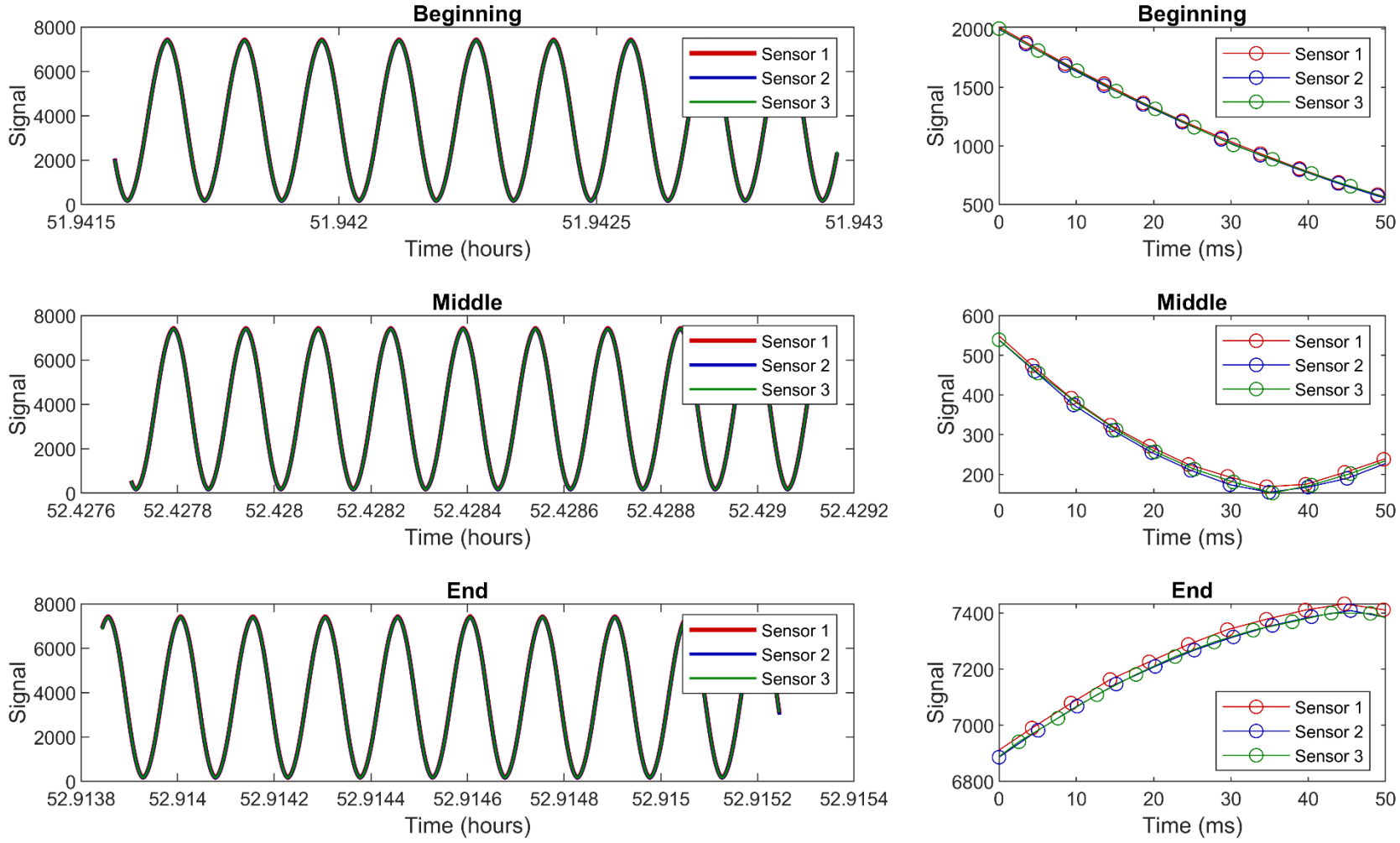


Figure 5.2.2: Results showing the timing precision achieved when three separate microcontroller systems (Teensy 3.6 microcontroller plus Ultimate GPS shield) were used to record the same sinusoidal signal. This is the timing accuracy when 2 min of GPS data before and after each hour of data collection is used to convert the signals recorded during that hour to real time. The figure shows how well the signals recorded by the three microcontroller systems align at the beginning, middle, and end of one hour of data collection.

5.2.2 Recording Seismic Data

The biggest consideration when designing the portion of the code used for seismic data collection was maximizing the sampling rates the microcontroller system could achieve. In terms of code design, there were two factors that had an influence on the sampling rate that needed to be considered: how frequently data should be printed to the microSD card and how often the file should be closed and the data on the microSD card saved. Both of these tasks (writing data to and opening and closing the file on the microSD card) require time to perform and therefore will interrupt data collection and either increase the sampling rate if performed after each data point is taken or create gaps in the data if performed at regularly spaced intervals.

Initially, seismic data were printed to the microSD card after each measurement, however, transition to the Teensy 3.6 microcontroller, which has greater memory, opened up the possibility of using storage arrays to temporarily store several measurements in storage variables, then print them all to the microSD card after a certain amount of time has passed. When storage arrays were not used, the Teensy 3.6 microcontroller was capable of recording data with an average time between measurements of 0.119 ms and a standard deviation of 0.995 ms. However, with storage arrays, the average time between measurements was 0.0532 ms and the standard deviation was reduced to $3.69 \cdot 10^{-4}$ ms. While the average time between measurements appears to double when storage arrays are not used, a closer look at the distribution of the data shows that the fastest sample interval of the Teensy 3.6 microcontroller without storage arrays is actually around 0.067 ms, which is only slightly higher than the average sample interval with storage arrays. This is depicted in Figure 5.2.3. The main improvement that comes from using storage arrays is in the reduction of the large variability in sample intervals, especially the elimination of the significantly longer sample intervals that occur periodically. Using storage

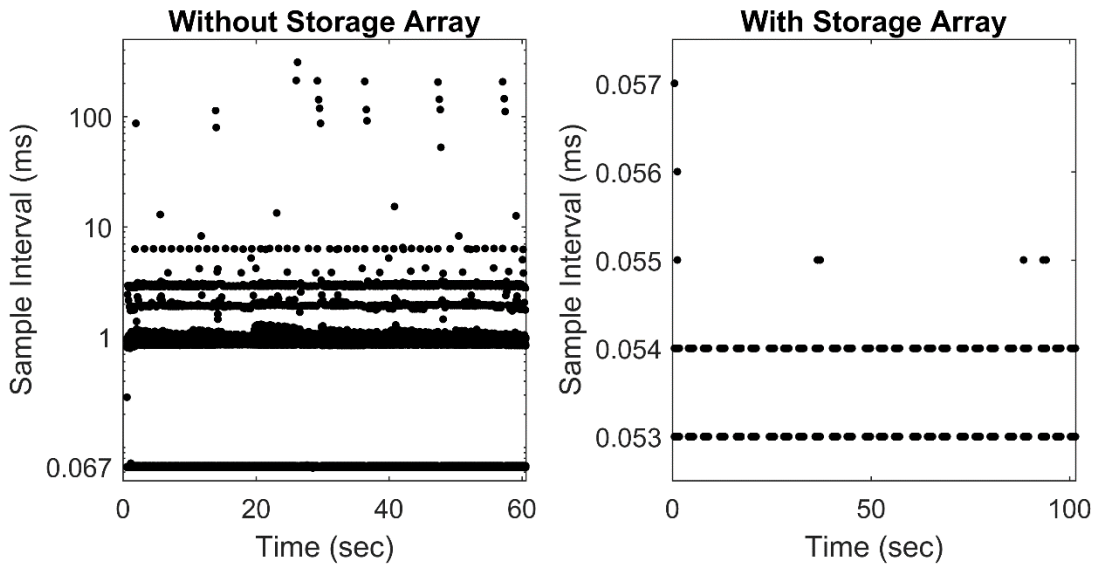


Figure 5.2.3: Comparison of the Teensy 3.6 microcontroller's sampling rate with and without the use of storage arrays. The sample interval is the time between consecutive measurements.

arrays results in a very consistent and reliable sampling rate that is just not possible without them, in addition to a slight improvement in the fastest possible sampling rate the microcontroller system can achieve. While there are delays between measurements as data are printed to the microSD card once the storage array is filled, these regular, controlled delays, though longer, are preferable to the random, periodic, and more frequent delays that occur when storage arrays are not used. The example provided above uses the Teensy 3.6 microcontroller's built-in ADC instead of the ADS1220 24-bit ADC shield, however, the same principle applies to the ADS1220 24-bit ADC shield and the benefits of using storage arrays remain the same.

The downside to using storage arrays is the limitations on the maximum amount of data that can be stored before it must be written to the microSD card. This affects the maximum length of the data files and the frequency of gaps in the data that occur while saving the files. When storage arrays are not used and the data are written to the microSD card after each measurement, there is no limit to the length of time that occurs before data files are closed and saved; the only consideration is how much data you are willing to lose, and therefore, how

frequently you want to ensure data are saved even if the microcontroller system stops functioning. When using storage arrays, however, the size of the microcontroller's memory determines how many data points can be saved. This means that there is a tradeoff between the sampling rate and how often data must be written to the microSD card. The Teensy 3.6 microcontroller has 256 KB of SRAM which is used for storing variables. While a large portion of SRAM can be used for storing seismic data, some memory must be left for other variables used in the code (such as GPS data or variables used to run functions or libraries) and it is wise to leave some SRAM as a buffer to ensure the microcontroller system never runs out of memory, which will cause it to crash. We decided to design the size of the storage arrays so that the code, when compiled, uses 90% of the available SRAM to store all the variables needed for the GPS data, seismic data, and to run the code and leaves 10% of the SRAM for minor variables used locally in functions or libraries, miscellaneous uses, and a bit of buffer to ensure the memory is not used up. When using the Teensy 3.6 microcontroller's 13-bit ADC, variables can be stored as 16-bit integers and it is possible to store 21,500 measurements. However, when using the ADS1220 24-bit ADC shield, 32-bit integers are required to store the data so only 13,400 measurements can be stored.

Because the storage arrays are set to contain a specific number of measurements, the maximum length of the files is related to the sampling rate through the following equation.

$$T = \Delta t * M \tag{5.2.1}$$

Where T is the length of the file in time, Δt is the sample interval (or average time between measurements), and M is the number of measurements that the storage arrays can store. This means that if using the maximum possible sampling rate of the Teensy 3.6 microcontroller, where measurements are taken approximately every 0.0532 ms, the maximum length of time that

data can be stored before it must be saved is only 1.14 seconds. However, since the average time it takes to save data containing 21,500 measurements is about 1.5 sec (with a full range of about 1.25 to 2.65 sec), this would mean that only 43.2% of the signal is being recorded. Therefore, to reduce the impact of gaps as data are saved, the sample interval was artificially increased to take a measurement every 4 ms (which corresponds to a sampling frequency of 250 Hz). When using the Teensy 3.6 microcontroller's 13-bit ADC, a sample interval of 4 ms used to save data containing 21,500 measurements results in a file length of 86 sec. This means that only 1.7% of the signal is lost in the gaps between files. While using the ADS1220 24-bit ADC shield, a sample interval of 4 ms used to save data containing 13,400 measurements results in a file length of 53.6 sec. For 13,400 measurements, it takes on average 1.4 sec to save data (full range of 1.18 to 2.62) so 2.6% of the signal is lost in the gaps between data. While the ADS1220 24-bit ADC shield results in a slightly higher percentage of lost signal, the percentage lost is still very reasonable and the increase in resolution is well worth it.

To ensure that seismic data can be paired to the correct GPS data needed to convert the Teensy 3.6 microcontroller's internal reference time to real time, the most recently received GPS data are printed at the top of every seismic data file. This means that during every hour interval of seismic data collection, the last set of GPS data printed to the preceding GPS file is recorded. Seismic data can then be matched to the GPS file that contains the matching GPS data as the last entry to determine which two GPS files to use for converting to real time.

5.2.3 LED Indicator

To provide a visual indication that the microcontroller system was working, an LED was incorporated into the microcontroller system. When deciding how to implement the LED, we wanted the LED to function in a way that would provide a continuous indication that the

microcontroller system was working. This means that the LED indication must occur frequently enough that the user does not have to worry about missing it and will not have long to wait for the next indication. In addition, because of early issues with reliability and the code freezing, we needed the LED to continuously alternate between on and off to act as an indication that the code is still running and continuing to collect data. Lastly, the LED must provide an indication of what stage the code is at: GPS or seismic data collection. To meet these requirements, the code blinks the LED as data are stored to storage arrays.

For the GPS data, the LED is switched on or off every time an RMC sentence is stored. This results in the LED alternating between on and off about once every second. Because both the RMC and GGA sentences are updated at the same time, if the LED state is switched every time any sentence is stored, the LED would switch two times sequentially very quickly (possibly too quickly to be visible) and the LED would remain almost entirely in one state. Therefore, to get the desired blinking effect, the code was designed to only switch the LED state if data from an RMC sentence was stored. The RMC sentence was chosen over the GGA sentence because the timing data pulled from the RMC sentence is far more important than the location data stored in the GGA sentence. Without the timing data, the seismic data recorded by the microcontroller system cannot be correlated to real time.

For the seismic data, the LED is blinked at a faster rate in order to provide a visual indication of whether GPS or seismic data are being stored. Since seismic data are recorded every 4 ms, simply switching the LED state after each measurement would cause the LED to blink faster than the eye could detect. Therefore, the LED state is instead switched every 50 measurements which results in the LED switching about every 200 ms. This rate is five times faster than the rate of blinking while GPS data are being stored, making it easily to distinguish

between the two rates and determine whether the microcontroller system is currently storing GPS or seismic data.

One disadvantage to this method is that the LED indicates that the microcontroller system is still running and that data are being stored, however, it does not necessarily indicate that data have been logged to the microSD card. However, because data storage must pause while data are being saved to the microSD card, there is a noticeable pause in the blinking of the LED as data are saved. Since data are stored relatively infrequently (every hour after 2.5 minutes of data collection for GPS data or approximately every minute for seismic data), this is not a continuous indication that data are being logged. The pause in LED blinking that occurs every minute is very difficult to miss and waiting for the next pause to check that data are being saved is not ideal. Therefore, we determined that the LED indications that the code was still running and data were being stored should be sufficient when combined with an indication that the microSD card was successfully initialized at the beginning of the code. Before the GPS shield is turned on and data collection begins, if there was an error with the initialization of the microSD card, the LED will be blinked five times. While there is still the potential for something to go wrong and data to not be logged, these measures, at the very least, mitigate that risk.

5.3 Compiling Data

The seismic sensors function by saving many different files containing GPS and seismic data that must be compiled and processed to convert the signals to real time. A MatLab code was developed to automate this process. This code looks at all the GPS and seismic data files contained in a given directory, uses the GPS files to develop trends relating the Teensy 3.6 microcontroller's internal reference time to real time, automatically matches the seismic files to

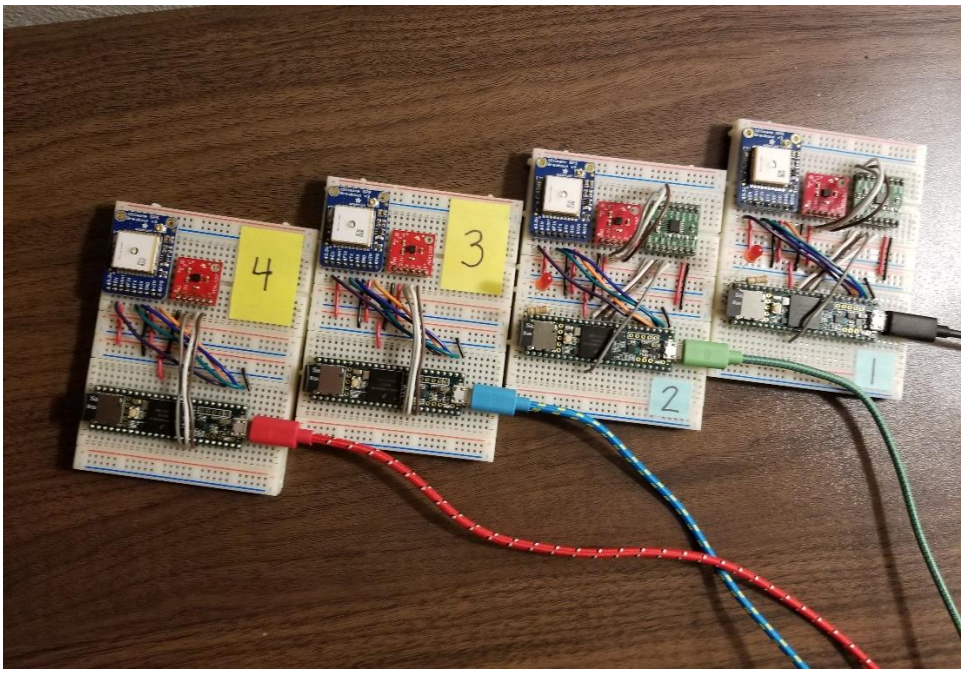
the appropriate trends, and converts and compiles the seismic data into real-time data. An annotated copy of this code can be found in the appendix.

5.4 Results

5.4.1 Timing Accuracy

Initial testing was performed to ensure that the microcontroller systems had sufficient timing accuracy. To test the timing accuracy, four microcontroller systems were used to record signals and the alignment of the signals was compared once they had been converted to real time. This test took place while development was still occurring, so the components used do not match the design of the final microcontroller system. At the time of this test, we were still using analog accelerometers, so all four microcontroller systems used a 3D analog accelerometer in place of geophones. In addition, we were still evaluating whether to use the Teensy 3.6 microcontroller's built-in 13-bit ADC or Protocentral's ADS1220 24-bit ADC shield, so systems 1 and 2 used the ADS1220 24-bit ADC shield and systems 3 and 4 used the Teensy 3.6 microcontroller's 13-bit ADC. However, all these components only affect the quality of the signal recorded and have no impact on the timing accuracy of the microcontroller systems, so the results are still valid for the final microcontroller system.

The test was performed by arranging the four microcontroller systems adjacent to one another and using adapters to plug all four microcontroller systems into the wall outlet simultaneously. This allows all microcontroller systems to be powered for the full duration of the test without having to worry about monitoring battery levels. Pictures showing the testing setup are shown in Figure 5.4.1. All four microcontroller systems were plugged in and allowed to obtain GPS signal. Once all the microcontroller systems had finished collecting the first file of GPS data, a seismic signal was produced by knocking on the table the microcontroller systems



(a)

(b)

Figure 5.4.1: Indoor testing setup to test timing accuracy: (a) the four microcontroller systems tested (1 and 2 – Protocentral's ADS1220 24-bit ADC shield; 3 and 4 – Teensy 3.6 microcontroller's 13-bit ADC) and (b) microcontroller systems powered using wall outlet and adapters.

were resting on three times consecutively. This was done at the beginning, middle, and end of the first hour of seismic data collection to evaluate how the timing accuracy varies between files of GPS data. At the end of the first hour of seismic data collection the microcontroller systems all collected another file of GPS data, which was used along with the first file to relate the signals recorded between them to real time. The results showing the alignment of the signals at

the beginning, middle, and end of the hour once the signals have been converted to real time are shown in Figure 5.4.2.

Figure 5.4.2 shows the signals from the z-axis of the 3D analog accelerometers. Since each microcontroller system uses its own accelerometer and the frequency of the signals is relatively high compared to the sampling frequency, we would not expect the signals to match exactly and are not concerned with the quality of the signals. In addition, because different ADC shields were used and the accelerometers were not calibrated, the signals had to be adjusted

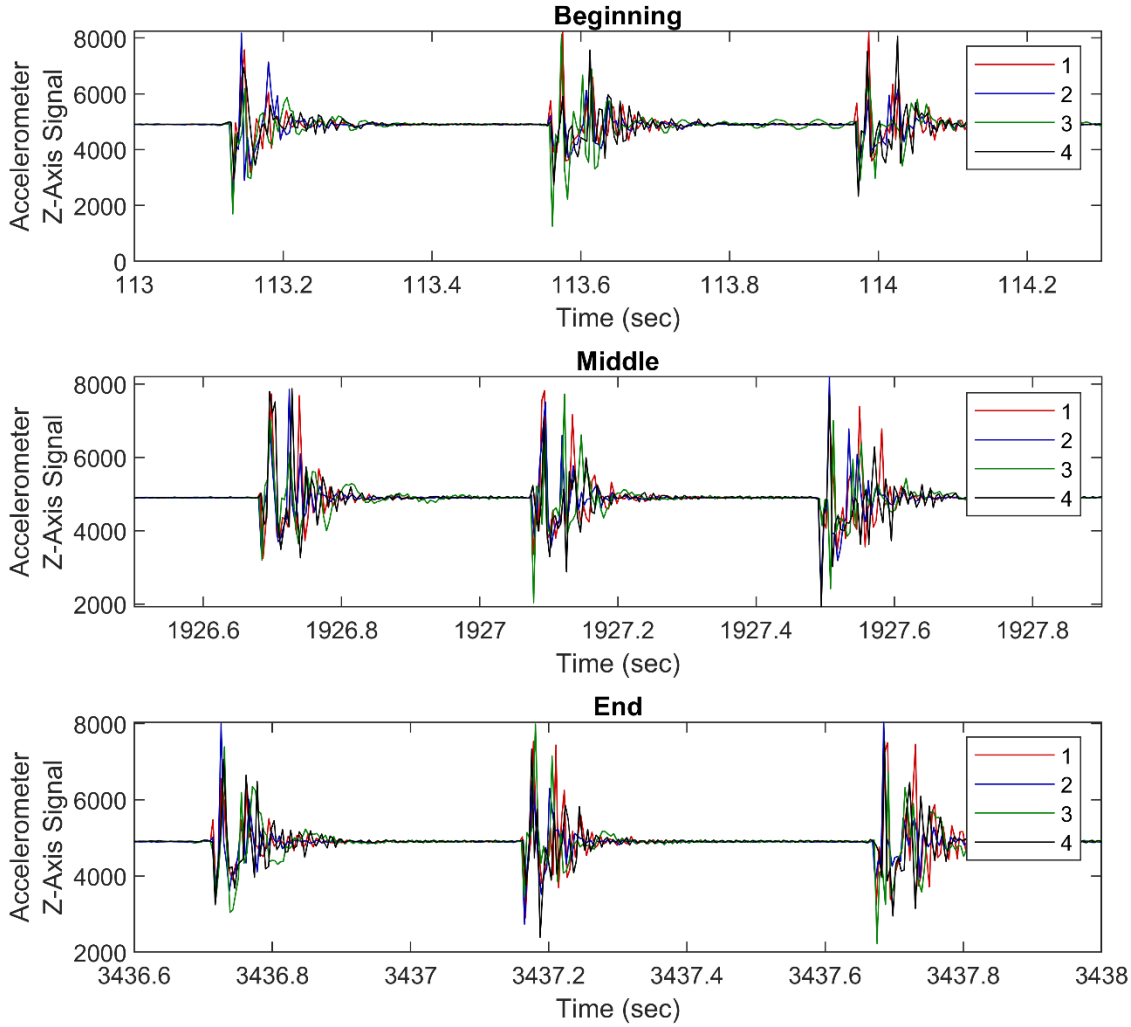


Figure 5.4.2: Results of timing accuracy test showing signals at the beginning, middle, and end of the first hour.

manually by dividing by a gain and adding an offset to get similar amplitudes and align the signals vertically. What is important is whether the signals have the same arrival time.

The results in Figure 5.4.2 show that all signals share the same arrival time regardless of when during the hour of data collection the signals were recorded. A close-up look of the arrival of the third knock during the middle of the hour is shown in Figure 5.4.3. This signal was selected because it has a very sharp arrival and the timing accuracy is expected to be the worst during the middle of the hour where the times differ the most from those recorded in the GPS files. In this figure, the data points corresponding to the arrival of the signal are shown as filled circles. These arrivals have a maximum difference of 0.0024 sec which is smaller than the sampling interval of 0.004 sec. Since the actual arrival time may occur between data points, it's possible that the timing resolution may be even greater.

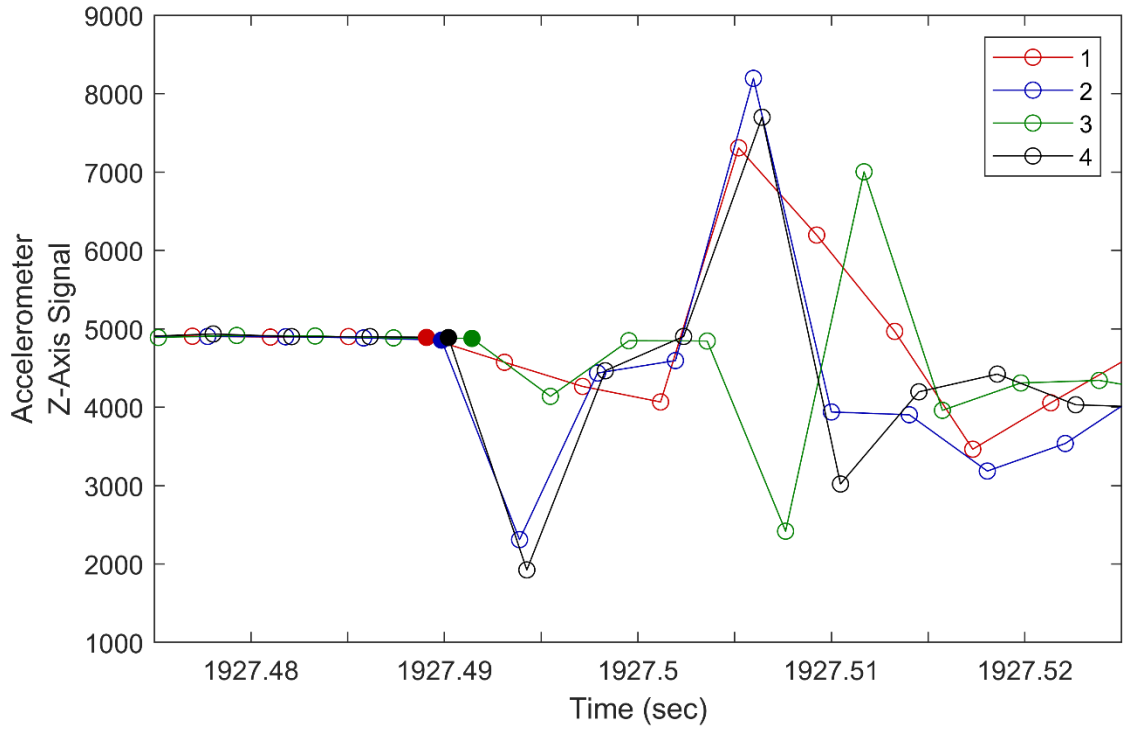
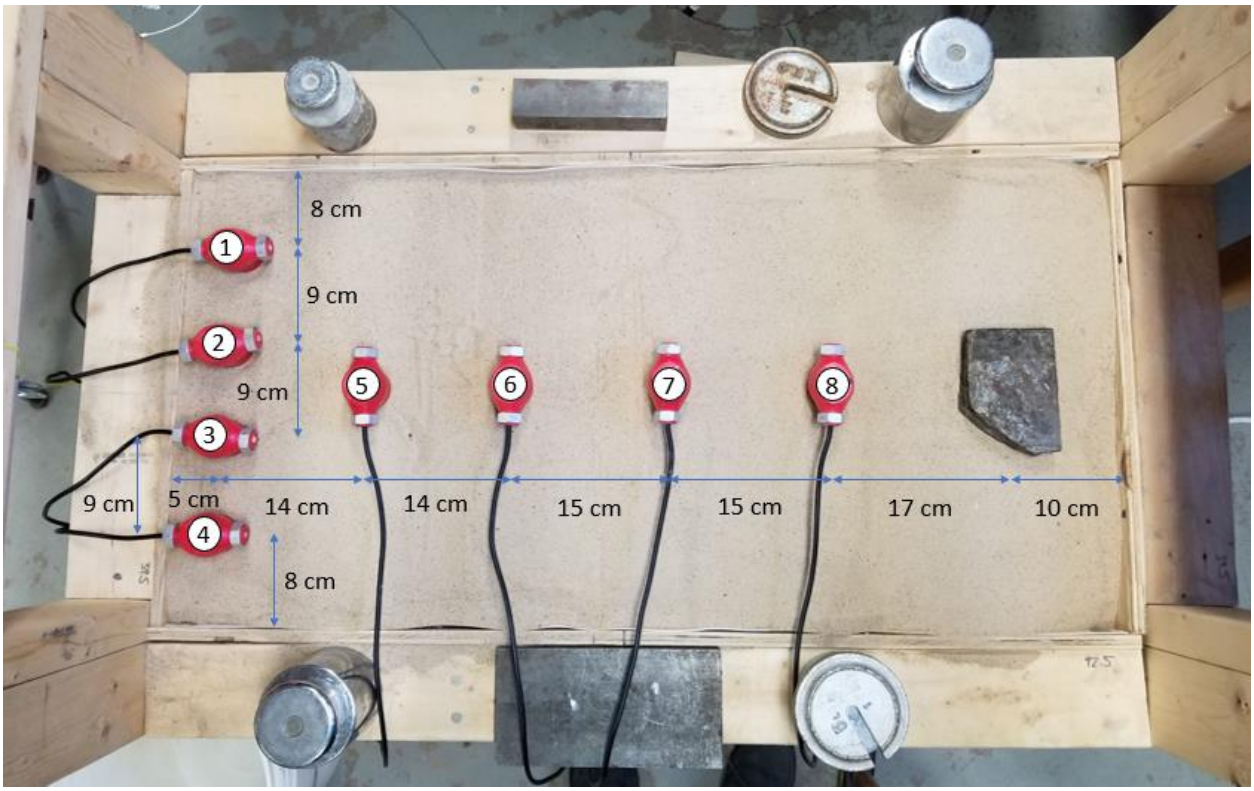


Figure 5.4.3: Close up of signal from third knock in the middle of the first hour of data collection.

5.4.2 Small-Scale Testing

A small-scale test was also performed using geophones to record seismic signals generated in dry sand to evaluate the timing and quality of the recorded signals in an environment that more closely resembles field conditions. This testing took place before RC filters were added to the microcontroller systems, however, adding RC filters should impact all the microcontroller systems equally so they should not impact the relative timing, only the quality of the signals. A rectangular, wooden box containing dry sand (constructed by Mike Zimmerman) was used to perform this test. The arrangement of the geophones and strike plate is shown in Figure 5.4.4. The geophones were arranged so that four geophones farthest from the strike plate were located the same distance from the strike plate (geophones 1 through 4). This should mean that the signals recorded by these geophones should be very similar and have approximately the same arrival time. These four geophones, therefore, can be used to evaluate the quality of the recorded signals and the timing accuracy. In addition, four more geophones were evenly spaced between the far line of geophones and the strike plate (geophones 5 through 8). We would expect that the arrival times should vary between these geophones such that the geophones nearest to the strike plate should have the earliest arrival times. In addition, we can monitor how the quality of the recorded signals varies with distance from the source of the signal. Our expectation is that all of the signals should share a similar shape, but the amplitude of the signals should decrease with distance from the strike plate.

In order for the microcontroller systems to obtain a GPS signal, the test began by taking the microcontroller systems outdoors, waiting for them to obtain a GPS signal, and allowing them to collect a file of GPS data. Once GPS data collection was complete for all microcontroller systems, they were brought indoors and hooked up to the geophones. An image showing the



(a)

(b)

Figure 5.4.4: Small scale testing setup: (a) arrangement of geophones and strike plate and (b) attachment of geophones to microcontroller systems.

microcontroller systems once they've been attached to the geophones is shown in Figure 5.4.4. Once all the microcontroller systems were hooked up and recording, a 1.36-kg (3-lb) metal hammer was used to hit the strike plate several times and generate seismic signals. Because data collection completed fairly quickly, the timing accuracy using only one file of GPS data should be sufficient for times close to the end of the GPS data. Therefore, once the microcontroller systems had saved the data from the strikes, the microcontroller systems were unplugged and a second GPS file at the end of an hour was not collected.

The results for one of the strikes are shown plotted in Figure 5.4.5. These signals are plotted against the time in seconds that has passed since the approximate beginning of the test. To determine the elapsed time, each of the signals was first converted to real time and a time determined to be the start of the test was then subtracted. This way, the magnitudes of the times on the x-axis are meaningful and the signals are still plotted so that the relative times between them are accurate. The results closely matched our expectations. As expected, the signals from the geophones closest to the strike plate had the largest amplitude. In addition, as the geophones get farther from the strike plate the signals show a later arrival time corresponding to the longer time required to traverse the greater distance. Finally, all the signals showed a similar shape and the signals from geophones 1 through 4 matched very closely, especially in the lower-frequency tail end of the signals. This suggests that the microcontroller systems are capable of recording seismic signals with sufficient quality and timing accuracy.

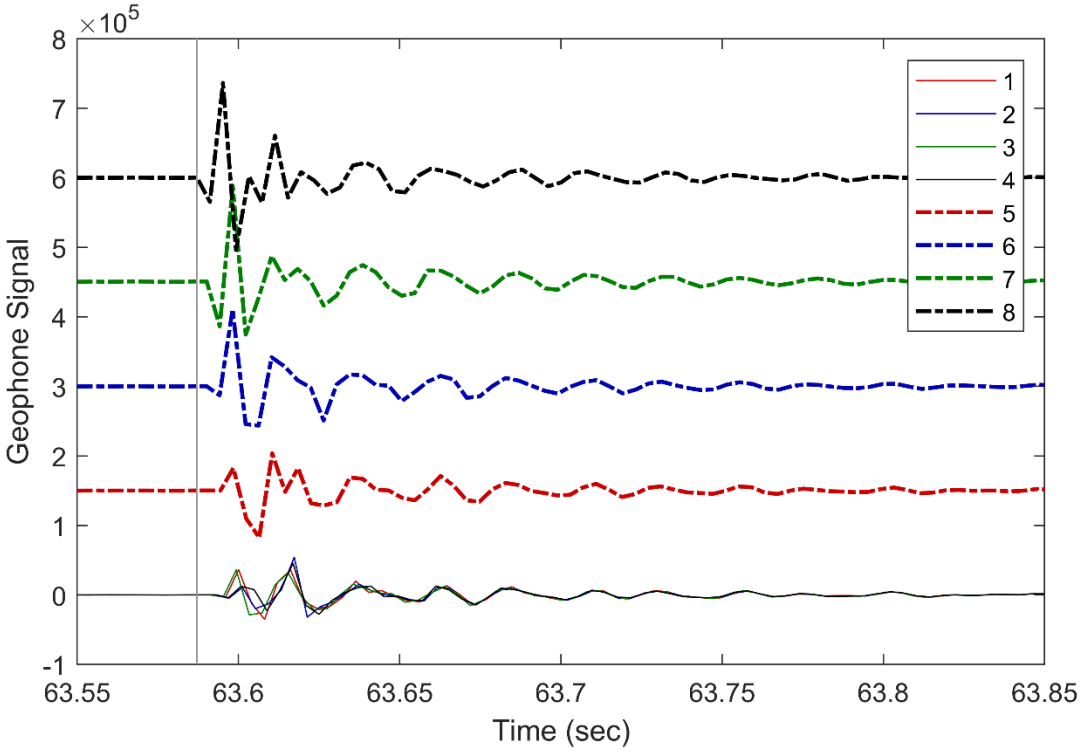
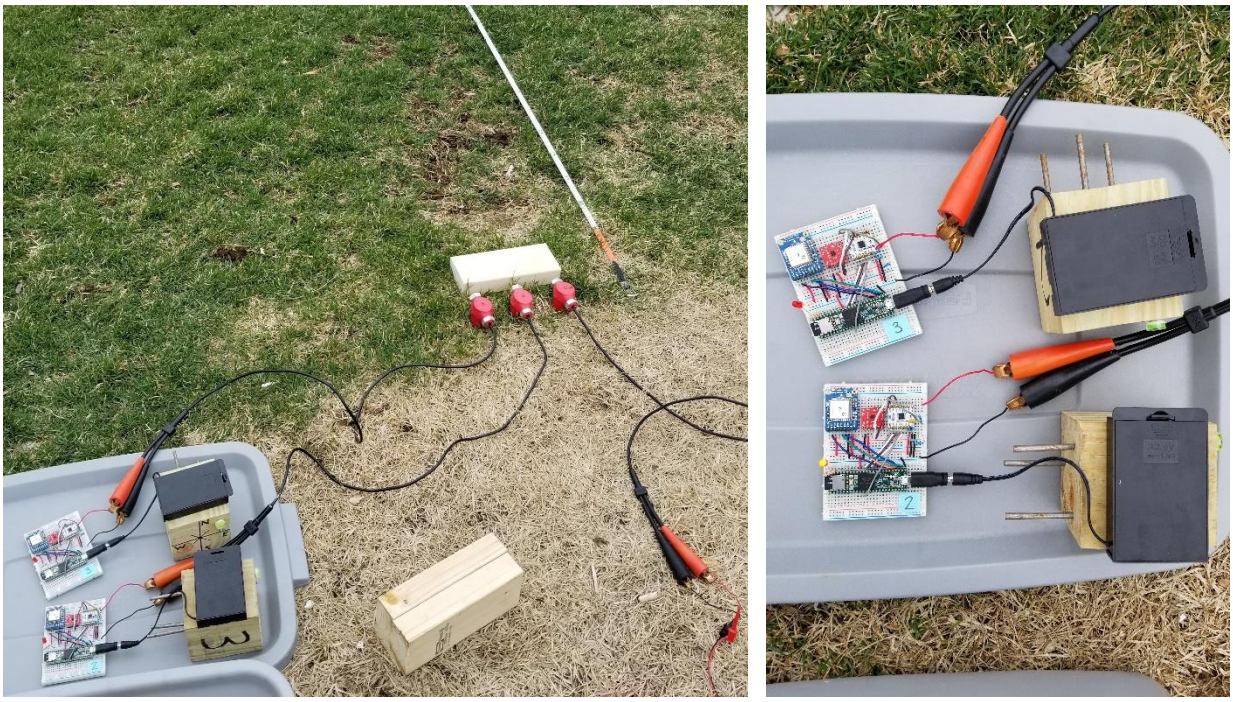


Figure 5.4.5: Results of the small-scale testing.

5.4.3 Field Testing

Once we had established the microcontroller systems worked properly in a small-scale test, we proceeded to test how well they functioned in the field. In addition, we wanted to test how including an RC filter would impact the recorded signals. To do this, we used three systems each with their own geophone. These systems were one microcontroller system without an RC filter (Seismic System 3), one microcontroller system with a first order RC filter (cutoff frequency of 125 Hz; Seismic System 2), and an oscilloscope. Using the oscilloscope as a reference, we can compare the quality of the signals recorded with the seismic sensors with and without a filter. For this test we use a sand-filled hammer striking a 20-cm length of 2x4 as the seismic source. Several blows were recorded at distances 1, 2, and 4 m away from the geophones. The setup showing the placement of the geophones and attachment to the seismic sensors can be seen in Figure 5.4.6. In order for the oscilloscope to be able to detect the signals



(a) (b)

Figure 5.4.6: Seismic field testing setup: (a) location of geophones and (b) connection of geophones to seismic sensors.

from the geophone, an amplifier with a built-in filter was used. This filter-amplifier was set to have a cutoff frequency of 125 Hz (to match the cutoff frequency of the RC filter of the microcontroller system) and adjust the gain by 20 dB.

Because the geophones are not calibrated and the oscilloscope does not record signals in real time, the signals needed to be adjusted manually so that the approximate amplitudes and timing match. To do this, the signals recorded by the seismic sensors were normalized by subtracting the mean and adjusted to approximately match the amplitude of the oscilloscope signals by dividing by a gain. To align the oscilloscope signals, the signals from the seismic sensors were plotted in real time and an offset was added to each oscilloscope signal to match it to the other two signals by eye. Since the timing accuracy of the microcontroller systems has already been established, it is not important that the oscilloscope signals be exactly aligned, rather, the alignment need only be close enough to compare the signals to one another. Any shift

between the signals from the two microcontroller systems can still be used to evaluate if there is a shift in the signals due to the RC filter.

The aligned seismic signals after data processing are shown in Figure 5.4.7. This figure shows the signals recorded by each of the three microcontroller systems for one blow each at distances 1, 2, and 4 m away from the geophones. For all three distances, the signals show a very

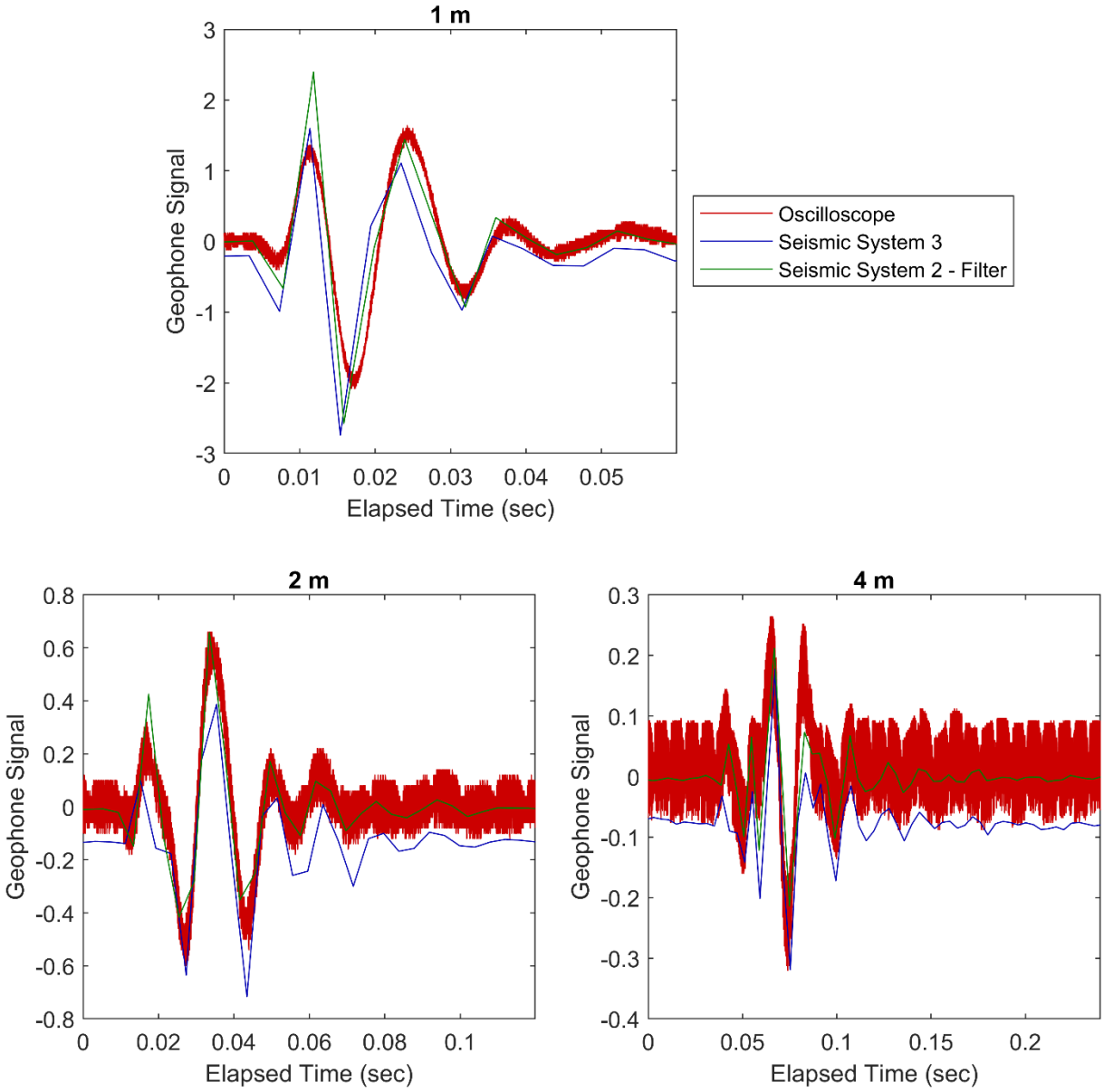


Figure 5.4.7: Recorded signals for source located at various distances from geophones.

close match in both shape and timing. The signals recorded by both seismic sensors, with and without the RC filter, very closely match the shape of the signal recorded by the oscilloscope, though with far less high frequency noise. The signal from the microcontroller system with an RC filter may or may not be shifted very slightly in time, however, this shift, if present, is very small (on the order of magnitude of less than 1 ms), does not influence the shape or quality of the signal, and should not significantly influence the timing accuracy.

In order to ensure the microcontroller systems were accurately capturing the frequency of the seismic signals, the power spectral densities (PSDs) were also analyzed. The PSDs were calculating using the pwelch function in MatLab. For the microcontroller sensors, 128 points centered around the signal were used to calculate the PSDs. Because of irregular steps in the data for Seismic Sensor 3, the mean of these 128 points was first subtracted from the signal before calculating the PSDs for this sensor. This removes a large peak in the PSDs that occurred at low frequencies due to the DC portion of the signal. For the oscilloscope, in order to increase the resolution of the PSDs, the oscilloscope signals were padded with a number of zeros equal to twice the number of points contained in the signal. The PSDs corresponding to the signals in Figure 5.4.7 are shown plotted in Figure 5.4.8. This figure shows a close match in the peaks corresponding to the frequency of each of the signals for all three systems. This suggests that the sampling rate of the microcontroller sensors is sufficient to adequately capture the frequency of the signals.

Without using a much higher sampling frequency for the seismic sensors (which is impractical due to limitations in memory), it is difficult to determine what affect the RC filter has on higher frequency components of the signal. However, the microcontroller systems were able

to capture seismic signals in the field well both with and without the filter. Because the filter had no negative impact on the quality of the recorded signals, it is good practice to include it.

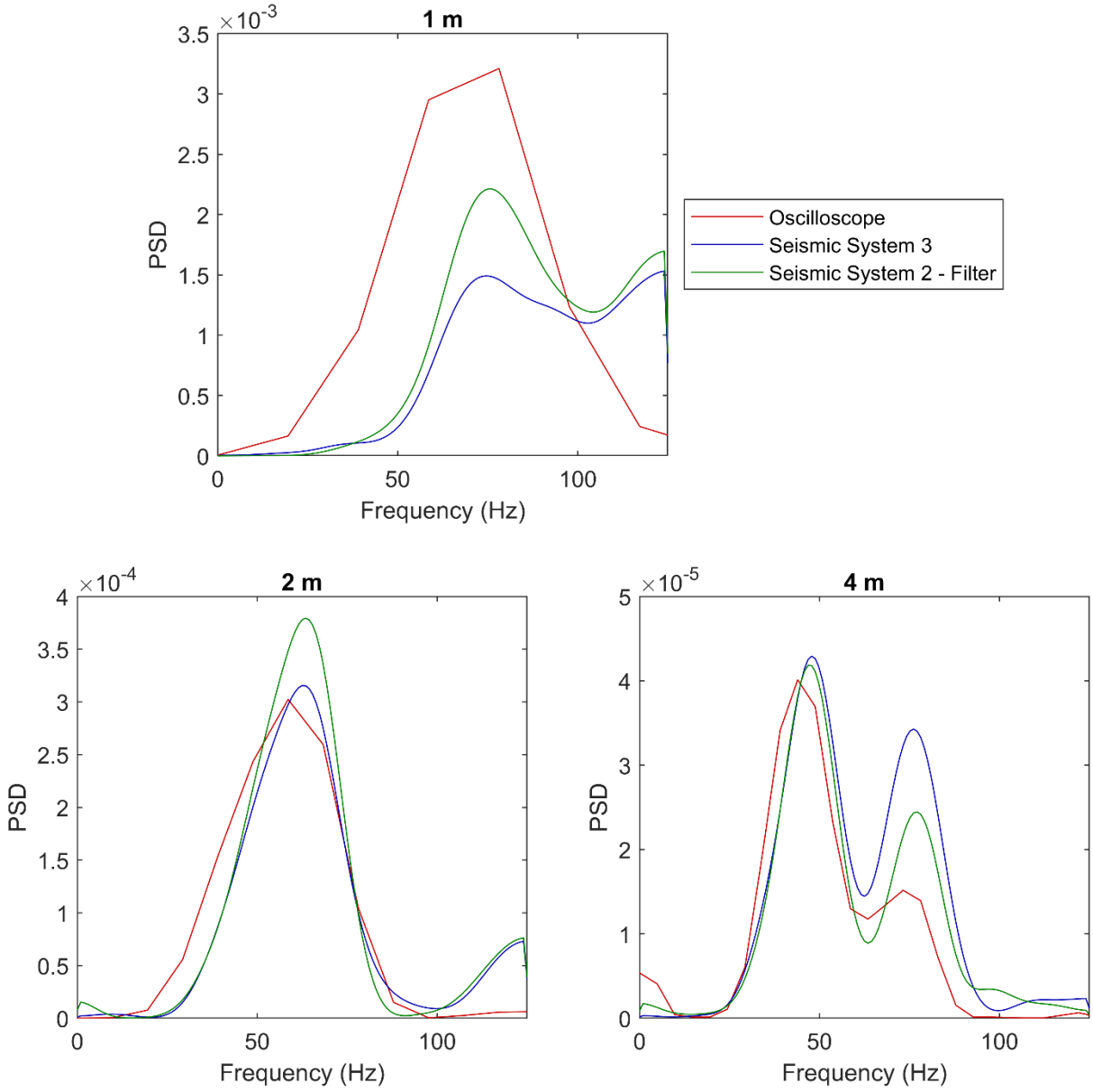


Figure 5.4.8: PSDs of signals for source located at various distances from geophones.

6 Low-Cost ER Tomography Sensor

Just like electromagnetic sensors, electrical resistivity tomography sensors can be used to evaluate the properties of the subsurface. Since electrical conductivity (the property measured by electromagnetic sensors) is the inverse of electrical resistivity, both types of sensors can often be used for the same applications. An advantage of electrical resistivity methods, however, is that the spacings of the electrodes can be varied to obtain measurements at different depths to create a two- or three-dimensional image of the resistivity of the subsurface. This is much more difficult to do with electromagnetic sensors, which often have a fixed coil spacing. In this section, the development of a laboratory-scale electrical resistivity sensor that can be used for environmental and hydrological imaging is presented.

6.1 System Setup and Wiring

For our electrical resistivity microcontroller system, we required the following components: 16 electrodes, an AC power supply, a microSD card for data storage, four multiplexers to select which electrodes the current is sent through and which the potential difference is measured across, 64 relays to act as a switch between each electrode and the positive current, negative current, and two digital pins used to measure potential difference, an ADC to increase the resolution of voltage measurements, a 100-Ω resistor to measure the supplied current across, and a microcontroller to control and run the entire system. In the following section, why specific components were selected and how they were combined to form a complete system is discussed.

To control the microcontroller system, we considered using an Arduino Mega 2560 microcontroller and a Teensy 3.6 microcontroller. The Arduino Uno microcontroller was not considered because it did not have enough digital pins to control the entire microcontroller

system. While the Arduino Mega 2560 microcontroller was initially chosen because of its simplicity of use (as an Arduino brand microcontroller) and large number of digital pins, we ultimately chose to use the Teensy 3.6 microcontroller. Initially, we thought to use the Teensy 3.6 microcontroller because it had a digital to analog converter (DAC) that had the potential to be used as a power supply, however, while we did not use the Teensy 3.6 microcontroller's DAC in this way, we chose to continue using the Teensy 3.6 microcontroller because its much faster processing speed allows the microcontroller to capture/use a much higher frequency AC signal and therefore allowed for much faster sample collection.

To create an electrical resistivity system, you need to be able to easily send current through and measure the potential across different electrodes. Each measurement only requires the use of four total electrodes: two electrodes to send current through the ground (one positive, one negative/ground) and two electrodes to measure the potential difference across. However, to obtain useful information, several different measurements must be taken at different locations to determine how the subsurface varies horizontally and with different electrode spacings to determine how the subsurface varies vertically. To do this efficiently, it is best to set up an array of several electrodes (16 in our microcontroller system) and have a system that can automatically take measurements for several different electrode combinations. To do this, you must have four sources (positive current, negative current/ground, and two digital pins to measure the potential) that are connected to each of the 16 electrodes and some method of turning the connection between the electrodes and the sources on and off. For our microcontroller system, we chose to use four multiplexers with 16 different channels (one multiplexer for each source, one channel for each electrode) to control which electrode each of the sources is connected to and 64 relays (one for each source-electrode combination) to act as a switch between each of the electrodes

and the four sources. The four sources mentioned above (positive current from power supply, negative current/ground from power supply, and two digital pins on Teensy 3.6 microcontroller) are each connected to every electrode through a relay so that they default to normally disconnected when the relay is turned off. The pin associated with opening and closing each relay is then connected to a multiplexer channel so that the multiplexers can be used to control which relays are turned on (and therefore which electrodes are connected to each of the four sources). A digital pin on the Teensy 3.6 microcontroller set to low is used as the signal sent through the multiplexers to switch the relays on. Therefore, for each measurement, the multiplexers are set to select which electrode should be connected to each of the four sources, the multiplexers are turned on to send the digital low signal through the multiplexers to turn the selected relays on, and the digital low signal turns on and closes the relays so that the selected electrodes are connected to the four sources. All the other relays remain turned off, default to open, and remain disconnected, so only the four selected electrodes are connected and used during the measurement. We chose to connect the sources to the electrodes through the relays in this way, rather than through the multiplexers themselves, due to concerns that random, inconsistent voltage drops may occur across the multiplexers that would influence the voltage measurements needed to determine the current and potential difference. In addition, using relays adds the ability for much higher voltages and currents to be sent through the electrodes than the Teensy 3.6 microcontroller is capable of handling, making it easier for the microcontroller system to be adapted for field use in the future (though an alternative means of measuring the higher voltages and current would still need to be added).

Each multiplexer is also connected to four digital pins on the Teensy 3.6 microcontroller that control which channel, and therefore which relay, the input is connected to and one digital

pin to control whether the multiplexer is open or closed (whether or not the input is connected to the channel selected as the output). In addition, each multiplexer and relay are connected to the Teensy 3.6 microcontroller's 3.3 V voltage supply and ground to provide power. Since the multiplexers default to open, each of the multiplexers is offset by one channel so that the supplied current and ground are not connected to the same electrode directly, which could short circuit the microcontroller system. As an additional safety measure, the digital pin used to send the low signal to close the relays is not set to low until after the multiplexers have been set up and closed, so that no relays are turned on until the microcontroller system is ready to begin taking measurements. The connections of the multiplexer channels, relays, and electrodes, as well as the rest of the components included in the microcontroller system, are shown in Figure 6.1.1.

Because the Teensy 3.6 microcontroller comes with a built-in microSD card slot, we chose to use a microSD card for data storage. Since microSD cards have sufficient storage space for the needs of our microcontroller system, choosing another data storage method will only add unnecessary complexity to the microcontroller system.

To take an electrical resistivity measurement, current is supplied through a pair of electrodes and the potential difference across a separate pair of electrodes is measured. While the Teensy 3.6 microcontroller can measure the potential of the different electrodes easily, it cannot directly measure the current. Therefore, to be able to measure the current we used a small resistor with a known resistance (100 Ω) and measured the potential before and after the resistor. The current supplied to the subsurface can then be calculated using Ohm's law.

To increase the minimum voltage difference we would be able to detect and improve the precision of our measurements, we included an ADS1220 24-bit ADC shield offered by

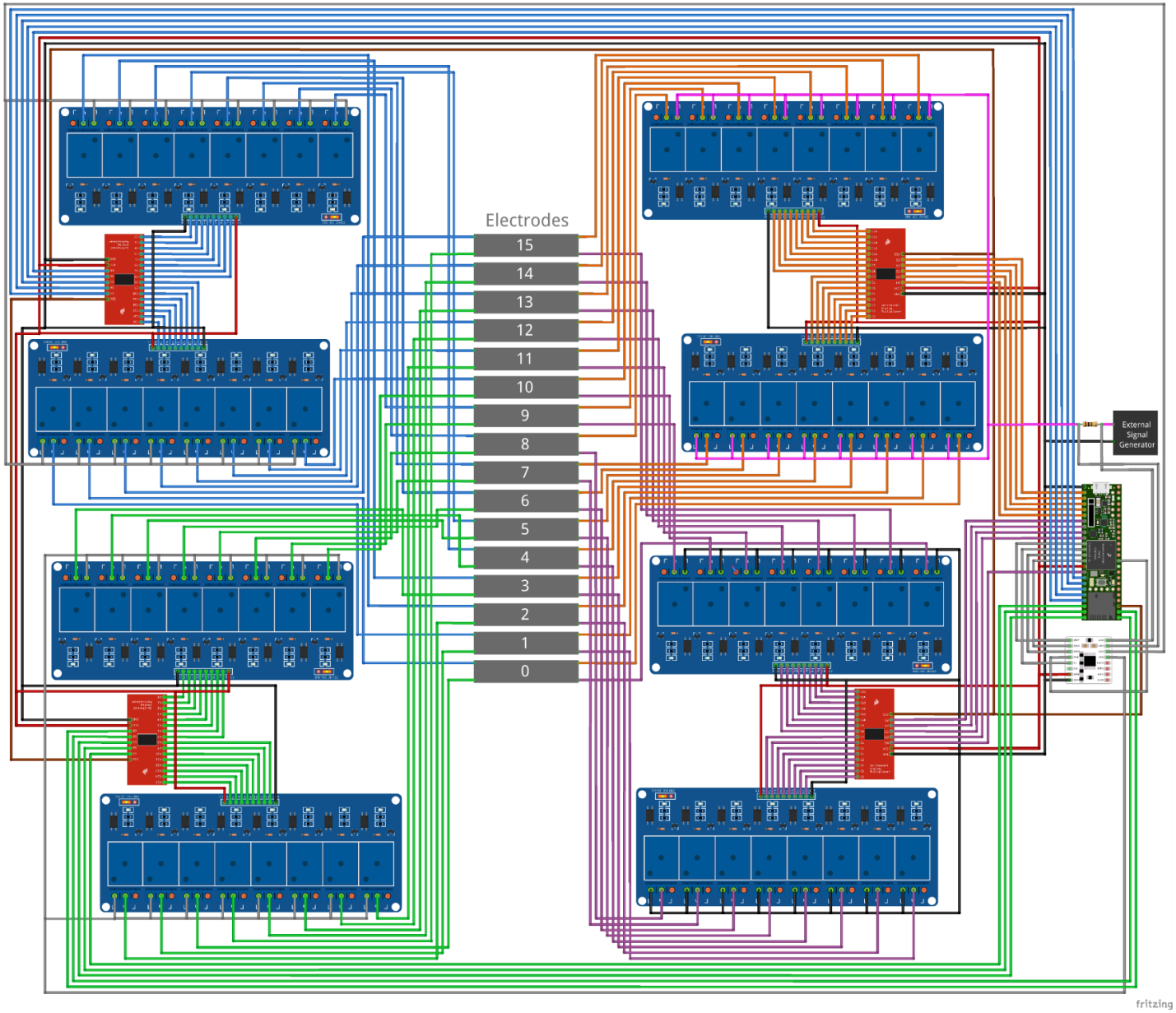


Figure 6.1.1: Diagram of wiring connections for 16 electrode electrical resistivity microcontroller system (made with Fritzing using parts from Bruneau, 2019, Skymoo, 2019, and other unknown sources).

Protocentral. We selected this ADC shield because it was one of the highest resolutions available that came on a shield that allows it to be easily interfaced with a microcontroller. To reduce the time it takes for the microcontroller system to complete a measurement and maximize the frequency of the AC signal we could use, we selected the settings which correspond to the fastest possible sampling rate the ADS1220 24-bit ADC shield was capable of (turbo mode and sampling rate of 2000 sps). We found that, when storing four consecutive voltage measurements per sample, this results in an average time between measurements of 2.15 ms and an effective sampling rate of about 465 Hz.

We initially hoped to use the Teensy 3.6 microcontroller's DAC as an AC power supply, so the entire microcontroller system could be completely self-contained and run by the Teensy 3.6 microcontroller. However, the Teensy 3.6 microcontroller's DAC cannot be used to supply a truly sinusoidal AC signal; the DAC can only set the output signal to a specified value. You can change the output value of the DAC, step by step, every loop to approximate a sinusoidal signal, however, this becomes more complicated if you want to create a signal and record it at the same time. Because we were concerned that this would result in the microcontroller system essentially recording measurements that correspond to a DC signal with the DC value of the signal adjusted between consecutive measurements, we decided that an external power supply with a truly sinusoidal AC signal would be better, despite having to add an extra component to the microcontroller system to do so. We selected the frequency of our AC signal based on the sampling rate of the ADS1220 24-bit ADC shield. Our goal was to choose a frequency such that we would be able to record 25 samples per cycle. Based on our sampling rate, this frequency is about 18.6 Hz as calculated below.

$$T = \left(2.149 \frac{ms}{sample}\right)(25\ samples) = 53.721\ ms \qquad (6.1.1)$$

$$f = \frac{1}{T} = \frac{1}{0.053721\ sec} = 18.6\ Hz \tag{6.1.2}$$

6.2 Code Design

Most Arduino programs use two main phases: a setup phase which occurs once at the beginning when the microcontroller system is powered on or a new code is uploaded and a loop phase which runs after the setup phase and repeats continuously until either power is lost or a new code is uploaded. An electrical resistivity system is typically used by setting up an array of electrodes, allowing the system to automatically collect apparent resistivity values for specified electrode combinations, then either running the system again with different parameters/specified electrode combinations or moving the entire array to a new location before repeating the process. Therefore, the entirety of the code was included in the setup phase so that data collection would occur once, then the microcontroller system would wait for the code to be reuploaded with or without adjustments before running again. This way, the user will have time to move the array or adjust various settings in the code before it runs a second time.

To calculate the apparent resistivity, the signals are normalized by subtracting off the mean and the RMS voltage at four locations (immediately before and after the 100-Ω resistor, used to calculate the current supplied to the subsurface, and at the two potential electrodes, used to calculate the potential difference) is calculated using the following equation.

$$V_{RMS} = \frac{\sqrt{\sum V_i^2}}{N} \tag{6.2.1}$$

In this equation, V_{RMS} is the RMS voltage of a location, V_i are all the normalized potentials at that location, and N is the total number of recorded potentials. The current and potential difference can then be calculated using these RMS voltages with the following equations.

$$I = \frac{\left|V_{RMS,C1} - V_{RMS,C2}\right|}{R_{ref}} \quad (6.2.2)$$

$$\Delta V = \left|V_{RMS,P1} - V_{RMS,P2}\right| \quad (6.2.3)$$

Where I is the current, $V_{RMS,C1}$ and $V_{RMS,C2}$ are the RMS voltages immediately before and after the 100 Ω resistor, R_{ref} is the measured voltage of the 100 Ω resistor, ΔV is the potential difference, and $V_{RMS,P1}$ and $V_{RMS,P2}$ are the RMS voltages of the two potential electrodes. The apparent resistivity of the measurement is calculated using the current and potential difference in the appropriate apparent resistivity equation for the selected array type (see Chapter 2.3.8).

As an alternative to apparent resistivity, the code can be easily altered to output the measured apparent resistance, which is calculated as the current measured across the 100-Ω resistor divided by the measured potential difference. This method was used during testing to compare the microcontroller system to an ABEM Terrameter SAS 300 B, a four-electrode system which directly outputs the measured apparent resistance. This was used as a means of evaluating the quality of our microcontroller system and verifying that the values of apparent resistance were reasonable. When intending to use our microcontroller system with Res2Dinv software, apparent resistivity should be used.

The code was designed to calculate the apparent resistivity for several different electrode combinations and configurations, then output a data file that is formatted and ready to be used with Res2Dinv software. The Res2Dinv software requires a data file that follows the following basic format. In the case of the Wenner array, there is no n integer and that entry is omitted.

Name of Survey Line

Smallest Electrode Spacing

Array type (1 = Wenner, 3 = Dipole-Dipole, 7 = Schlumberger)

Number of Data Points

Type of x-location (0 = location of first electrode, 1 = mid-point of array)

Flag for IP Data (0 = resistivity data only)

x-location Electrode Spacing n Apparent Resistivity

x-location Electrode Spacing n Apparent Resistivity

:

x-location Electrode Spacing n Apparent Resistivity

x-location Electrode Spacing n Apparent Resistivity

0

0

0

0

6.3 Electrode Configurations

For each of the arrays, the goal is to include as many electrode combinations for a 16-electrode array as possible without creating overlapping points in the pseudosection. For the Wenner array this is easily done by including all possible electrode combinations for all possible a spacings that can be included in a 16-electrode array (a spacings up to 5). All possible electrode combinations, and those included in the code, can be found in the appendix. For the Dipole-Dipole and Schlumberger arrays this becomes more challenging, since multiple electrode

combinations may be associated with the same point in the pseudosection for different combinations of a spacing and n separation. To determine which electrode combinations are plotted at the same point in the pseudosection, all the possible electrode combinations for the Dipole-Dipole (a spacings up to 7 and n separations up to 13, including non-integer values of n separation) and Schlumberger (a spacings up to 13 and n separation up to 7, including non-integer values of n separation) arrays must first be determined. Then, the point in the pseudosection that corresponds to each of these possible electrode combinations can be calculated using the appropriate equations (see Chapter 2.3.8) and compared to see which overlap. For the points that had overlapping positions in the pseudosection, one combination of a spacing and n separation was chosen, and the rest were omitted. For simplicity, electrode combinations with smaller a spacings were chosen over electrode combinations with larger a spacings corresponding to the same position in the pseudosection. This makes the pattern of electrode combinations more regular, which helps simplify the coding of the for loops necessary to iterate through the different electrode combinations. The electrode combinations that were ultimately selected and included in the code for the Dipole-Dipole and Schlumberger arrays can be found in the appendix.

6.4 Early Testing and Development

Different versions of the code were used during early testing. The purpose of this phase of testing was to discover how the different components of the microcontroller system worked and to develop and create a functioning system through trial and error. The microcontroller system used in early tests may have differed from the final microcontroller system in several ways, such as using an Arduino Mega 2560 microcontroller instead of a Teensy 3.6 microcontroller, using a DC signal, using the Teensy 3.6 microcontroller as the signal generator,

using the Teensy 3.6 microcontroller's built-in ADC instead of a higher resolution, external one, not using storage arrays, and outputting a file that is not formatted to work with Res2Dinv software and that contains additional information, such as the entire recorded signal and calculated current and potential difference. These early tests provided many insights into what worked best and led to many of the decisions we made when deciding which components to use in the microcontroller system and how to design the final code. These decisions were discussed in previous sections, and further discussions of why they were made, details of specific tests that were run, and how specific results led to those decisions are unnecessary. Early testing used a simple circuit of resistors to evaluate if the microcontroller system was working. The resistor circuits used in these early tests (first and second circuits in Figure 6.5.1) as well as more complicated circuits used in later tests and to test the final microcontroller system (third circuit in Figure 6.5.1) are discussed in the following section.

6.5 Testing of Resistor Circuits

Initial testing of the microcontroller system used resistors of known value to check that the microcontroller system could correctly measure the resistance between electrodes. For these tests (specifically the first and second circuits shown in Figure 6.5.1), resistance, rather than or in addition to apparent resistivity, was output to compare against the known resistances of the resistors used in the circuit. To output the resistance, the current supplied to the microcontroller system and the potential difference across the potential electrodes were determined in the same way, however, the measured current and potential difference were used in Ohm's law to calculate resistance instead of using the equations for apparent resistivity. Three different electrode circuits of increasing complexity were tested and are depicted in Figure 6.5.1.

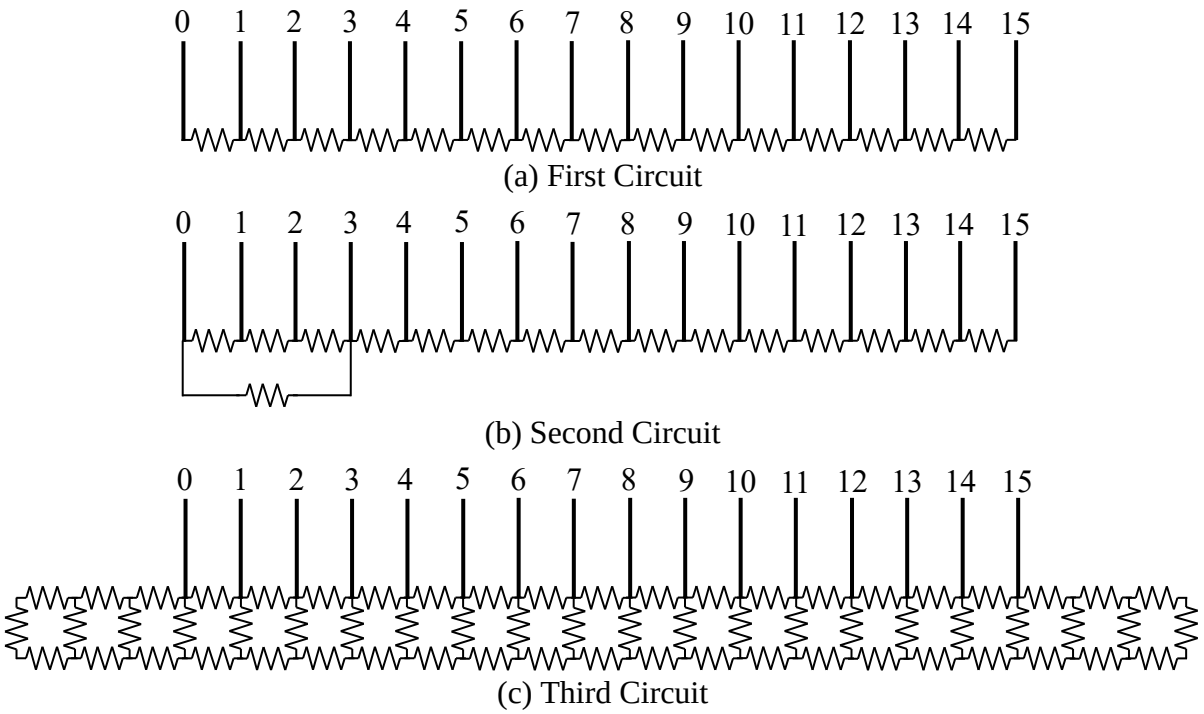


Figure 6.5.1: Circuits of resistors used to test electrical resistivity microcontroller system. All resistors used were 1000 Ω resistors.

In the first circuit, the measured resistance should simply be equal to the sum of the resistors between the potential electrodes. This circuit was used during early testing to evaluate if the microcontroller system was capable of correctly measuring a known resistance between electrodes. For the Wenner and Schlumberger arrays, the measured resistances corresponded to approximately the known resistances, however, this was not the case for the Dipole-Dipole array. Since the potential electrodes are not located between the current electrodes in the Dipole-Dipole array, no current travelled past the second current electrode. Therefore, there was no current and consequently no potential difference for the microcontroller system to measure at the potential electrodes. This limitation led to the second circuit.

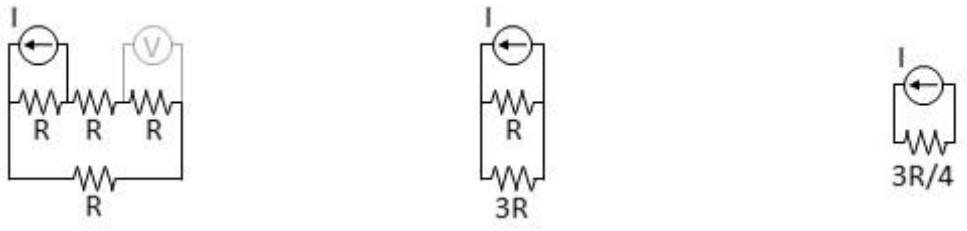
The purpose of the second circuit was to alter the first circuit in such a way that current would travel to the potential electrodes of the Dipole-Dipole array for the first measurement. If the microcontroller system was capable of correctly measuring the resistance for this first

measurement, then it likely would work for all measurements. However, due to the increased complexity of the circuit, the resistance between the potential electrodes will not simply equal the value of the resistor between them. Therefore, to test the second circuit, the apparent resistance had to be calculated. Since the Wenner and Schlumberger arrays have already been shown to work, I will only go through the calculation for the Dipole-Dipole array. First, we must consider the circuit of the first measurement, which is depicted in Figure 6.5.2a.

For this measurement, a constant, known current is supplied as shown so that current enters the circuit at electrode 0 and exits the circuit at electrode 1. As the current flows through the circuit, several potential drops occur as the current passes through the four resistors. The potential difference associated with the potential electrodes is the potential difference across the resistor between them; therefore, to determine what the apparent resistance will be we must determine what the potential drop will be across that resistor. First, you must simplify the circuit to one equivalent resistor to calculate the total potential drop that will occur. To simplify the circuit, you first add the resistors in series as shown in Figure 6.5.2b and in the following equation.

$$R_{eq} = R + R + R = 3R \tag{6.5.1}$$

Then you combine the resistors in parallel as shown in Figure 6.5.2c and in the following two equations.



(a) (b) (c)

Figure 6.5.2: Simplification of the circuit corresponding to the first measurement of the Dipole-Dipole array.

$$\frac{1}{R_{eq}} = \frac{1}{R} + \frac{1}{3R} = \frac{3}{3R} + \frac{1}{3R} = \frac{4}{3R} \tag{6.5.2}$$

$$R_{eq} = \frac{3R}{4} \tag{6.5.3}$$

Using this equivalent resistance of the circuit, the total potential drop can be calculated using Ohm's law. In the following equations I will use lowercase v, i, and r to indicate the form of Ohm's law being used and capital I and R to indicate the known current supplied to the microcontroller system and the known resistance of the resistors.

$$v = ir = \frac{3IR}{4} \tag{6.5.4}$$

Now that the total potential drop is known, it can be used to calculate the proportion of current that will travel through the resistor of interest. To do this, you use the equivalent resistances of the two parallel paths from Figure 6.5.2b which both experience a potential drop equal to the total potential drop. Once again using Ohm's law, the proportion of current that will pass through the three resistors can be calculated as in the equation below.

$$i = \frac{v}{r} = \frac{\frac{3IR}{4}}{3R} = \frac{I}{4} \tag{6.5.5}$$

This means that one quarter of the total current will travel through the resistor of interest. Using this, we can calculate what the measured potential drop across the resistor of interest will be, once again using Ohm's law.

$$v = ir = \frac{IR}{4} \tag{6.5.6}$$

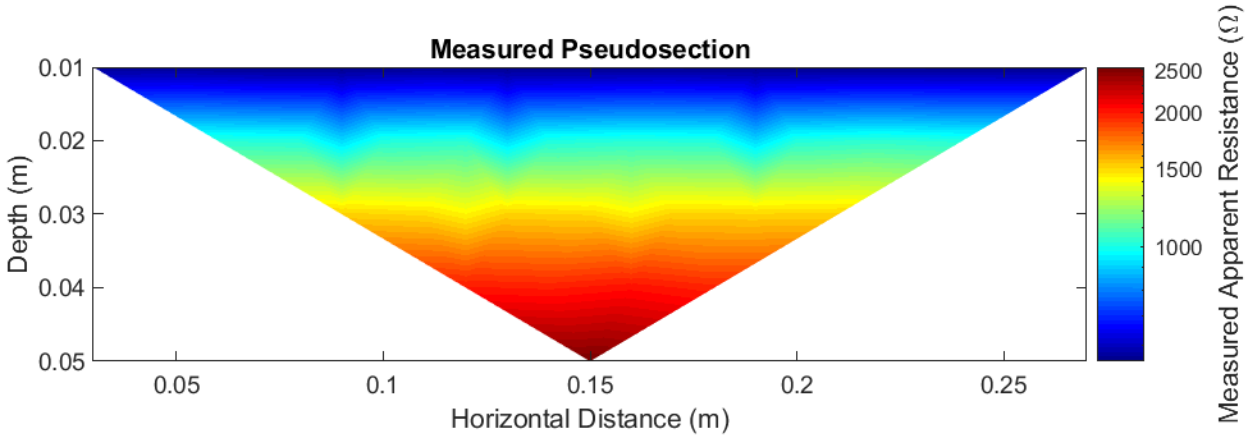
Finally, the apparent resistance can be determined by again using Ohm's law to calculate the resistance determined from the measured total current supplied to the circuit and the measured potential drop across the resistor of interest.

$$r = \frac{v}{i} = \frac{\frac{IR}{4}}{I} = \frac{R}{4} \tag{6.5.7}$$

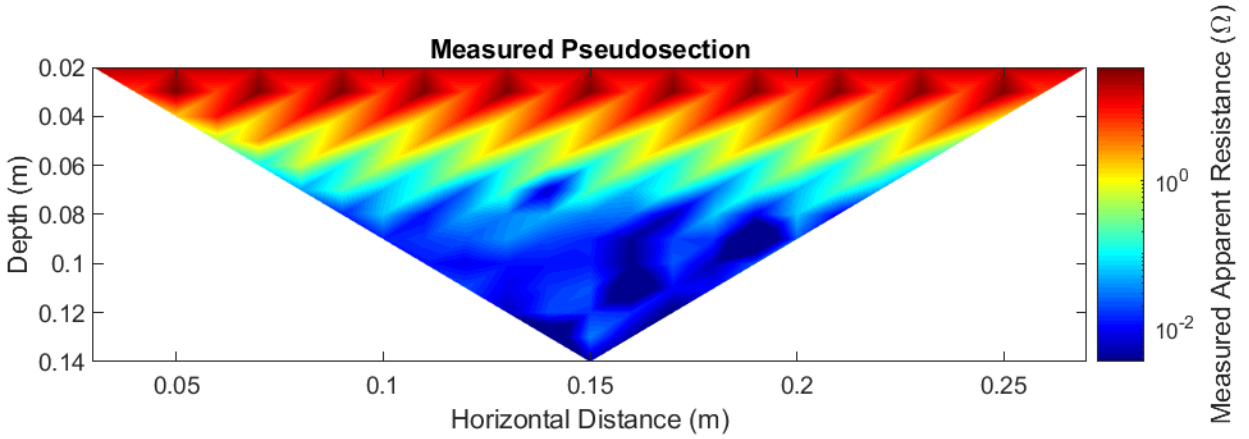
Therefore, when using 1000 Ω resistors, the measured resistance should be approximately 250 Ω. Testing of the second circuit for the Dipole-Dipole array did show a measured resistance around 250 Ω, suggesting that for all of the arrays, the microcontroller system was indeed capable of correctly determining the resistance between electrodes.

The next step in our testing was to develop a circuit that could be used to evaluate how well the microcontroller system worked by mimicking the subsurface. There were two main goals while evaluating system performance: first, to develop a circuit that would verify that the microcontroller system was correctly measuring resistance for all measurements of the dipole-dipole array and second, to develop a more complex circuit that could better approximate current flow through the subsurface. For this second goal, we wanted to increase the complexity enough that current would at least have alternative paths to follow rather than following a single, direct path between current electrodes, but without adding too much complexity that the circuit was difficult to create. For this purpose, we proposed the third circuit.

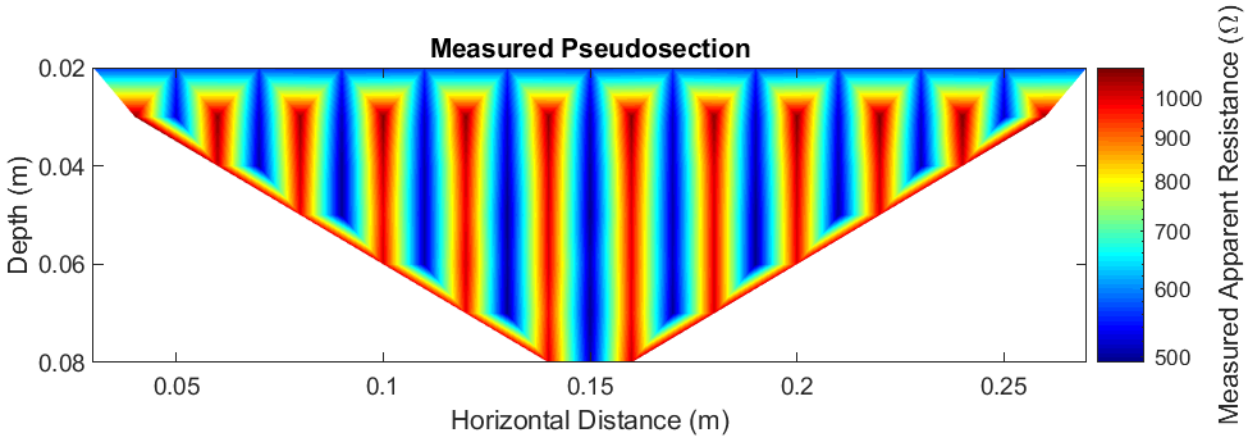
For the third circuit we added a second line of resistors with additional resistors connecting the two lines. These resistors force the current to take indirect paths that would otherwise be ignored. Figure 6.5.3 through Figure 6.5.5 show the results generated from testing with this circuit using the final microcontroller system. Figure 6.5.3 shows the pseudosection of measured resistances from Res2Dinv software for all three array configurations. While the exact resistance of each measurement is unknown, due to the symmetry of the circuit we would expect the pseudosections to reflect that symmetry. In addition, we expect that as the separation of the potential electrodes increases, the resistance between the electrodes should also increase. Therefore, for the Wenner array, since greater electrode separations correspond to greater depths



(a)



(b)



(c)

Figure 6.5.3: Res2Dinv results for measured resistance of an electrical circuit: (a) Wenner array, (b) Dipole-Dipole array, and (c) Schlumberger array.

in the pseudosection, we expect that for this array the resistance should increase with depth. For the Dipole-Dipole array, we would expect that as the potential electrodes move away from the current electrodes, the measured current should remain constant but the measured potential difference should decrease. Since greater separations between current and potential electrodes correspond to greater depths in the pseudosection for the Dipole-Dipole array, this should result in a decrease in resistance with depth.

While there are some variations in the pseudosections, especially for the Dipole-Dipole and Schlumberger arrays, in general the pseudosections are symmetrical, as expected, and variations may be explained by electrical noise. In addition, the resistance of the Wenner array does show the expected trend of increasing resistance with depth and the resistance of the Dipole-Dipole array shows the expected trend of decreasing resistance with depth. This is a good indication that the microcontroller system is likely capable of measuring the resistance of an electrical circuit.

Figure 6.5.4 shows how the measured resistance from the Dipole-Dipole array decreases as the separation between current and potential electrodes increases. In this figure, each line represents the measured resistances that correspond to a single pair of current electrodes. As expected, the measured resistance decreases approximately exponentially with increasing electrode separation. In addition, the curves for different pairs of current electrodes show very similar values of measured resistance for the same separation, as is expected for a homogenous medium. This corroborates the results of the Res2Dinv inversions and provides a further indication that microcontroller system is likely capable of accurately measuring the resistance of electrical circuits.

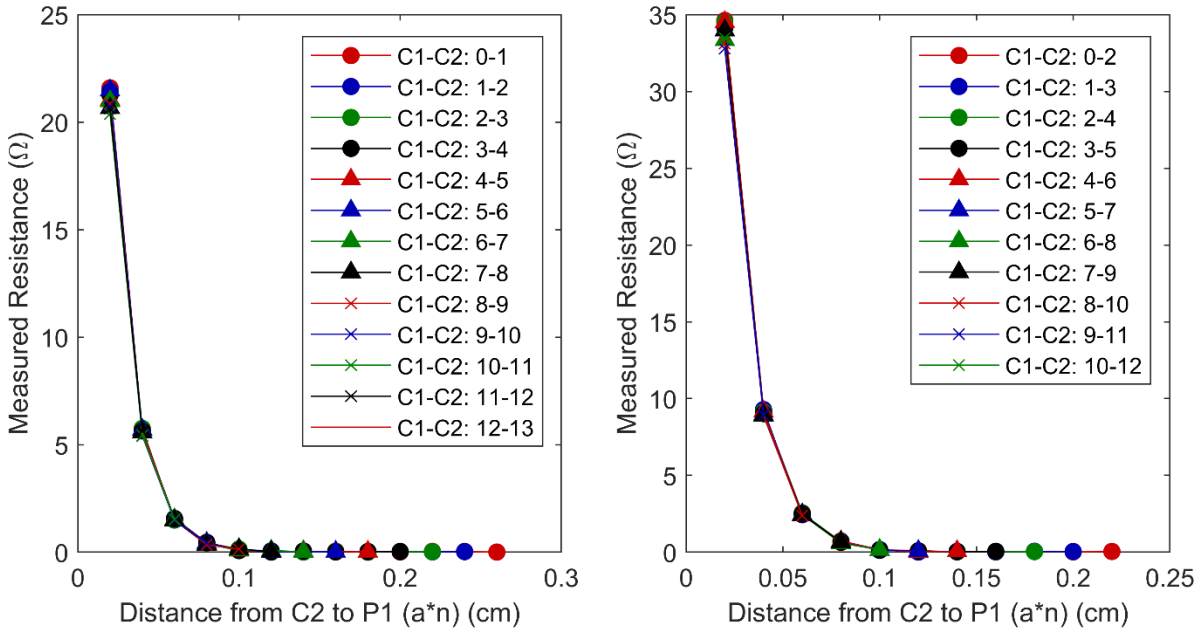


Figure 6.5.4: Change in measured resistance of electrical circuit with distance between current and potential electrodes for Dipole-Dipole array.

To ensure that the values of resistance are accurate in addition to measured resistances following expected behaviors, we compared the measured resistance of our microcontroller system with the professionally developed ABEM Terrameter SAS 300 B system for the first pair of current electrodes. Only the data for the first pair of current electrodes was compared to reduce the time necessary to complete the measurements using the ABEM Terrameter SAS 300 B. Since subsequent pairs of electrodes should show very similar values, comparing only the first pair should be sufficient to ensure that the values of measured resistance are correct. The results of this comparison are shown in Figure 6.5.5 which shows a very good match between the resistances measured with the two systems, though the resistances do begin to differ at high separations. This may be because the microcontroller system's resolution is not sufficient for measurements at these low currents and voltages. Based on these and previous results, we concluded that the microcontroller system can correctly measure the resistance of a circuit.

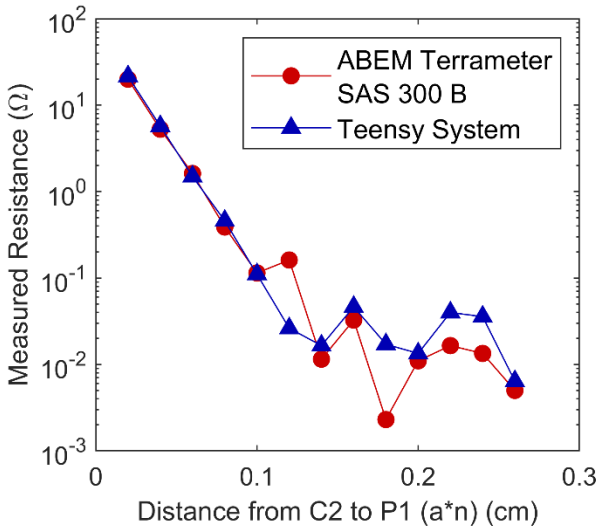


Figure 6.5.5: Comparison between measured resistances of an electrical circuit using our microcontroller system with the measured resistances of the ABEM Terrameter SAS 300 B.

6.6 Testing with Materials

Once we had verified that the microcontroller system could correctly measure the resistance of an electrical circuit, we needed to test that the microcontroller system could also measure the resistance of materials. For this stage of testing, we used a rectangular tank to determine whether the microcontroller system could correctly measure the resistance of real materials. The tank is 30.3 cm wide, 75.4 cm long, and 31 cm high.

For our lab testing we chose to use small, approximately 4.3 cm long screws as our electrodes. The screws were inserted into a 50 cm long section of a wooden meter stick to keep them spaced 2 cm apart and held in place by 2 small nuts located above and below the meter stick. To connect the screws to the relays, wires were wrapped below the top nut before tightening, secured by sufficiently tightening the lower nut, and connected to a breadboard containing the relays and the rest of the wiring and components.

6.6.1 Water

We first tested the microcontroller system using tap water because it is a relatively conductive material and is very close to homogeneous, which represents the simplest case. Once we had established that the microcontroller system works for the simple case, we could proceed to testing more complex materials. The tank was filled with water and the electrodes were suspended so that their tips were submerged. The electrodes were placed in the approximate center of the tank, as shown in Figure 6.6.1. The wires connected to the electrodes were then hooked up to the microcontroller system and the apparent resistivity was measured using all three arrays. To serve as a comparison and ensure the measured resistances were correct, the apparent resistance was also measured with the microcontroller system using the Dipole-Dipole array and the ABEM Terrameter SAS 300 B using the Dipole-Dipole array for the first pair of current electrodes. Both systems are shown connected to the electrode wires in Figure 6.6.2.

The apparent resistivities measured using the microcontroller system were analyzed using the Res2Dinv software to produce the inverse model resistivity cross sections. These cross sections are shown in Figure 6.6.3. For a homogenous material, we would expect the inverse model resistivity cross section to show a constant resistivity throughout and we would expect the cross sections to have the same resistivity regardless of which array was used. While there are some variations, all three cross sections show similar resistivity values and the magnitude of variation is relatively small. Some electrical noise is to be expected and may be contributing to this variation. The Dipole-Dipole array shows the most variation, which may be the result of electrical noise having a greater effect on the very small potential differences that are measured when the separation between current and potential electrodes is large. The Wenner array shows the least variation, which may be the result of larger measured potential differences when the

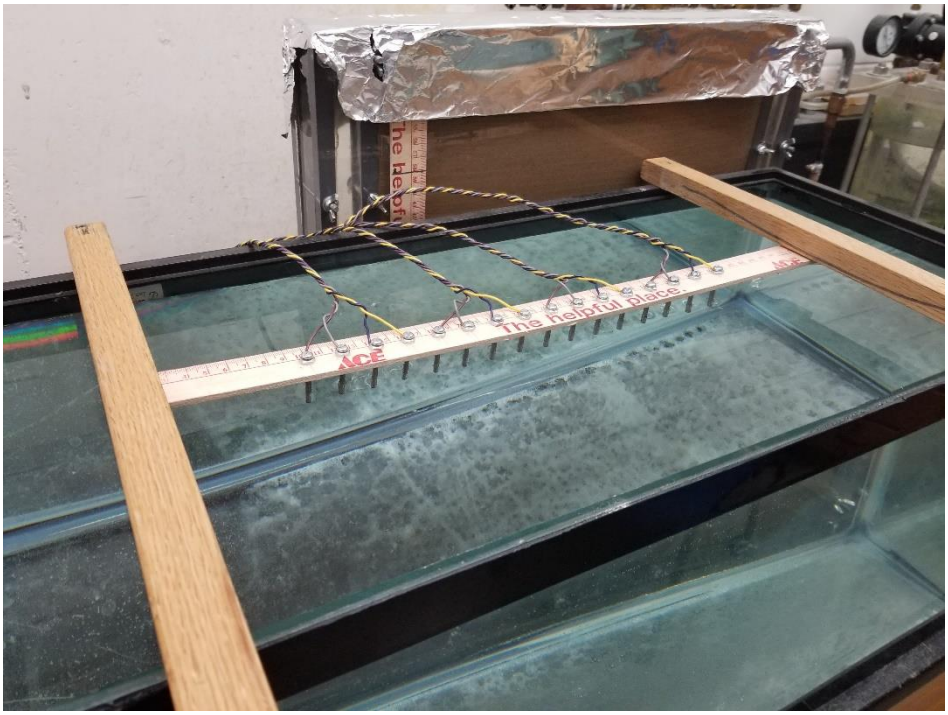
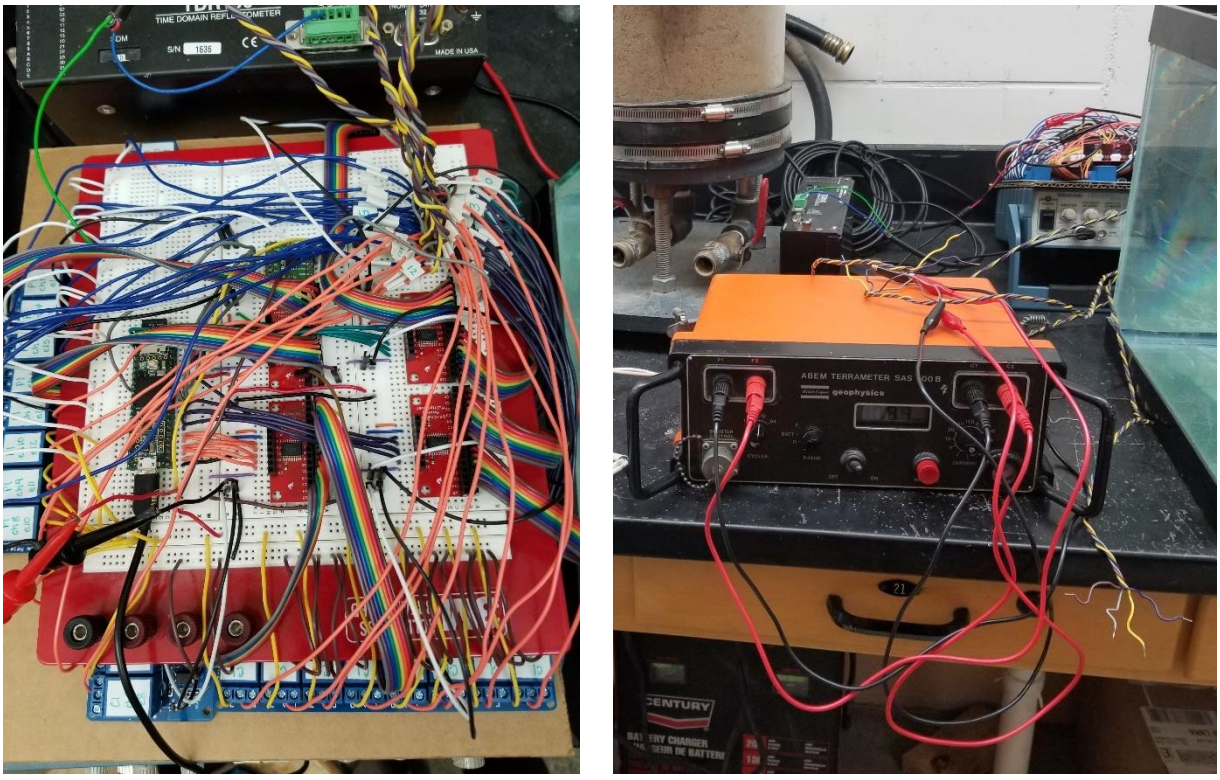


Figure 6.6.1: Electrodes suspended with tips submerged in water.



(a) (b)

Figure 6.6.2: Microcontroller system and ABEM Terrameter SAS 300 B connected to electrode wires: (a) microcontroller system and (b) ABEM Terrameter SAS 300 B.

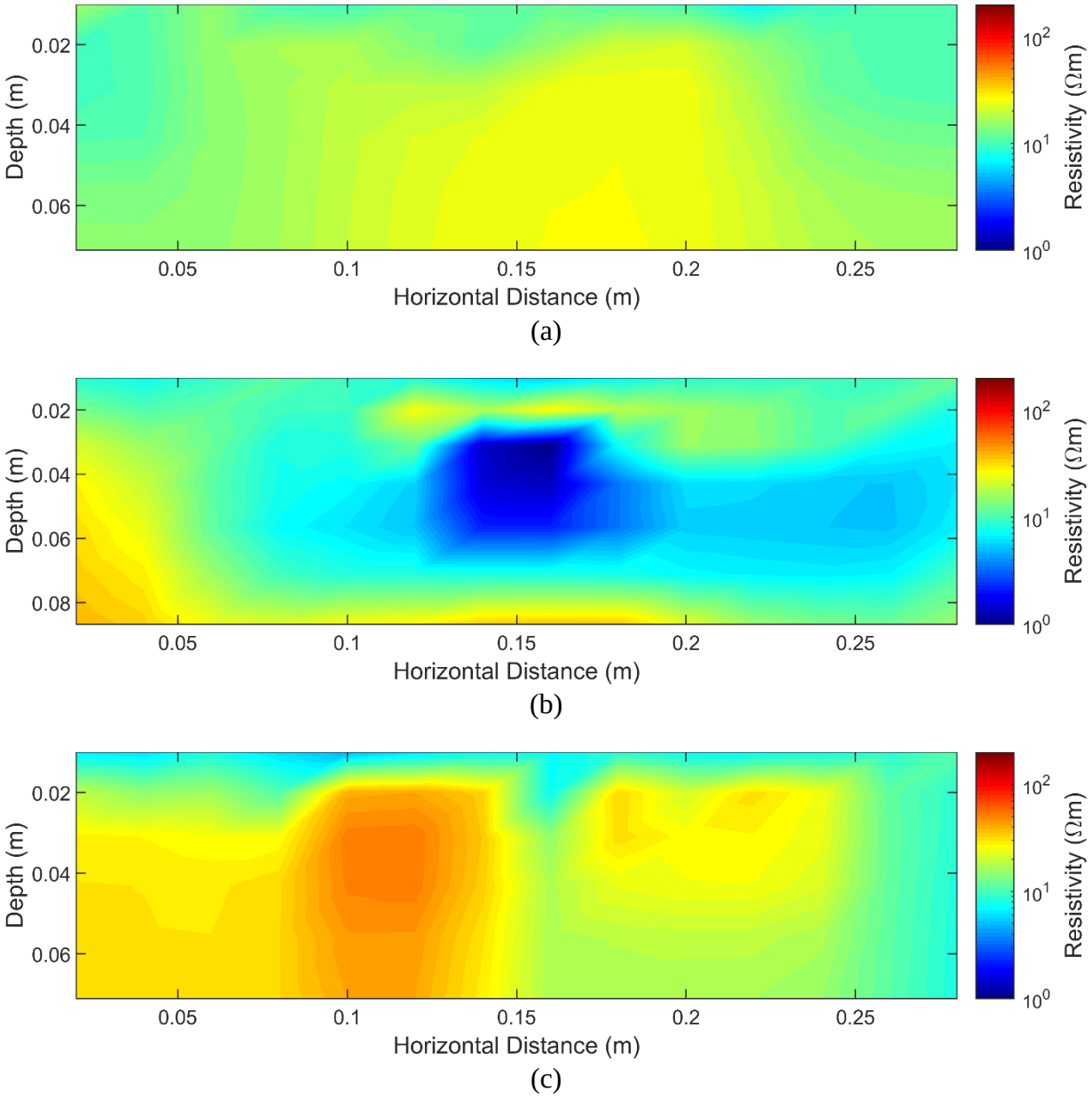


Figure 6.6.3: Res2Dinv inverse model resistivity cross sections for water measured with our microcontroller system: (a) Wenner array, (b) Dipole-Dipole array, and (c) Schlumberger array.

potential electrodes are located between the current electrodes (unlike the Dipole-Dipole array) and the spacing between electrodes is relatively large (relative to the Schlumberger array). These results suggest that the microcontroller system may be capable of measuring the resistance of materials.

For the Dipole-Dipole array, as the potential electrodes move away from the current electrodes, we expect the measured resistance to decrease approximately exponentially. In addition, the measured resistance should be the same for different electrode pairs with the same geometry (a and n spacing). To evaluate if the results follow this behavior, the measured resistance as the separation between current and potential electrodes increases were plotted as shown in Figure 6.6.4 for each pair of current electrodes. As expected, the results show that the measured resistance decreases with separation and the values of measured resistance are similar for different curves indicating when the electrode geometry is the same, so is the measured resistance. This provides further proof that the microcontroller system is likely capable of measuring the resistance of materials.

Finally, to ensure that the magnitude of the measured resistances is correct, the resistances measured with the microcontroller system and the ABEM Terrameter SAS 300 B

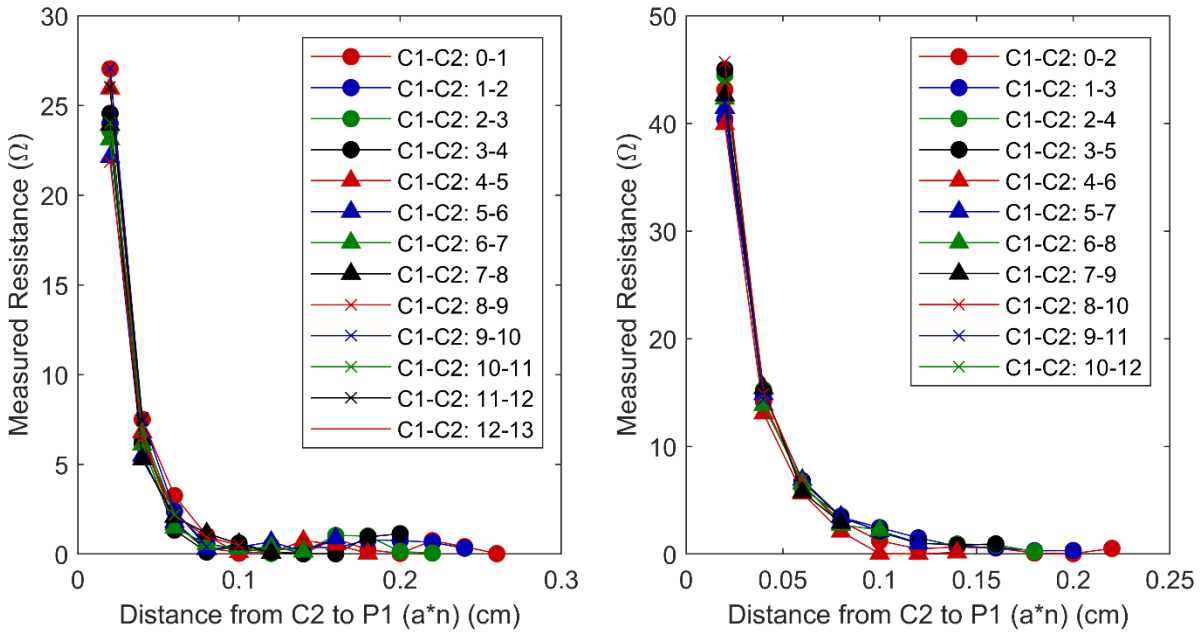


Figure 6.6.4: Change in measured resistance of water with distance between current and potential electrodes for Dipole-Dipole array.

were compared for one pair of current electrodes for the Dipole-Dipole array. These results are shown in Figure 6.6.5. Both systems measured very similar values of resistance at low separations, however, once again the resistances begin to differ at high separations. The magnitude of these differences is small, however, indicating the microcontroller system is capable of accurately measuring the resistance of materials. Therefore, we can conclude that the microcontroller system is capable of accurately measuring the resistance of real materials in addition to electrical circuits.

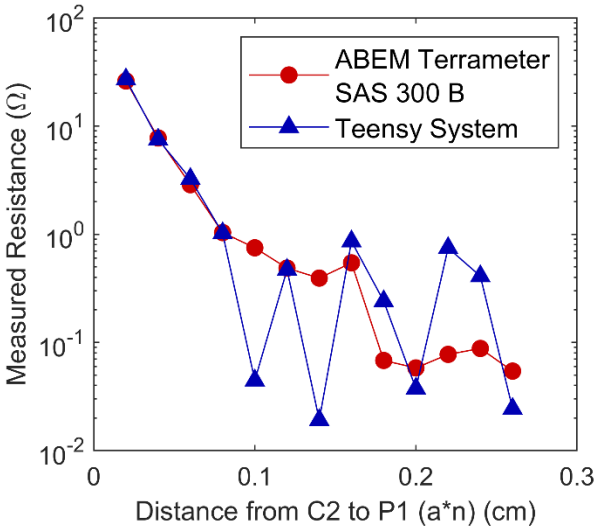


Figure 6.6.5: Comparison between measured resistances of water using our microcontroller system with the measured resistances of the ABEM Terrameter SAS 300 B.

6.6.2 Soil

Now that we had verified the microcontroller system could be used to measure the resistivity of real materials, we could move on to testing more complex materials and evaluating the quality of the results. For this, we tested a sand in both a saturated and unsaturated state. In order to be able to adjust the water level once the tank was filled with sand, we installed two clear plastic pipes in two opposite corners of the tank. The pipes had an outer diameter of 2.3 cm and wall thickness of 0.25 cm. Holes were drilled approximately every 2 cm along the length of

these pipes to allow water to pass through and a geotextile was used to seal off the corner to prevent the soil from entering. This design allows us to adjust and view the water level in the pipes, therefore adjusting the water level in the sand.

To fill the tank with sand and ensure it was completely saturated (or as close as reasonably possible), we first filled the tank with water and sprinkled sand in slowly, allowing it to settle to the bottom. Approximately 77.3 kg of sand was added in this manner, filling the tank to a height of 20 cm. The water level was then lowered to be even with the top of the sand for the saturated condition.

To test the sand in the saturated condition, the electrodes were placed in the approximate center of the tank and pushed into the sand until the ruler made contact. The ruler was slightly warped so that the ends were curled up away from the sand, so two weights were used to hold the ruler down and in place. A picture showing the electrodes in place can be found in Figure 6.6.6. Because the ruler was made of wood, once it was in contact with the sand it began absorbing water. In addition, significant evaporation occurred. Therefore, while testing the saturated sand water occasionally needed to be added to the reservoirs to keep the water level equal with the top of the sand.

For the unsaturated condition, the reservoirs were completely emptied and the sand was allowed to drain/dry over the course of a weekend. Due to the losses of water that occurred from both absorption by the wooden ruler and evaporation, for this test the ruler was not placed in contact with the soil and the tank was left covered with aluminum foil to mitigate evaporation. Two smaller weights were used to hold the ruler in place without forcing it into contact with the sand.

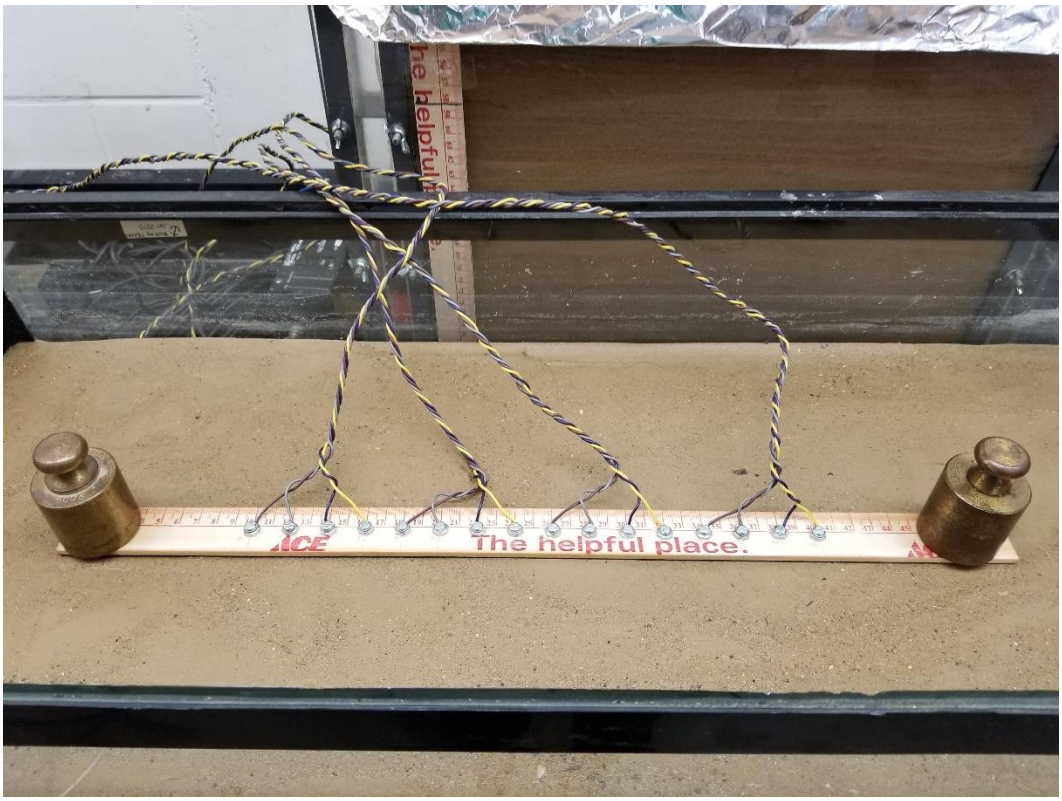


Figure 6.6.6: Placement of electrodes in saturated sand.

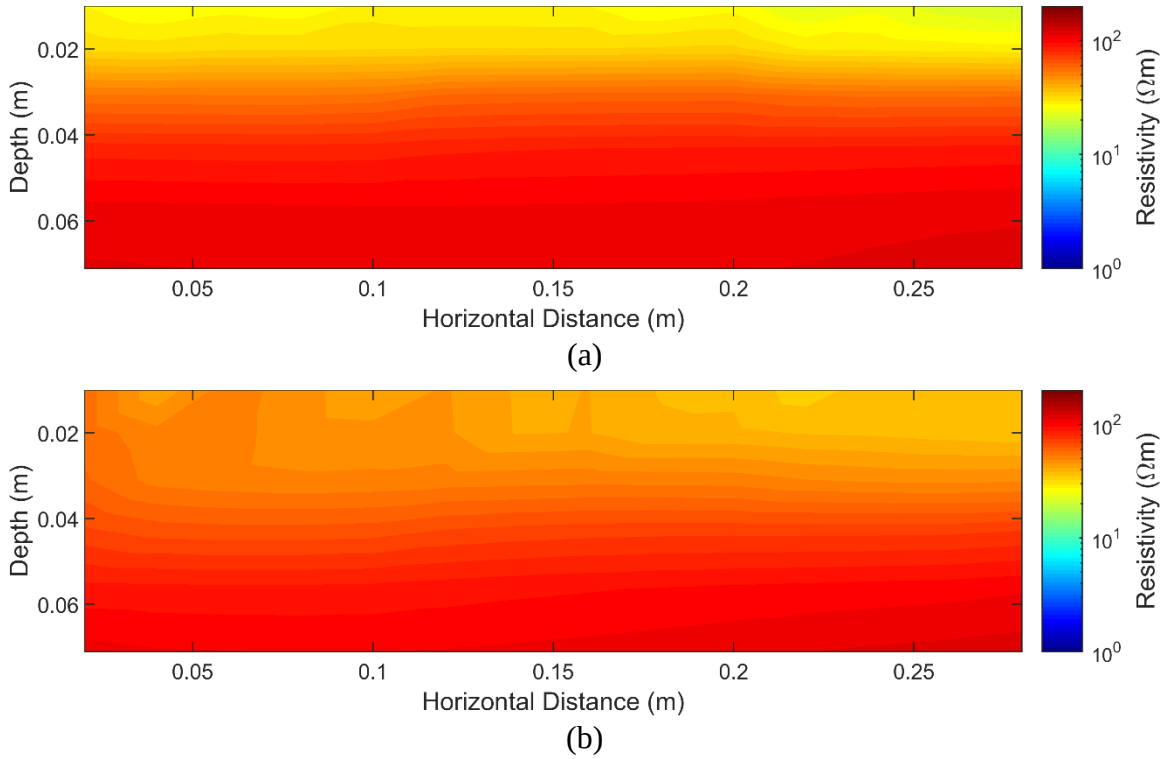
Just like when testing with water, for both the saturated and unsaturated conditions the wires connected to the electrodes were connected to the microcontroller system and the ABEM Terrameter SAS 300 B for testing. Testing consisted of measuring the apparent resistivity of all three arrays with the microcontroller system, the apparent resistance of the Dipole-Dipole array with the microcontroller system, the apparent resistance of the Wenner array with the ABEM Terrameter SAS 300 B, and the apparent resistance of the Dipole-Dipole array for one pair of current electrodes with the ABEM Terrameter SAS 300 B. In order to evaluate the quality of the inverse model resistivity cross sections, the apparent resistances of the Wenner array measured with the ABEM Terrameter SAS 300 B were converted to apparent resistivities using the appropriate equation. The Res2Dinv software was used to determine the inverse model resistivity cross sections for all three arrays measured by the microcontroller system and for the Wenner

array measured by the ABEM Terrameter SAS 300 B. The apparent resistances measured by the microcontroller system using the Dipole-Dipole array were also evaluated by assessing how the measured resistances changed with increased separation between current and potential electrodes, as well as by comparing the values from one set of current electrodes to the values measured by the ABEM Terrameter SAS 300 B. All of these results are shown in the following figures.

Figure 6.6.7 through Figure 6.6.10 show the results from the Res2Dinv inversion. We would expect that the resistivity should be relatively constant throughout since the sand was close to a homogenous material. We expect that the cross sections should show the same resistivity for different arrays and that the resistivity should be higher than the resistivity of the water since sand is a less conductive material. In addition, we would expect the drained sand to have a higher resistivity than the saturated sand since it contains less of the more-conductive water. Finally, our expectation is that the Wenner cross sections of both systems should look similar.

The results for the Wenner and Schlumberger arrays look very similar. For these arrays, the cross sections all show an increase in resistivity with depth. Comparing the saturated and drained sands, the drained sand shows higher resistivities near the surface, however, at larger depths the drained sand actually showed a slightly lower resistivity than the saturated sand. This leads to the saturated sand having a larger gradient in resistivity with depth than the drained sand. The cross sections for the Wenner array from both systems look very similar, however, the cross section for the ABEM Terrameter SAS 300 B had slightly lower resistivities.

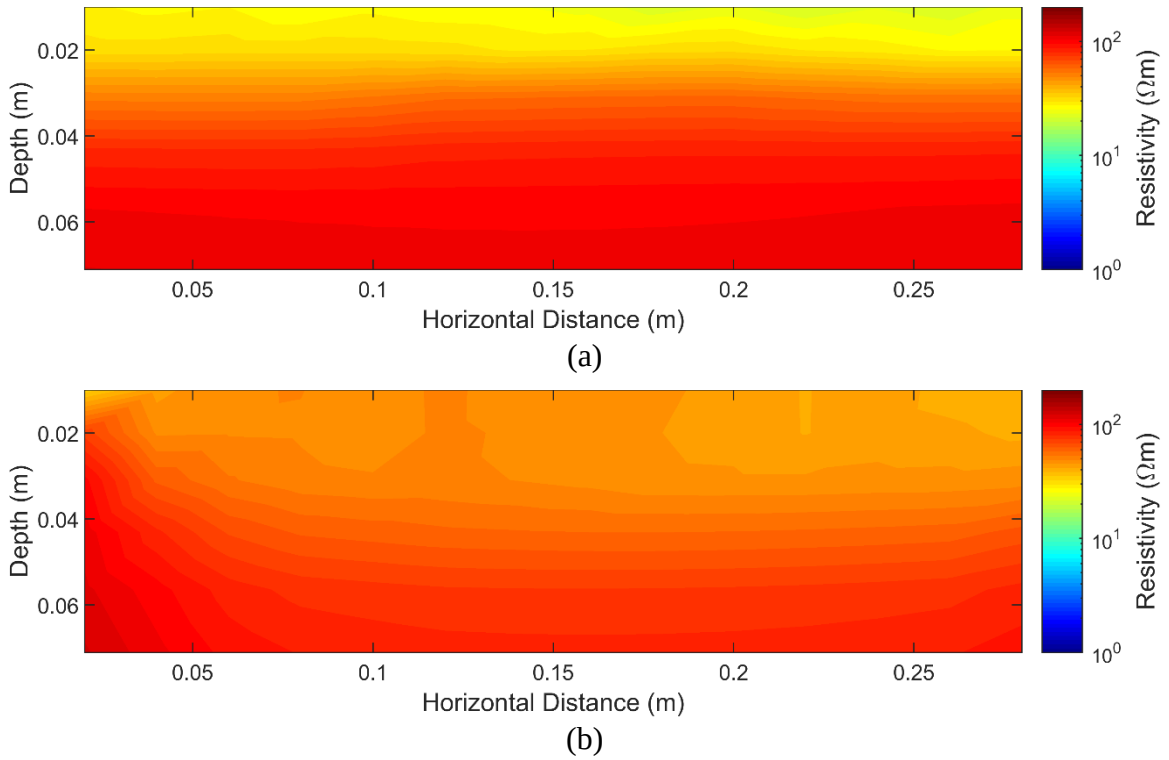
The cross sections for the Dipole-Dipole array show much lower resistivities than the other arrays at depth, but similar resistivities near the surface. The difference in resistivity at



(a)

(b)

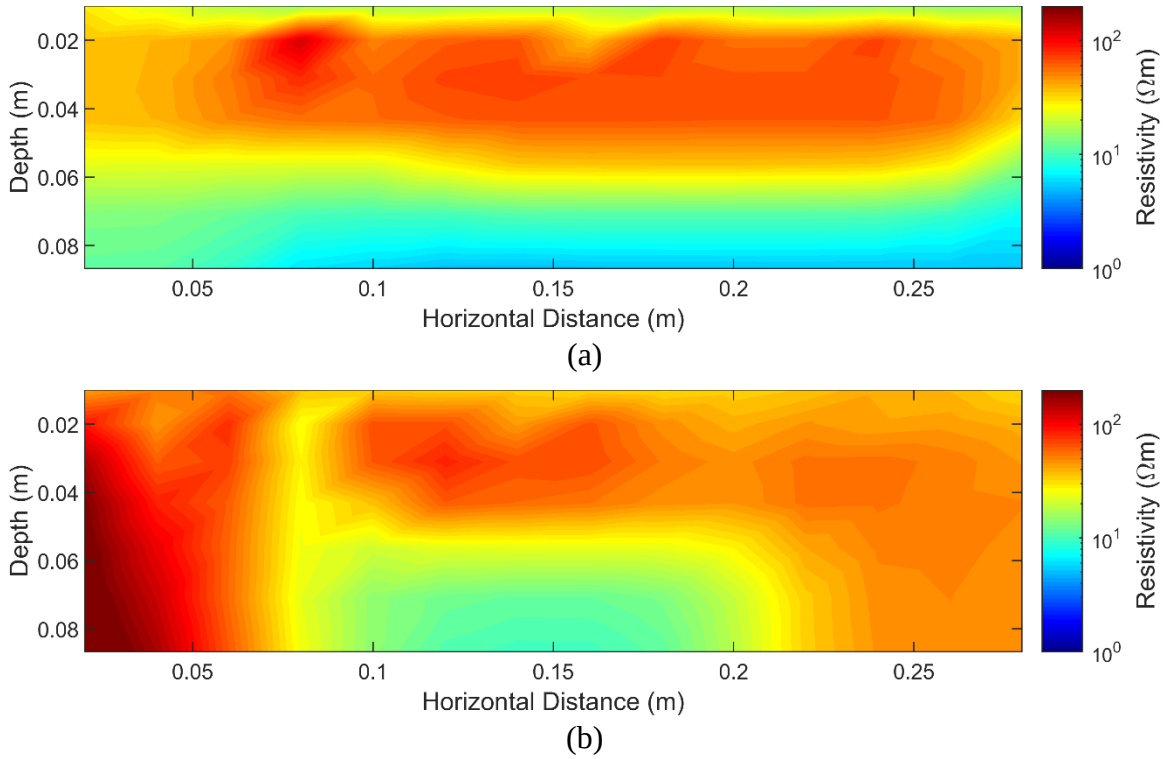
Figure 6.6.7: Res2Dinv inverse model resistivity cross sections for soil measured with our microcontroller system using the Wenner array: (a) saturated soil and (b) drained.



(a)

(b)

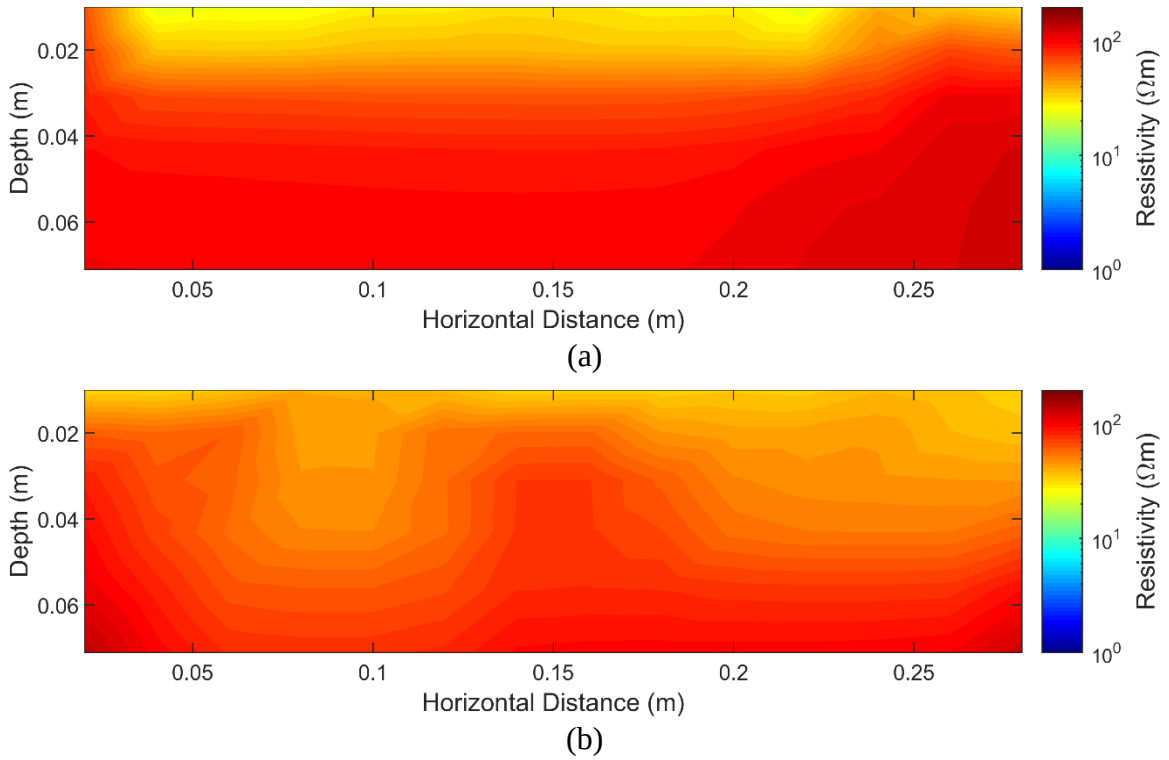
Figure 6.6.8: Res2Dinv inverse model resistivity cross sections for saturated soil measured with the ABEM Terrameter SAS 300 B using the Wenner array: (a) saturated soil and (b) drained.



(a)

(b)

Figure 6.6.9: Res2Dinv inverse model resistivity cross sections for saturated soil measured with our microcontroller system using the Dipole-Dipole array: (a) saturated soil and (b) drained.



(a)

(b)

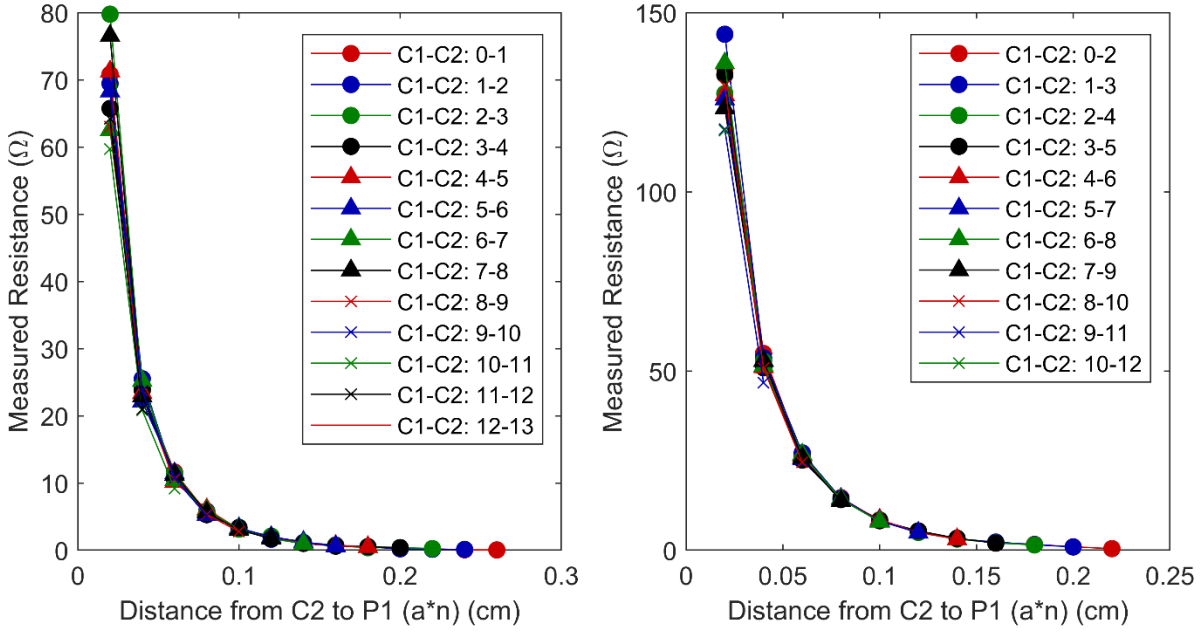
Figure 6.6.10: Res2Dinv inverse model resistivity cross sections for saturated soil measured with our microcontroller system using the Schlumberger array: (a) saturated soil and (b) drained.

depth may be due to the very small potential differences that are measured at high separations between current and potential electrodes, which correspond to greater depths in the cross section. If not enough current reaches the potential electrodes to create a detectable potential difference, a lower resistivity than the true resistivity may be measured. The cross sections for the Dipole-Dipole array do, however, also show an increase in resistivity from the saturated to drained sand.

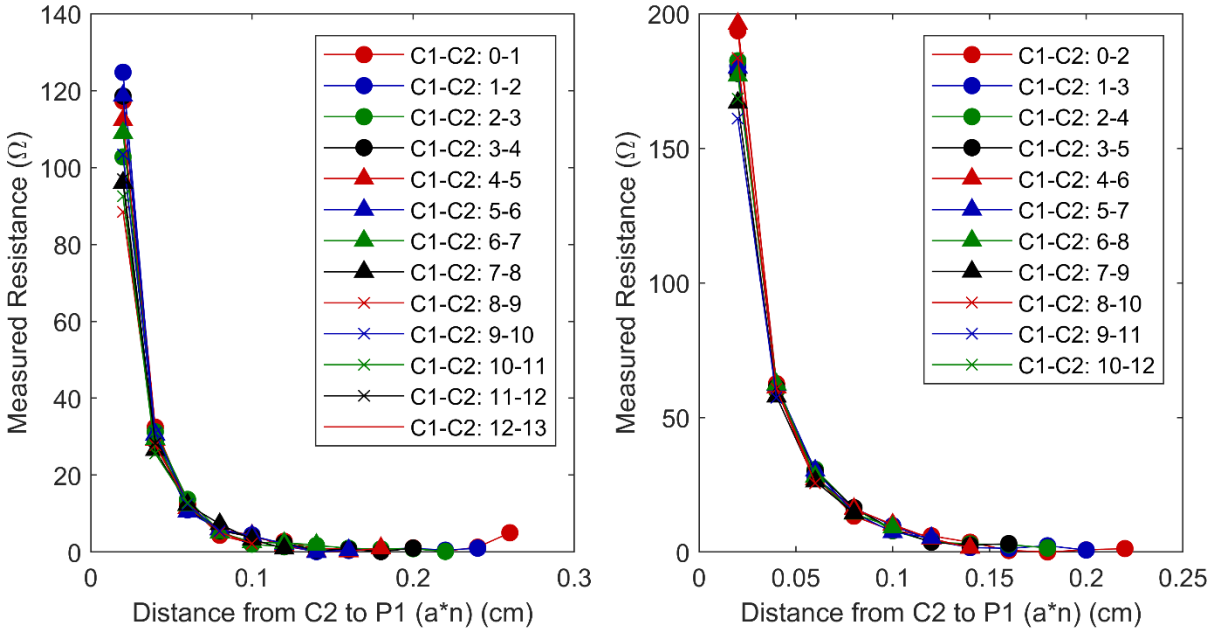
From the close match in cross sections for the Wenner array measured by both systems and the results showing the expected increase in resistivity for the drained soil, we can conclude that even for more complex systems the microcontroller system is capable of correctly measuring the resistivity. This conclusion is also verified by closer examination of the measured resistances of the Dipole-Dipole array.

The resistances measured by the microcontroller system for the Dipole-Dipole array are shown plotted in Figure 6.6.11. This plot shows how the resistances vary for each pair of current electrodes as the potential electrodes move farther away. The measured resistances show the expected trend of decreasing approximately exponentially with increased separation between current and potential electrodes for both the saturated and drained conditions. Similarly, the curves for different current electrodes match well suggesting that the same resistance is measured when the electrode geometry is the same. This further confirms that the microcontroller system can correctly measure the resistance of more complex, real materials.

Finally, the magnitude of the resistances measured using the Dipole-Dipole array are compared in Figure 6.6.12 for both systems. As expected, these values show a close match for both the saturated and drained conditions at low separations. For the saturated soil, the measured resistances matched well even for high separations, however, the measured resistances for the drained soil differ significantly for the largest separations. One explanation for this may be that,

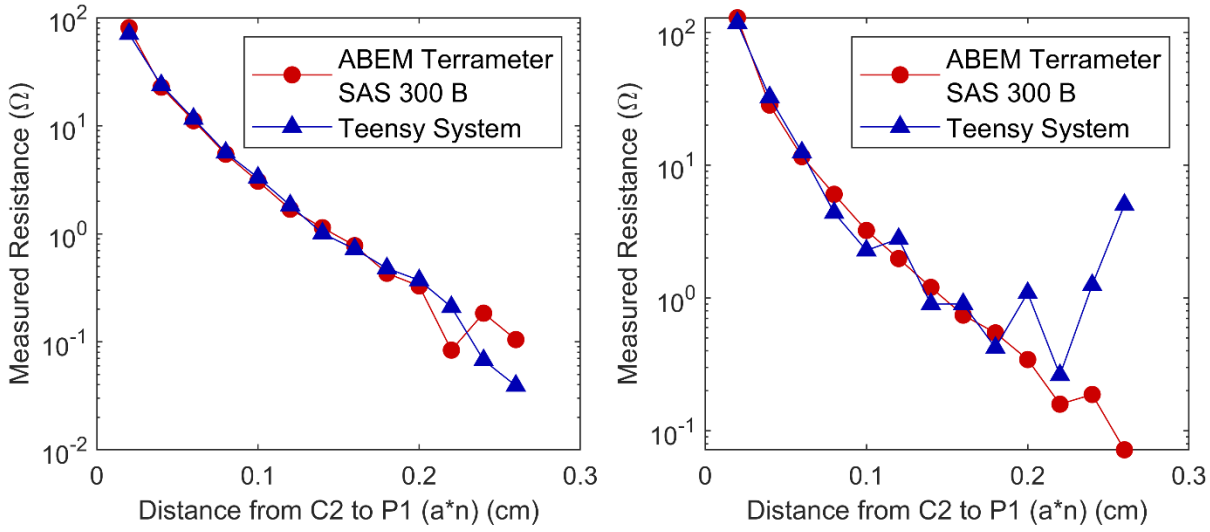


(a)



(b)

Figure 6.6.11: Change in measured resistance of saturated soil with distance between current and potential electrodes for Dipole-Dipole array: (a) saturated soil and (b) drained soil.



(a) (b)

Figure 6.6.12: Comparison between measured resistances of saturated sand using our microcontroller system with the measured resistances of the ABEM Terrameter SAS 300 B: (a) saturated soil and (b) drained soil.

due to the decrease in the amount of conductive water in the system, the amount of current that reaches the potential electrodes for these large separations and the corresponding potential difference is below the resolution of the microcontroller system. Because the magnitudes of current and voltage at large separations is small, this suggests that improvements in resolution for large separations may be necessary, but it does not indicate the microcontroller system is incapable of correctly measuring resistance. Along with the matching results from the Wenner inverse model resistivity cross sections, this further shows that the microcontroller system can correctly measure both changes in resistance and correct values. Together, these results show that the microcontroller system can be used in laboratory applications to evaluate the resistivity of materials.

7 Conclusions and Recommendations for Future Work

In this thesis, we have presented four different microcontroller-based systems and have shown that they work and can be useful for research applications. We developed a data logging system for an EM-31 electrical conductivity instrument that automatically records electrical conductivity data that is referenced with GPS location and time. In addition, we created a water quality monitoring system that can be used to spatially map various water quality properties. We also developed a seismic system capable of capturing seismic signals in the field and recording them in real time with sufficient accuracy and quality to allow for meaningful comparison between systems. Finally, we constructed a laboratory-scale ER system that can be used to assess the resistivity of real materials. Each of these systems works reliably and meets the goals of their design, however, they all have room for improvement.

While the EM-31 microcontroller system works well enough to meet the function it is meant to perform, there are a few ways that it could be improved in the future. The biggest issue with the EM-31 microcontroller system is in how the microcontroller system responds when the data recorder button is pressed. Because pressing this button does not directly cause a response in the recorder trigger pin, instead causing a jump in the voltage being measured to determine electrical conductivity, it was difficult to design the microcontroller system so it could detect when the button is pressed. Currently, the microcontroller system checks for a jump in voltage between automatic measurements, which means that the user has to time the button press correctly for it to be detected. In addition, if the button is pressed at an inopportune time, it could cause the automatic measurements to be altered and incorrect values of electrical conductivity to be recorded. Development of some type of interrupt-based detection method that could pause automatic measurements whenever the button is pressed would greatly improve the button's

functionality. If this was done it would also allow a faster sampling rate to be used, since the current sampling rate is limited by the time necessary between measurements for the user to be able to time the button press and the microcontroller system to be able to detect it. Another, more minor, issue with the microcontroller system is that the GPS location tends to fluctuate even if the microcontroller system is left in place. Future improvements could consider different GPS options such as a higher quality GPS or a GPS that can be used with an antenna to try to improve the mapping accuracy.

For the water quality microcontroller system, future work could include making improvements to the nitrate, chloride, and turbidity sensors. For the nitrate and chloride sensors, the microcontroller system would be greatly improved if the sensors' sensitivity to temperature was accounted for so that the values are more accurate. Possible solutions may include altering the code to apply a four-point calibration that accounts for temperature and using the temperature values from the thermocouple when taking readings or looking for alternative sensors that are less sensitive to temperature or compensate for temperature changes on their own. In addition, the nitrate and chloride probes selected were not well suited for field use. Future work should include evaluating different sensor options for measuring these properties. For the turbidity sensor, the value of the sensor was greatly diminished by the use of filters to prevent the flow through cell from getting clogged and other sensors from getting damaged. Therefore, another great improvement that could be made would be to find a turbidity sensor that can be employed outside of the flow through cell. In addition, while the precision of the thermocouple was sufficient to detect changes in temperature, the accuracy of the values remained questionable. Future work should address this issue, either by considering different calibration techniques and

sensors or using a more accurate sensor with a slower response time to obtain an accurate baseline temperature.

For the seismic microcontroller system, the biggest limitation is how the sampling rate is limited by the available memory of the microcontroller. Future improvements could focus on either selecting a different microcontroller with more available memory or improving the efficiency of the code (e.g., using smart interrupts) to make better use of the existing memory. As it is, the code is only capable of performing steps sequentially. For example, when recording seismic data, the code must first store a given amount of data, then once those data are stored save it to the microSD card. Optimizing the code so that the microcontroller system can perform actions when not busy with something else could improve the efficiency of the microcontroller system and allow either a greater amount of data to be stored without gaps needed for saving data or a higher sampling rate. Another more minor issue is in the quality of the functions used to pair GPS data to PPS times and how GPS data are saved. Currently, the microcontroller system begins pairing GPS data to PPS times once 150 sets of GPS data have been stored. However, if the function that pairs the GPS data to PPS times cannot find a match, those GPS data are simply discarded. Sometimes, this can result in a lot of GPS data being thrown out and less than the desired two minutes' worth of GPS data being saved. Improvements to the function used to pair GPS data to PPS times and checks that better ensure an appropriate amount of GPS data are saved could improve the microcontroller system by making the timing accuracy more reliable when used long term.

In addition to software improvements, the seismic microcontroller system could also be greatly improved by the addition of an amplifier. While the microcontroller system functioned well during field testing, the signals generated only 4 m away had relatively small amplitudes.

Adding an amplifier to the microcontroller system would allow the microcontroller system to detect signals generated at farther distances, allowing it to be used for longer or larger seismic arrays. Future work should also include the addition of an LED indicator that data are being saved to the microSD card. Since one LED is blinked continuously as data are stored in storage arrays, a second LED could be designed to remain on as long as data are successfully being saved to the microSD card and to turn off if there is ever an error with saving the data. Together, both LEDs would indicate that the system is still running and that data are continuing to be saved. Finally, the microcontroller system currently consists of the components and wire connections mounted on breadboards. This would leave the components exposed to potential damage when deployed in the field. Before implementing the microcontroller system, it should be transferred to a more permanent enclosure, potentially with some of the following features: secure, permanent wired connections between components, a water-proof container with a built-in power source or space for batteries, geophones that are permanently attached and included in the container, spikes to allow for coupling with the ground, and a level to ensure the sensors are properly aligned.

For the electrical resistivity microcontroller system, the next steps would be altering the microcontroller system to a more permanent, usable state and adaptation for field use. Currently, the microcontroller system exists as an exposed collection of wires attached to a breadboard. To increase its usability, the microcontroller system should be moved to some type of enclosure, the connections should be made more permanent and secure, and an improved method of connecting the microcontroller system to different electrodes should be added. One consideration that future design will need to account for is how best to incorporate the external signal generator into the microcontroller system. In order for the microcontroller system to be used in the field, it would

need to be adapted to handle much larger currents and voltages. This could either include the addition of components capable of measuring large currents and voltages or a means of altering large currents and voltages to levels that the microcontroller system can currently handle. Both of these changes would make the microcontroller system easier to use and usable in a larger variety of applications. In addition, since the variations in measured resistances of the Dipole-Dipole array at large electrode separations indicate an insufficient amount of current is reaching the potential electrodes, future work should consider using a smaller resistor to measure the current for laboratory applications. The selection of an appropriately sized resistor should be based on finding a balance between using a large enough resistor that the potential drop can be measured by the resolution of the microcontroller system while minimizing the loss of current across the resistor.

While this thesis has discussed the details of four specific applications of microcontroller-based systems, the full range of applications that microcontroller-based systems can be applied to is in no way limited to these four. To this end, there are limitless possibilities for future work consisting of the development of new microcontroller-based systems.

References

Adafruit. (2019). *Fritzing-Library*. https://github.com/adafruit/Fritzing-Library

American Society for Testing and Materials. (1981). *Manual on the Use of Thermocouples in Temperature Measurement* (ASTM Publication Code Number 04-470020-40).

Analog Devices. (2010). *Small, Low Power, 3-Axis ±3 g Accelerometer*.

Analog Devices. (2011). *Precision Thermocouple Amplifiers with Cold Junction Compensation*.

Arduino. (2019a). *Arduino Mega 2560 Rev3*. http://store.arduino.cc/usa/arduino-mega-2560-rev3

Arduino. (2019b). *Arduino Uno Rev3*. http://store.arduino.cc/usa/arduino-uno-rev3

AtlasScientific. (2017a). *Dissolved Oxygen EZO™ Circuit*. (Version 3.7).

AtlasScientific. (2017b). *Dissolved Oxygen Probe*. (Version 2.5).

AtlasScientific. (2017c). *Conductivity EZO™ Circuit*. (Version 4.2).

AtlasScientific. (2017d). *Conductivity Probe K 1.0*. (Version 2.5).

AtlasScientific. (2017e). *pH EZO™ Circuit*. (Version 4.2).

AtlasScientific. (2017f). *pH Probe*. (Version 2.5).

Bolanakis, D. E. (2018). *Microcontroller Education: Do it Yourself, Reinvent the Wheel, Code to Learn*. San Rafael, CA: Morgan & Claypool Publishers LLC.

Bruneau, R. (2019). *Fritzing Parts – Sixth Set*. http://omnigatherum.ca/wp/?p=555

Cammann, K. (1979). *Working with Ion-Selective Electrodes: Chemical Laboratory Practice*. Berlin, Germany: Springer-Verlag.

Davies, J. H. (2008). *MSP430 Microcontroller Basics*. Burlington, MA: Newnes/Elsevier.

DFRobot. (2019). *Turbidity sensor SKU: SEN0189*. http://www.dfrobot.com/wiki/index.php/Turbidity_sensor_SKU:_SEN0189

Down, R. D. and Lehr, J. H. (2005). *Environmental Instrumentation and Analysis Handbook*. Hoboken, NJ: John Wiley & Sons, Inc.

Freiser, H. (1978). *Ion-Selective Electrodes in Analytical Chemistry*. New York, NY: Springer.

Fitterman, D. V. and Labson, V. F. (2005). Electromagnetic Induction Methods for Environmental Problems. In D. K. Butler (Ed.), *Near-Surface Geophysics* (pp. 301-355). Tulsa, OK: Society of Exploration Geophysicists.

Geddes, L., Kunihiro, K., and Turner, E. (2014). *Simplified Procedures for Water Examinations: Manual of Water Supply Practices*. Denver, CO: American Water Works Association.

Geonics Limited. (1991). *EM31: Operating Manual*.

George, B., Roy, J. K., Kumar, V. J., and Mukhopadhyay, S. C. (Eds.). (2017). *Advanced Interfacing Techniques for Sensors: Measurement Circuits and Systems for Intelligent Sensors*. Cham, Switzerland: Springer International Publishing.

Lakowicz, J. R. (Ed.) (1992). *Topics in Fluorescence Spectroscopy: Principles*. New York, NY: Kluwer Academic Publishers.

Morf, W. E. (1981). *The Principles of Ion-Selective Electrodes and of Membrane Transport*. Hungary: Elsevier.

Mori, Y. (2017). *Mechanical Vibrations: Applications to Equipment*. Hoboken, NJ: ISTE Ltd and John Wiley & Sons, Inc.

Pollock, D. D. (1971). *The Theory and Properties of Thermocouple Elements* (ASTM Special Technical Publication 492). Baltimore, MD: American Society for Testing and Materials.

PJRC. (2019). *Pin Assignments*. https://www.pjrc.com/teensy/pinout.html

Radiometer Analytical SAS. (2004). *Conductivity Theory and Practice*.

Regtien, P. P. L., van der Heijden, F., Korsten, M. J., and Olthius, W. (2004). *Measurement Science for Engineers*. Sterling, VA: Kogan Page Science.

Ripka, P. and Tipek, A. (Eds.). (2007). *Modern Sensors Handbook.* Newport Beach, CA: ISTE Ltd.

Rundle, C. C. (2000). *A Beginners Guide to Ion-Selective Electrode Measurements*. http://www.nico2000.net/Book/Guide1.html

Sharma, P. V. (2002). *Environmental and Engineering Geophysics*. Cambridge, UK: Cambridge University Press.

Skymoo. (2019). *Fritzing custom part for Teensy 3.6*. http://forum.fritzing.org/t/fritzing-custom-part-for-teensy-3-6/2781

SparkFun Electronics. (2019a). *ADXL345 Hookup Guide*. https://learn.sparkfun.com/tutorials/adxl345-hookup-guide

SparkFun Electronics. (2019b). *Thermocouple Type-K – Glass Brain Insulated (Bare Wire)*. (SEN-00251 ROHS). https://www.sparkfun.com/products/251

SparkFun Electronics. (2019c). *Teensy 3.6*. (DEV-14057 ROHS). http://www.sparkfun.com/products/14057

Streiff, C. and Hart, D. (2017). *Geophysical Surveys at Haskell Lake*. Wisconsin Geological and Natural History Survey: unpublished document.

Telford, W. M., Geldart, L. P., and Sheriff, R.E. (1990). *Applied Geophysics*. New York, NY: Cambridge University Press.

Vernier. (2017). *Chloride Ion-Selective Electrode*.

Vernier. (2014). *Nitrate Ion-Selective Electrode*.

Wilson, J. S. (2005). *Sensor Technology Handbook*. Burlington, MA: Newnes/Elsevier.

Yoon, J. -Y. (2016). *Introduction to Biosensors: From Electric Circuits to Immunosensors* (2nd ed.). Cham, Switzerland: Springer International Publishing.

YSI. (2009). *The Dissolved Oxygen Handbook: A Practical Guide to Dissolved Oxygen Measurements*. YSI Incorporated.

Zhang, D. and Wei, B. (Eds.). (2017). *Advanced Mechatronics and MEMS Devices II*. Switzerland: Springer.

Zhang, X., Ju, H., and Wang, J. (2008). *Electrochemical Sensors, Biosensors and their Biomedical Applications*. Amsterdam: Academic Press.

Zonge, K., Wynn, J., and Urquhart, S. (2005). Resistivity, Induced Polarization, and Complex Resistivity. In D. K. Butler (Ed.), *Near-Surface Geophysics* (pp. 265-300). Tulsa, OK: Society of Exploration Geophysicists.

Appendix A: Description of EM-31 Code

Included at the top of the code are four libraries which are necessary for using the GPS and microSD card. These include the TinyGPS++ and built-in SoftwareSerial libraries for the GPS and the built-in SPI and SD libraries for the microSD card. In addition, the column headers for the data file and the sampling rate are also defined at the beginning of the code. We set the sampling rate to collect measurements with a sampling interval of two seconds. Sampling any faster than this makes it difficult to hit the button so that the system can detect it between measurements.

The code begins by executing a setup phase once upon startup. In this phase, all the necessary communications and pins are set up and a file is created to store the data. First, communication is initialized with the GPS and microSD card. Then, the digital pins connected to pins D, E, and F of the recorder connector are set as inputs so the state of the pins can be read. Next, the digital pins used to control the LEDs are set as outputs so they can be used to turn the LEDs on and off. Once the LED pins are set up, the pin corresponding to the red power LED is set to high to turn on the LED and indicate that the system has power. Finally, the system creates and names a new file to store the data.

To name a new file, the system uses GPS data to create a name with the following format: DDHHMMSS.csv. To do this, the system must first wait for the GPS to obtain a signal before it can pull the time information to name the file. First, until the GPS location data are updated, the system continuously checks if new data are available from the GPS and reads and parses any available data using the functions of the TinyGPS++ library. Then, to ensure the time data are valid, the system waits until the day is not equal to 0 (it's default value) and once again

continuously checks if new data are available from the GPS and reads and parses any available data while waiting. Once the GPS data are valid, a function is run to name the file.

The function that names the file begins by clearing the variable that holds the file name so the new file name can be constructed. Then it pulls the day, hour, minute, and second from the GPS and uses this data to create the file name with the DDHHMMSS.csv format. Finally, this function runs another function which cycles through the column headers defined at the beginning of the file and prints them to the newly created file. Now the file is ready to begin storing data and the setup phase is complete.

Next the code enters the loop phase, a phase which continues to repeat endlessly and during which data are collected and stored automatically. First, the code checks if the button has been pressed. To do this, the code reads the voltage across pins A and B of the recorder connector and checks if the voltage is above 2 V. Through experimentation it was determined that during normal operation, the voltage across pins A and B (which corresponds to the electrical conductivity) does not exceed 2 V, however, when the button is pressed and pin G is hooked up to a digital pin, the voltage across pins A and B briefly spikes above 2 V. If the button was pressed, the code sets a variable to indicate that the button was pressed, waits two seconds to make sure the voltage has restabilized, then, if GPS data are available, runs the function to log the data. In the case where the button has been pressed, the code checks if GPS is available in case the specific location at which the button was pressed is important. For the automatic measurements, data are logged regardless of whether GPS data are available and valid. If the function indicates that data were successfully logged, the code blinks the green LED to indicate that data were logged with the button indicator. Finally, the variable to indicate that the button was pressed is reset.

If the code determines that the button was not pressed, the code uses the time of the last log and the sampling rate to determine if it is time to take another automatic measurement, which it does whenever two seconds has passed since the last automatic measurement was recorded. If it is time to take a new measurement, the code runs the same function to log the data. If the function indicates that the data were successfully logged, the code then sets the time of the last data log to the current time and this time blinks the blue LED to indicate that automatic data were logged. At the very end of the loop phase the code checks if new GPS data are available, and if there are it reads and parses the new data using the TinyGPS++ functions. Once the loop phase is complete, the code starts from the beginning and checks once again if the button has been pressed.

To log the data, the same function is used regardless of whether or not the button has been pressed. First the code checks the state of pins D, E, and F which are used to indicate the range and reads the voltage across pins A and B of the recorder connector. The code then uses the state combinations from Table 3.1.1 to determine the range. Next, the code uses the range and voltage reading to calculate the electrical conductivity. Once the conductivity has been calculated, the code then begins logging data to the microSD card. First, the code checks if there is new GPS data available. If new data are available the code logs the longitude, latitude, altitude, date, time, and number of satellites. If no new data are available, the code skips these entries. Next, the code logs the range and electrical conductivity reading in mS/m. Finally, the code checks the variable indicating if the button was pressed, and if it was it logs a B next to the data to indicate that this measurement corresponds to the time and location where the button was pressed. The code then returns to the loop phase and indicates whether or not the data were successfully logged.

Appendix B: EM-31 Code

```
// Libraries for GPS
#include <TinyGPS++.h>
#include <SoftwareSerial.h>

// Libraries for SD Card
#include <SPI.h>
#include <SD.h>

// Set Up for GPS
TinyGPSPlus tinyGPS;                                    // Name for GPS
SoftwareSerial ssGPS(8, 9);                             // Initiate software serial
communication with GPS

char logFileName[13];                                   // Char to store the current file
name
#define LOG_COLUMN_COUNT 8                                  // Number of
columns of GPS and timing information
char * log_col_names[LOG_COLUMN_COUNT] = {                      // Names to
be printed at head of columns
  "longitude", "latitude", "altitude", "date", "time", "satellites", "range", "reading"
};
#define LOG_RATE 2000 ////////////////////!SAMPLING RATE!///////////////////////       // Define the
sampling rate
unsigned long l = 0;                                    // Set the time of the last log to 0
int j;                                                  // Define variable to indicate if the button
was pressed

void setup() {
  ssGPS.begin(9600);                                    // Start GPS communication
  SD.begin(10);                                         // Start micro SD communication
  pinMode(2, INPUT);                                    // Pin D - Indicates Range
(green wire)
  pinMode(3, INPUT);                                    // Pin E - Indicates Range (blue
wire)
  pinMode(4, INPUT);                                    // Pin F - Indicates Range (red
wire)
  pinMode(5, OUTPUT);                                     // Green LED - Auto Data
  pinMode(6, OUTPUT);                                     // Blue LED - Button Data
  pinMode(7, OUTPUT);                                     // Red LED - Power
  digitalWrite(7, HIGH);                                // Turn red LED on to indicate
there is power

  // Begin a New File and Print Header to the File
```

```
  while (!tinyGPS.location.isUpdated()) {                    // Wait for valid GPS data
before creating the file name
    while (ssGPS.available()) {                              // While waiting, if GPS
communication is available
      tinyGPS.encode(ssGPS.read());                          // Read GPS data from the
GPS
    }
  }
  while (tinyGPS.date.day() == 0) {                          // Make sure the day is not
zero (the first time is sometimes is)
    while (ssGPS.available()) {                              // While waiting, if GPS
communication is available
      tinyGPS.encode(ssGPS.read());                          // Read GPS data from the
GPS
    }
  }
  updateFileName();                                          // Create a new file
}

void loop() {
  int G = analogRead(A1);                                    // Read pin G - Used to see if
button was pressed (black wire with 2 silver bands)
  float G2 = G * (5.0 / 1023.0);                             // Convert reading to voltage
  if (G2 > 2) {                                              // If voltage is above 2, the button is
pressed
    j = 1;                                                   // Set button indicator to indicate button
was pressed
    delay(2000);                                             // Wait for voltage to stabilize
    if (tinyGPS.location.isUpdated()) {                      // If there is GPS data
      if (logGPSData()) {                                    // Log the data
        digitalWrite(6, HIGH);                               // Blink the blue light to indicate
data was logged
        delay(250);
        digitalWrite(6, LOW);
      }
    }
    j = 0;                                                   // Reset the button indicator
  }
  else if ((l + LOG_RATE) <= millis()) {                     // If the button was not
pressed log the data at the sampling rate, check if enough time has passed since last data was
logged
    if (logGPSData()) {                                      // Log the data
      l = millis();                                          // Update the time of the last data log
      digitalWrite(5, HIGH);                                 // Blink the green light to
indicate data was logged
      delay(250);
```

```
    digitalWrite(5, LOW);
   }
  }
  while (ssGPS.available())                                    // Continue to get data from
GPS
   tinyGPS.encode(ssGPS.read());
}

byte logGPSData() {                                            // Log the data
  int range = 0;                                              // Define a variable for the range
  int D = digitalRead(2);                                      // Check if pin D is high or low
  int E = digitalRead(3);                                      // Check if pin E is high or low
  int F = digitalRead(4);                                      // Check if pin F is high or low
  int A = analogRead(A1);                                      // Read pin A (white wire) to
get a conductivity reading
  float A2 = A * (5.0 / 1023.0);                               // Convert reading to a voltage
  if (D == 1) {                                               // Use the combination of highs and
lows for pind D, E, and F to determine the range (mS/m)
   if (E == 1) {
    range = 30;
   }
   else {
    if (F == 1) {
     range = 300;
    }
    else {
     range = 3;
    }
   }
  }
  else {
   if (E == 1) {
    if (F == 1) {
     range = 1000;
    }
    else {
     range = 10;
    }
   }
   else {
    range = 100;
   }
  }
  float A3 = A2 * (range / 0.5);                               // Convert the voltage to
conductivity (mS/m)
```

```
  File logFile = SD.open(logFileName, FILE_WRITE);                    // Open the file
  if (logFile) {                                                      // Log the GPS, range, and
conductivity data
    if (tinyGPS.location.isUpdated()) {                               // If GPS data is available,
log the GPS data
    logFile.print(tinyGPS.location.lng(), 6);                         // Log longitude in degrees
    logFile.print(',');
    logFile.print(tinyGPS.location.lat(), 6);                         // Log latitude in degrees
    logFile.print(',');
    logFile.print(tinyGPS.altitude.meters(), 1);                      // Log altitude in meters
    logFile.print(',');
    logFile.print(tinyGPS.date.value());                              // Log date in DDMMYY
format
    logFile.print(',');
    logFile.print(tinyGPS.time.value());                              // Log time in HHMMSSCC
format
    logFile.print(',');
    logFile.print(tinyGPS.satellites.value());                        // Log the number of
satellites
    logFile.print(',');
    }
    else{                                                             // Otherwise log nothing for each value
missed
      logFile.print(",,,,,,")
    }
    logFile.print(range);                                             // Log the range in mS/m
    logFile.print(',');
    logFile.print(A3);                                                // Log the conductivity in mS/m
    if (j == 1) {                                                     // Indicate if the button was pressed
      logFile.print(',');
      logFile.print("B");
    }
    logFile.println();
    logFile.close();                                                  // Close the file
    return 1;                                                         // Indicate data was successfully
logged
  }
  return 0;                                                           // Otherwise, indicate data was not
logged
}

void printHeader() {                                                  // Print the column headers
  File logFile = SD.open(logFileName, FILE_WRITE);                    // Open the file
  if (logFile) {
    int i = 0;
```

```
   for (; i < LOG_COLUMN_COUNT; i++) {                                  // Print each of the
column headers
     logFile.print(log_col_names[i]);
     if (i < LOG_COLUMN_COUNT - 1)
       logFile.print(',');                                            // With a comma between each
     else
       logFile.println();
   }
   logFile.close();                                                   // Close the file
  }
}

void updateFileName() {                                               // Name the file
   memset(logFileName, 0, strlen(logFileName));                       // Clear the char used
to store the file name
   //String m = String(tinyGPS.date.month());
   String d = String(tinyGPS.date.day());                             // Get day, hour, minutes,
and seconds from the GPS
   String h = String(tinyGPS.time.hour());
   String mn = String(tinyGPS.time.minute());
   String s = String(tinyGPS.time.second());
   //  if (m.length() < 2)
   //  {
   //    m = "0" + m;
   //  }
   if (d.length() < 2)                                                // If day, hour, minutes, or seconds
has only one digit, add a 0 in front
   {
     d = "0" + d;
   }
   if (h.length() < 2)
   {
     h = "0" + h;
   }
   if (mn.length() < 2)
   {
     mn = "0" + mn;
   }
   if (s.length() < 2)
   {
     s = "0" + s;
   }
   String FileName = d + h + mn + s + ".csv";                         // Put file name together
in a string
   FileName.toCharArray(logFileName, 13);                             // Convert file name
string to char
```

```
    printHeader();                                  // Print the column headers
}
```

Appendix C: Description of Multi-Sensor Array Calibration Code

There are four libraries necessary to include for the calibration code. In order to use the microSD card, the built-in SD library is necessary. In order to perform the mathematical operations needed to calculate the nitrate and chloride concentrations based on voltage (calculating exponents and natural logs), the built-in math library is necessary. Finally, in order to use the ADS1115 16-bit ADC shield, the built-in Wire library and the ADS1015 library provided by Adafruit are necessary.

At the top of the code, in addition to the libraries, are several variables and definitions that make editing of the code easier. These definitions consist of pin definitions for the microSD card, chloride probe, nitrate probe, type J thermocouple, and type K thermocouple. Variables included here include those for controlling the log rate, strings for reading from files or reading from and writing to serials, variables for storing and converting parameters, and various indicators.

The code for calibration begins with a setup phase. In this phase, communication is initialized with the computer, electrical conductivity EZO™ circuit, pH EZO™ circuit, dissolved oxygen EZO™ circuit, ADS1115 16-bit ADC shield, and microSD card. If the microSD card was not successfully initialized, an error message is printed to the serial monitor to indicate that the system will not work. The gain of the ADS1115 16-bit ADC shield is set to two, which corresponds to a voltage range of 0 to 2.048 V and should cover a sufficient range of temperatures. Next, bytes are reserved for communication with the AtlasScientific EZO™ circuits. Then, the microSD card is checked to see if the necessary calibration files for the nitrate and chloride probes and the two thermocouples exist and if the calibration files exist the calibration parameters are calculated. Otherwise, if one or more calibration files are missing for a

sensor, a message is printed indicating that calibration is needed for that sensor. Finally, headers are printed to the serial monitor to indicate which column of data corresponds to which sensor.

Following the setup phase, the code begins to repeat the loop phase. In this phase, the code takes readings from the analog pins of the microcontroller and ADS1115 16-bit ADC shield and uses them to calculate the nitrate concentration, chloride concentration, and temperatures. Then, the code checks if it should print this new data to the serial monitor, which it does once five seconds have passed since the last time data were printed. If it is time to print new data to the serial monitor, the code prints the recently calculated nitrate concentration, chloride concentration, and temperatures, as well as any new electrical conductivity, pH, and dissolved oxygen data. If no new electrical conductivity, pH, and dissolved oxygen data are available, the code prints nothing for that value and continues to the next parameter. Finally, the code checks if there are any commands to send to the electrical conductivity, pH, or dissolved oxygen probes, and, if there are, sends them to the appropriate probes, clears the strings containing the commands so they are ready for the next command, and indicates that there is no longer a command waiting to be sent.

Between each loop phase, if new data are available at any of the serial communications a serialEvent function is run. There are four serialEvent functions (serialEvent, serialEvent1, serialEvent2, and serialEvent3) that each correspond to one of the devices communicating through serial communication (1 – computer, 2 – electrical conductivity, 3 – pH, and 4 – dissolved oxygen). The serialEvent1, serialEvent2, and serialEvent3 functions are used to read any available data from the electrical conductivity, pH, and dissolved oxygen probes, respectively, and indicate that data are available to be logged. These functions also check that the

incoming string of data contains the appropriate number of characters, and if not, rejects the data as an error (resets the indicator so it does not indicate that there is data available to be logged).

The serialEvent function is used to control several different actions when specific commands are detected, as discussed in Chapter 4.3.1. First, the incoming string is read and then it is compared to several different strings to determine if a command was sent. The code first checks if the string matches a command to change the default communication to the electrical conductivity, pH, or dissolved oxygen EZO™ circuits, and if so, the default communication is changed to that sensor and a message is printed indicating that communication is now set to that sensor. Then, the code checks if the string matches a command to run one of the four calibration functions and, if so, a message is printed indicating that calibration is beginning for that sensor and the appropriate calibration function is run. Lastly, if no matching command was found, the string is set as a command to send to the appropriate sensor and an indicator is used to indicate that a command is waiting to be sent. Then, at the end of the next loop phase, the command will be sent to the current default AtlasScientific EZO™ circuit. The serialEvent function runs until either the string is matched to a command or the string is set up to be sent to an AtlasScientific EZO™ circuit. If the string is matched to a command, the appropriate actions are taken and the serialEvent function is ended.

The last portion of the code is the calibration functions for the nitrate probe, chloride probe, type J thermocouple, and type K thermocouple. Each of these functions performs the same actions, but has minor differences such as different messages, file names, and equations used. The nitrate and chloride calibration functions begin by printing a message reminding the user that the probes must be soaked in the high standard solution for 30 minutes before use. Then they print a message prompting the user to enter the concentration of the high standard solution

used for calibration. This value is saved in the file for the high standard value and the high standard value is printed to the serial monitor. Next, the function prints a message prompting the user to send any key when ready to calibrate and, while waiting, prints the analog value corresponding to the parameter being calibrated every half a second so the user can determine when the readings have stabilized. Once any key is sent, the analog value is saved in the file for the high calibration value and printed to the serial monitor. This same process is then repeated for the low standard. Finally, the calibration parameters are calculated using the new calibration values and a message is printed indicating that calibration is complete.

The calibration functions for the thermocouples are nearly identical, however there is no initial reminder that the sensors need to be soaked (since they do not need to be) and these functions begin by calibrating the low temperature first. For the nitrate and chloride probes, the high standard must always be calibrated before the low standard, however, for the thermocouples the order does not matter, so the low temperature is chosen to be first arbitrarily. The code could be adjusted to calibrate the high temperature first, if preferred.

Appendix D: Multi-Sensor Array Calibration Code

```
// Library for SD Card
#include <SD.h>

// Library for Vernier Probes (Nitrate, Chloride, and Thermistor)
#include <math.h>

// Libraries for ADS1115 ADC (for Thermocouples)
#include <Adafruit_ADS1015.h>
#include <Wire.h>

// Pin Definitions
#define SD_PIN 53                                   // Digital pin 53
#define Chloride_PIN A0                             // Analog pin A0
#define Nitrate_PIN A1                              // Analog pin A1
#define TypeJ_PIN 0                                 // ADS1115 pin 0
#define TypeK_PIN 1                                 // ADS1115 pin 1

// Set Up for ADS1115 ADC (for Thermocouples)
Adafruit_ADS1115 ads1115;

String i = "";                                      // String for reading from serial
String f = "";                                      // String for reading from files
int id = 0;                                         // ID used to switch communication between EC,
pH and DO
int lr = 1000;                                      // Log rate in ms
unsigned long ll = 0;                               // Time of last log

// Variables for Nitrate
int n, nvh, nvl;                                    // n = analog value of nitrate, nvh = analog value
of high standard, nvl = analog value of low standard
float nh, nl, nvhmV, nvlmV, nEo, nm;                // Variables for Nitrate Calibration, nh
= high standard concentration (mg/L), nl = low standard concentration (mg/L), nvhmV and
nvlmV = analog values of standard converted to mV, nEo and nm = calibration parameters
float nV, nmV, nC;                                  // Variables for Nitrate Readings, nV = n
converted to V, nmV = nV converted to mV, nC = nmV converted to mg/L

// Variables for Chloride
int c, cvh, cvl;                                    // c = analog value of chloride, cvh = analog value
of high standard, nvl = analog value of low standard
float ch, cl, cvhmV, cvlmV, cEo, cm;                // Variables for Chloride Calibration, ch
= high standard concentration (mg/L), cl = low standard concentration (mg/L), cvhmV and
cvlmV = analog values of standards converted to mV, cEo and cm = calibration parameters
float cV, cmV, cC;                                  // Variables for Chloride Readings, cV = c
converted to V, cmV = cV converted to mV, cC = cmV converted to mg/L
```

```
// Variables for Electrical Conductivity Communication
String iEC = "";                                    // String to hold strings sent from serial to EC
String sEC = "";                                    // String to hold strings sent from EC to be read
boolean iEC_complete = false;                       // Indicator for if string is ready to be sent
to EC
boolean sEC_complete = false;                       // Indicator for if EC string is ready to be
read

// Variables for pH Communication
String ipH = "";                                    // String to hold strings sent from serial to pH
String spH = "";                                    // String to hold strings sent from pH to be read
boolean ipH_complete = false;                       // Indicator for if string is ready to be sent
to pH
boolean spH_complete = false;                       // Indicator for if pH string is ready to be
read

// Variables for Dissolved Oxygen Communication
String iDO = "";                                    // String to hold strings sent from serial to DO
String sDO = "";                                    // String to hold strings sent from DO to be read
boolean iDO_complete = false;                       // Indicator for if string is ready to be sent
to DO
boolean sDO_complete = false;                       // Indicator for if DO string is ready to be
read

// Variables for Type J Thermocouple
int16_t t_J;                                        // t_J = analog value of temperature
float tl_J, th_J, tvl_J, tvh_J, tm_J, tb_J, tC_J;   // tl_J = low calibration temperature,
th_J = high calibration temperature, tvl_J = analog value of low calibration temperature, tvh_J =
analog value of high calibration temperature, tm_J = slope of linear calibration, tb_J = intercept
of linear calibration, tC_J = temperature reading in deg. C

// Variables for Type K Thermocouple
int16_t t_K;                                        // t_K = analog value of temperature
float tl_K, th_K, tvl_K, tvh_K, tm_K, tb_K, tC_K;   // tl_K = low calibration
temperature, th_K = high calibration temperature, tvl_K = analog value of low calibration
temperature, tvh_K = analog value of high calibration temperature, tm_K = slope of linear
calibration, tb_K = intercept of linear calibration, tC_K = temperature reading in deg. C

void setup() {
  // Begin Communications
  Serial.begin(9600);                               // Begin communication with computer
  Serial1.begin(9600);                              // Begin communication with EC (Rx - 19; Tx
- 18);
  Serial2.begin(9600);                              // Begin communication with pH (Rx - 17; Tx
- 16);
```

```
  Serial3.begin(9600);                                  // Begin communication with DO (Rx - 15; Tx
- 14);
  ads1115.begin();                                     // Begin communication with ADS1115 ADC
(for Thermocouples)
  if (!SD.begin(SD_PIN)) {                              // Indicate if micro SD card was
successfully initiated
    Serial.println("SD Error");
  }
  else {
    Serial.println("SD Ready");
  }

  // ADS1115 Settings
  ads1115.setGain(GAIN_TWO);                            // Sets gain to 2 (+/- 2.048 V)

  // Reserve Bytes for Communication with EC, pH, and DO
  iEC.reserve(10);
  sEC.reserve(30);
  ipH.reserve(10);
  spH.reserve(30);
  iDO.reserve(10);
  sDO.reserve(30);

  // If Calibration Data Exists, Calculate Calibration Parameters
  if (SD.exists("nvh.csv") && SD.exists("nvl.csv") && SD.exists("nh.csv") &&
SD.exists("nl.csv")) { // Check if all nitrate calibration files exist, and if they do
    File f_nvh = SD.open("nvh.csv");                    // Open the calibration file for the high
standard nitrate calibration value
    if (f_nvh) {                                       // If the file was opened
      f = f_nvh.readStringUntil(13);                   // Get the high standard calibration value
for nitrate
      nvh = f.toFloat();                               // Convert that value to a number
      f_nvh.close();                                   // Close the file
    }
    File f_nvl = SD.open("nvl.csv");                    // Open the calibration file for the low
standard nitrate calibration value
    if (f_nvl) {                                       // If the file was opened
      f = f_nvl.readStringUntil(13);                   // Get the low standard calibration value for
nitrate
      nvl = f.toFloat();                               // Convert that value to a number
      f_nvl.close();                                   // Close the file
    }
    File f_nh = SD.open("nh.csv");                      // Open the calibration file for the high
standard nitrate concentration
    if (f_nh) {                                        // If the file was opened
```

```
    f = f_nh.readStringUntil(13);                    // Get the high standard concentration for
nitrate
    nh = f.toFloat();                                // Convert that value to a number
    f_nh.close();                                    // Close the file
   }
   File f_nl = SD.open("nl.csv");                    // Open the calibration file for the low
standard nitrate concentration
   if (f_nl) {                                       // If the file was opened
    f = f_nl.readStringUntil(13);                    // Get the low standard concentration for
nitrate
    nl = f.toFloat();                                // Convert that value to a number
    f_nl.close();                                    // Close the file
   }
   nvhmV = 137.55 * (nvh / 1023.0 * 5.0) - 0.1682;   // Using the nitrate calibration
values, calculate the nitrate calibration parameters
   nvlmV = 137.55 * (nvl / 1023.0 * 5.0) - 0.1682;   // These equations come from the
manual for the vernier nitrate probe
   nm = (nvhmV - nvlmV) / log(nh / nl);
   nEo = nvlmV - (nm * log(nl));
  }
  else {                                             // If not all nitrate calibration files exists, indicate
that nitrate needs calibration
   Serial.println("Need Nitrate Calibration");
  }
  if (SD.exists("cvh.csv") && SD.exists("cvl.csv") && SD.exists("ch.csv") &&
SD.exists("cl.csv")) { // Check if all chloride calibration files exists, and if they do,
   File f_cvh = SD.open("cvh.csv");                  // Open the calibration file for the high
standard chloride calibration value
   if (f_cvh) {                                      // If the file was opened
    f = f_cvh.readStringUntil(13);                   // Get the high standard calibration value
for chloride
    cvh = f.toFloat();                               // Convert that value to a number
    f_cvh.close();                                   // Close the file
   }
   File f_cvl = SD.open("cvl.csv");                  // Open the calibration file for the low
standard chloride calibration value
   if (f_cvl) {                                      // If the file was opened
    f = f_cvl.readStringUntil(13);                   // Get the low standard calibration value for
chloride
    cvl = f.toFloat();                               // Convert that value to a number
    f_cvl.close();                                   // Close the file
   }
   File f_ch = SD.open("ch.csv");                    // Open the calibration file for the high
standard chloride concentration
   if (f_ch) {                                       // If the file was opened
```

```
    f = f_ch.readStringUntil(13);                    // Get the high standard concentration for
chloride
    ch = f.toFloat();                                // Convert that value to a number
    f_ch.close();                                    // Close the file
   }
   File f_cl = SD.open("cl.csv");                    // Open the calibration file for the low
standard chloride concentration
   if (f_cl) {                                       // If the file was opened
    f = f_cl.readStringUntil(13);                    // Get the low standard concentration for
chloride
    cl = f.toFloat();                                // Convert that value to a number
    f_cl.close();                                    // Close the file
   }
   cvhmV = 137.55 * (cvh / 1023.0 * 5.0) - 0.1682;   // Using the chloride calibration
values, calculate the chloride calibration parameters
   cvlmV = 137.55 * (cvl / 1023.0 * 5.0) - 0.1682;   // These equations come from the
manual for the vernier chloride probe
   cm = (cvhmV - cvlmV) / log(ch / cl);
   cEo = cvlmV - (cm * log(cl));
  }
  else {                                             // If not all chloride files exist, indicate that chloride
needs calibration
   Serial.println("Need Chloride Calibration");
  }
  if (SD.exists("tvl_J.csv") && SD.exists("tvh_J.csv") && SD.exists("tl_J.csv") &&
SD.exists("th_J.csv")) { // Check if all type J thermocouple calibration files exists, and if they do,
   File f_tvl_J = SD.open("tvl_J.csv");              // Open the calibration file for the lower
temperature calibration value
   if (f_tvl_J) {                                    // If the file was opened
    f = f_tvl_J.readStringUntil(13);                 // Get the lower temperature calibration
value
    tvl_J = f.toFloat();                             // Convert that value to a number
    f_tvl_J.close();                                 // Close the file
   }
   File f_tvh_J = SD.open("tvh_J.csv");              // Open the calibration file for the
higher temperature calibration value
   if (f_tvh_J) {                                    // If the file was opened
    f = f_tvh_J.readStringUntil(13);                 // Get the higher temperature calibration
value
    tvh_J = f.toFloat();                             // Convert that value to a number
    f_tvh_J.close();                                 // Close the file
   }
   File f_tl_J = SD.open("tl_J.csv");                // Open the calibration file for the lower
temperature value
   if (f_tl_J) {                                     // If the file was opened
    f = f_tl_J.readStringUntil(13);                  // Get the lower temperature value
```

```
    tl_J = f.toFloat();                          // Convert that value to a number
    f_tl_J.close();                              // Close the file
   }
   File f_th_J = SD.open("th_J.csv");                    // Open the calibration file for the higher
temperature value
   if (f_th_J) {                                // If the file was opened
    f = f_th_J.readStringUntil(13);                      // Get the higher temperature value
    th_J = f.toFloat();                          // Convert that value to a number
    f_th_J.close();                              // Close the file
   }
   tm_J = (th_J - tl_J) / (tvh_J - tvl_J);               // Using the temperature calibration and
temperature values, calculate the type J thermocouple calibration parameters
   tb_J = th_J - (tm_J * tvh_J);                         // These equations assume a linear
relationship
  }
  else {                                        // If not all type J thermocouple files exist, indicate
that type J thermocouple needs calibration
   Serial.println("Need Type J Thermocouple Calibration");
  }
  if (SD.exists("tvl_K.csv") && SD.exists("tvh_K.csv") && SD.exists("tl_K.csv") &&
SD.exists("th_K.csv")) { // Check if all type K thermocouple calibration files exists, and if they
do,
   File f_tvl_K = SD.open("tvl_K.csv");                  // Open the calibration file for the
lower temperature calibration value
   if (f_tvl_K) {                               // If the file was opened
    f = f_tvl_K.readStringUntil(13);                     // Get the lower temperature calibration
value
    tvl_K = f.toFloat();                         // Convert that value to a number
    f_tvl_K.close();                             // Close the file
   }
   File f_tvh_K = SD.open("tvh_K.csv");                  // Open the calibration file for the
higher temperature calibration value
   if (f_tvh_K) {                               // If the file was opened
    f = f_tvh_K.readStringUntil(13);                     // Get the higher temperature calibration
value
    tvh_K = f.toFloat();                         // Convert that value to a number
    f_tvh_K.close();                             // Close the file
   }
   File f_tl_K = SD.open("tl_K.csv");                    // Open the calibration file for the lower
temperature value
   if (f_tl_K) {                                // If the file was opened
    f = f_tl_K.readStringUntil(13);                      // Get the lower temperature value
    tl_K = f.toFloat();                          // Convert that value to a number
    f_tl_K.close();                              // Close the file
   }
```

```
  File f_th_K = SD.open("th_K.csv");                    // Open the calibration file for the
higher temperature value
  if (f_th_K) {                                         // If the file was opened
    f = f_th_K.readStringUntil(13);                     // Get the higher temperature value
    th_K = f.toFloat();                                 // Convert that value to a number
    f_th_K.close();                                     // Close the file
  }
  tm_K = (th_K - tl_K) / (tvh_K - tvl_K);               // Using the temperature calibration
and temperature values, calculate the type K thermocouple calibration parameters
  tb_K = th_K - (tm_K * tvh_K);                         // These equations assume a linear
relationship
 }
 else {                                                 // If not all type K thermocouple files exist, indicate
the type K thermocouple needs calibration
  Serial.println("Need Type K Thermocouple Calibration");
 }

 // Print Headers
 Serial.print("EC");
 Serial.print('\t');
 Serial.print("pH");
 Serial.print('\t');
 Serial.print("DO");
 Serial.print('\t');
 Serial.print("NO3-");
 Serial.print('\t');
 Serial.print("Cl-");
 Serial.print('\t');
 Serial.print("Type J");
 Serial.print('\t');
 Serial.println("Type K");
}

void serialEvent() {                                    // Triggered if there is an incoming string sent
from the serial
 i = Serial.readStringUntil(13);                        // Read the incoming string from serial

 // Change Communication Between EC, pH and DO          // These are the commands to
tell the Arduino which sensor you want to send commands to, needed for EC, pH, and DO
 if (i == "EC") {                                       // If the string was "EC" change the id to 3 to
switch communication to EC
  id = 1;
  Serial.println("Communicating with EC");
 }
 else if (i == "pH") {                                  // If the string was "pH" change the id to 2 to
switch communication to pH
```

```
   id = 2;
   Serial.println("Communicating with pH");
  }
  else if (i == "DO") {                                  // If the string was "DO" change the id to 1 to
switch communication to DO
   id = 3;
   Serial.println("Communicating with DO");
  }

  // Begin Nitrate Calibration
  else if (i == "n cal") {                               // If the string was "n cal"
   Serial.println("Begin Nitrate Calibration");                  // Begin nitrate calibration
   Caln();                                             // Run the function for nitrate calibration
  }

  // Begin Chloride Calibration
  else if (i == "c cal") {                               // If the string was "c cal"
   Serial.println("Begin Chloride Calibration");                 // Begin chloride calibration
   Calc();                                             // Run the function for chloride calibration
  }

  // Begin Type J Thermocouple Calibration
  else if (i == "tJ cal") {                              // If the string was "tJ cal"
   Serial.println("Begin Type J Thermocouple Calibration");       // Begin type J thermocouple
calibration
   CaltJ();                                            // Run the function for type J thermocouple
calibration
  }

  // Begin Type K Thermocouple Calibration
  else if (i == "tK cal") {                              // If the string was "tK cal"
   Serial.println("Begin Type K Thermocouple Calibration");       // Begin type K thermocouple
calibration
   CaltK();                                            // Run the function for type K thermocouple
calibration
  }

  // If None of the Above, Send String to EC, pH, or DO Depending on Current ID
  else {
   if (id == 1) {                                       // If the current ID is 1
    iEC = i;                                           // Set the EC incoming string variable equal to the
incoming string, to be sent to EC
    iEC_complete = true;                                   // Indicate that there is a string to be sent to
EC
   }
   else if (id == 2) {                                  // If the current ID is 2
```

```
    ipH = i;                                    // Set the pH incoming string variable equal to the
incoming string, to be sent to pH
    ipH_complete = true;                        // Indicate that there is a string to be sent to
pH
   }
   else if (id == 3) {                          // If the current ID is 3
    iDO = i;                                    // Set the DO incoming string variable equal to the
incoming string, to be sent to DO
    iDO_complete = true;                        // Indicate that there is a string to be sent to
DO
   }
  }
}

// Get Data from EC
void serialEvent1() {                           // Triggered if there is an incoming string from
EC
  sEC = "";                                     // Clear the EC sensor string variable
  sEC = Serial1.readStringUntil(13);            // Store the incoming string in the EC
sensor string variable, to be sent to the serial and/or logged
  sEC_complete = true;                          // Indicate that there is a string to be sent to
the serial and/or data to be logged
  if (sEC.length() > 4) {                       // If there is an error and the data is too long
(two data points mashed together)
   sEC_complete = false;                        // Indicate that there is no data to be logged
instead
  }
}

// Get Data from pH
void serialEvent2() {                           // Triggered if there is an incoming string from
from pH
  spH = "";                                     // Clear the pH sensor string variable
  spH = Serial2.readStringUntil(13);            // Store the incoming string in the pH
sensor string variable, to be sent to the serial and/or logged
  spH_complete = true;                          // Indicate that there is a string to be sent to
the serial and/or data to be logged
  if (spH.length() > 6) {                       // If there is an error and the data is too long
(two data points mashed together)
   spH_complete = false;                        // Indicate that there is no data to be logged
instead
  }
}

// Get Data from DO
```

```
void serialEvent3() {                                   // Triggered if there is an incoming string from
DO
  sDO = "";                                            // Clear the DO sensor string variable
  sDO = Serial3.readStringUntil(13);                        // Store the incoming string in the DO
sensor string variable, to be sent to the serial and/or logged
  sDO_complete = true;                                  // Indicate that there is a string to be sent to
the serial and/or data to be logged
  if (sDO.length() > 5) {                               // If there is an error and the data is too long
(two data points mashed together)
    sDO_complete = false;                               // Indicate that there is no data to be logged
instead
  }
}

void loop() {
  // Get Nitrate Data
  n = analogRead(Nitrate_PIN);                          // Read the analog value from the nitrate
probe
  nV = n / 1023.0 * 5.0;                                // Convert that value to V using 10 bit
conversion from Arduino ADC
  nmV = 137.55 * nV - 0.1682;                           // Convert that value to mV using
equation given in the vernier nitrate probe manual
  double(nval) = ((nmV - nEo) / nm);                        // Using the equation given in the
vernier nitrate probe manual and the calibration parameters,
  nC = exp(nval);                                      // Calculate the concentration of nitrate in mg/L

  // Get Chloride Data
  c = analogRead(Chloride_PIN);                         // Read the analog value from the
chloride probe
  cV = c / 1023.0 * 5.0;                                // Convert that value to V using 10 bit
conversion from Arduino ADC
  cmV = 137.55 * cV - 0.1682;                           // Convert that value to mV using
equation given in the vernier nitrate probe manual
  double(cval) = ((cmV - cEo) / cm);                        // Using the equation given in the vernier
chloride probe manual and the calibration parameters,
  cC = exp(cval);                                      // Calculate the concentration of chloride in
mg/L

  // Get Type J Thermocouple Data
  t_J = ads1115.readADC_SingleEnded(TypeJ_PIN);             // Read the analog value from
the thermocouple
  tC_J = tm_J * t_J + tb_J;                             // Using the calibration parameters and an
assumed linear relationship, calculate temperature in deg. C from the average analog value

  // Get Type K Thermocouple Data
```

```
  t_K = ads1115.readADC_SingleEnded(TypeK_PIN);                  // Read the analog value
from the thermocouple
  tC_K = tm_K * t_K + tb_K;                                      // Using the calibration parameters and an
assumed linear relationship, calculate temperature in deg. C of the average analog value

  // Print Data to Serial
  if ((ll + lr) <= millis() && millis() >= 5000) {               // Check if it's time for next data point
    if (sEC_complete == true) {                                  // If there is a string to read from EC
      Serial.print(sEC);                                         // Log EC string/data
      Serial.print('\t');
      sEC_complete = false;                                      // Indicate string from EC was read
    }
    else {                                                       // If there is not a string to read from EC
      Serial.print('\t');                                        // Log nothing
    }
    if (spH_complete == true) {                                  // If there is a string to read from pH
      Serial.print(spH);                                         // Log pH string/data
      Serial.print('\t');
      spH_complete = false;                                      // Indicate string from pH was read
    }
    else {                                                       // If there is not a string to read from pH
      Serial.print('\t');                                        // Log nothing
    }
    if (sDO_complete == true) {                                  // If there is a string to read from DO
      Serial.print(sDO);                                         // Log DO string/data
      Serial.print('\t');
      sDO_complete = false;                                      // Indicate string from DO was read
    }
    else {                                                       // If there is not a string to read from DO
      Serial.print('\t');                                        // Log nothing
    }
    Serial.print(nC);                                            // Log the nitrate concentration in mg/L
    Serial.print('\t');
    Serial.print(cC);                                            // Log the chloride concentration in mg/L
    Serial.print('\t');
    Serial.print(tC_J);                                          // Log the type J thermocouple temperature in
deg. C
    Serial.print('\t');
    Serial.println(tC_K);                                        // Log the type K thermocouple temperature in
deg. C
    ll = millis();                                               // Save the time as the time of the last log
  }

  // Send String to EC
  if (iEC_complete == true) {                                    // If there is a string to send to EC
    Serial1.print(iEC);                                          // Send the string to EC
```

```
  Serial1.print('\r');                    // Send a return indicating string is complete
  iEC = "";                               // Clear the EC incoming string variable
  iEC_complete = false;                   // Indicate string was sent to EC
 }

 //Send String to pH
 if (ipH_complete == true) {              // If there is a string to send to pH
  Serial2.print(ipH);                     // Send the string to pH
  Serial2.print('\r');                    // Send a return indicating string is complete
  ipH = "";                               // Clear the pH incoming string variable
  ipH_complete = false;                   // Indicate string was sent to pH
 }

 // Send String to DO
 if (iDO_complete == true) {              // If there is a string to send to DO
  Serial3.print(iDO);                     // Send the string to DO
  Serial3.print('\r');                    // Send a return indicating string is complete
  iDO = "";                               // Clear the DO incoming string variable
  iDO_complete = false;                   // Indicate string was sent to DO
 }
}

// Nitrate Calibration
void Caln() {                             // Triggered when nitrate calibration fuction is
run
 Serial.println("Soak electrode in High Standard solution for 30 minutes");

 // Calibrate Higher Value
 Serial.println("Enter concentration of high standard solution (mg/L)");
 while (!Serial.available()) {            // Wait for a value
 }
 i = Serial.readStringUntil(13);          // Get new high standard value
 nh = i.toFloat();                        // Convert to a number
 if (SD.exists("nh.csv")) {               // If a file already exists,
  SD.remove("nh.csv");                    // delete the old file
 }
 File f_nh = SD.open("nh.csv", FILE_WRITE);          // Create and open a new file
 if (f_nh) {                              // If the file was opened
  f_nh.print(nh);                         // Save the new high standard value
  f_nh.close();                           // Close the file
  Serial.print("high standard: ");        // Print the high standard value to the serial
  Serial.println(nh);
 }
 Serial.println("Send any key when ready to calibrate high standard solution");
 while (!Serial.available()) {            // Wait for anything to be sent from serial
```

```
   n = analogRead(Nitrate_PIN);                    // While waiting, get the analog value
from nitrate
   Serial.println(n);                              // and print the analog value to the serial
   delay(500);                                     // every half second
  }
  i = Serial.readStringUntil(13);                  // Read the key that was sent to clear the
serial buffer (or it will just wait there to be read)
  nvh = analogRead(Nitrate_PIN);                   // Once a key has been pressed, get new
calibration value
  if (SD.exists("nvh.csv")) {                      // If a file already exists,
   SD.remove("nvh.csv");                           // delete the old file
  }
  File f_nvh = SD.open("nvh.csv", FILE_WRITE);     // Create and open a new file
  if (f_nvh) {                                     // If the file was opened
   f_nvh.print(nvh);                               // Save the new calibration value
   f_nvh.close();                                  // Close the file
   Serial.print("nvh = ");                         // Print the new calibration value to the serial
   Serial.println(nvh);
  }

  // Calibrate Lower value
  Serial.println("Enter concentration of low standard solution (mg/L)");
  while (!Serial.available()) {                    // Wait for a value
  }
  i = Serial.readStringUntil(13);                  // Get new low standard value
  nl = i.toFloat();                                // Convert to a number
  if (SD.exists("nl.csv")) {                       // If a file already exists,
   SD.remove("nl.csv");                            // delete the old file
  }
  File f_nl = SD.open("nl.csv", FILE_WRITE);       // Create and open a new file
  if (f_nl) {                                      // If the file was opened
   f_nl.print(nl);                                 // Save the new low standard value
   f_nl.close();                                   // Close the file
   Serial.print("low standard: ");                 // Print the low standard value to the serial
   Serial.println(nl);
  }
  Serial.println("Send any key when ready to calibrate low standard solution");
  while (!Serial.available()) {                    // Wait for anything to be sent from serial
   n = analogRead(Nitrate_PIN);                    // While waiting, get the analog value
from nitrate
   Serial.println(n);                              // and print the analog value to the serial
   delay(500);                                     // every half second
  }
  i = Serial.readStringUntil(13);                  // Read the key that was sent to clear the
serial buffer (or it will just wait there to be read)
```

```
  nvl = analogRead(Nitrate_PIN);                    // Once a key has been pressed, get new
calibration value
  if (SD.exists("nvl.csv")) {                       // If a file already exists,
    SD.remove("nvl.csv");                           // delete the old file
  }
  File f_nvl = SD.open("nvl.csv", FILE_WRITE);      // Create and open a new file
  if (f_nvl) {                                      // If the file was opened
    f_nvl.print(nvl);                               // Save the new calibration value
    f_nvl.close();                                  // Close the file
    Serial.print("nvl = ");                         // Print the new calibration value to the serial
    Serial.println(nvl);
  }

  // Calculate new calibration parameters
  nvhmV = 137.55 * (nvh / 1023.0 * 5.0) - 0.1682;
  nvlmV = 137.55 * (nvl / 1023.0 * 5.0) - 0.1682;
  nm = (nvhmV - nvlmV) / log(nh / nl);
  nEo = nvlmV - (nm * log(nl));

  Serial.println("Nitrate Calibration Done");
}

// Chloride Calibration
void Calc() {                                       // Triggered when chloride calibration function is
run
  Serial.println("Soak electrode in high standard solution for 30 minutes");

  // Calibrate Higher Value
  Serial.println("Enter concentration of high standard solution (mg/L)");
  while (!Serial.available()) {                     // Wait for a value
  }
  i = Serial.readStringUntil(13);                   // Get new high standard value
  ch = i.toFloat();                                 // Convert to a number
  if (SD.exists("ch.csv")) {                        // If a file already exists,
    SD.remove("ch.csv");                            // delete the old file
  }
  File f_ch = SD.open("ch.csv", FILE_WRITE);        // Create and open a new file
  if (f_ch) {                                       // If the file was opened
    f_ch.print(ch);                                 // Save the new high standard value
    f_ch.close();                                   // Close the file
    Serial.print("high standard: ");                // Print the high standard value to the serial
    Serial.println(ch);
  }
  Serial.println("Send any key when ready to calibrate high standard solution");
  while (!Serial.available()) {                     // Wait for anything to be sent from serial
```

```
   c = analogRead(Chloride_PIN);                    // While waiting, get the analog value
from chloride
   Serial.println(c);                               // and print the analog value to the serial
   delay(500);                                      // every half second
  }
  i = Serial.readStringUntil(13);                   // Read the key that was sent to clear the
serial buffer (or it will just wait there to be read)
  cvh = analogRead(Chloride_PIN);                   // Get new calibration value
  if (SD.exists("cvh.csv")) {                       // If a file already exists,
   SD.remove("cvh.csv");                            // delete the old file
  }
  File f_cvh = SD.open("cvh.csv", FILE_WRITE);      // Create and open a new file
  if (f_cvh) {                                      // If the file was opened
   f_cvh.print(cvh);                                // Save the new calibration value
   f_cvh.close();                                   // Close the file
   Serial.print("cvh = ");                          // Print the new calibration value to the serial
   Serial.println(cvh);
  }

  // Calibrate Lower Value
  Serial.println("Enter concentration of low standard solution (mg/L)");
  while (!Serial.available()) {                     // Wait for a value
  }
  i = Serial.readStringUntil(13);                   // Get new low standard value
  cl = i.toFloat();                                 // Convert to a number
  if (SD.exists("cl.csv")) {                        // If a file already exists,
   SD.remove("cl.csv");                             // delete the old file
  }
  File f_cl = SD.open("cl.csv", FILE_WRITE);        // Create and open a new file
  if (f_cl) {                                       // If the file was opened
   f_cl.print(cl);                                  // Save the new low standard value
   f_cl.close();                                    // Close the file
   Serial.print("low standard: ");                  // Print the low standard value to the serial
   Serial.println(cl);
  }
  Serial.println("Send any key when ready to calibrate low standard solution");
  while (!Serial.available()) {                     // Wait for anything to be sent from serial
   c = analogRead(Chloride_PIN);                    // While waiting, get the analog value
from chloride
   Serial.println(c);                               // and print the analog value to the serial
   delay(500);                                      // every half second
  }
  i = Serial.readStringUntil(13);                   // Read the key that was sent to clear the
serial buffer (or it will just wait there to be read)
  cvl = analogRead(Chloride_PIN);                   // Get new calibration value
  if (SD.exists("cvl.csv")) {                       // If a file already exists,
```

```
  SD.remove("cvl.csv");                                  // delete the old file
 }
 File f_cvl = SD.open("cvl.csv", FILE_WRITE);                  // Create and open a new file
 if (f_cvl) {                                          // If the file was opened
  f_cvl.print(cvl);                                    // Save the new calibration value
  f_cvl.close();                                       // Close the file
  Serial.print("cvl = ");                                  // Print the new calibration value to the serial
  Serial.println(cvl);
 }

 // Calculate new calibration parameters
 cvhmV = 137.55 * (cvh / 1023.0 * 5.0) - 0.1682;
 cvlmV = 137.55 * (cvl / 1023.0 * 5.0) - 0.1682;
 cm = (cvhmV - cvlmV) / log(ch / cl);
 cEo = cvlmV - (cm * log(cl));

 Serial.println("Chloride Calibration Done");
}

// Type J Thermocouple Calibration
void CaltJ() {                                         // Triggered when type J thermocouple
calibration function is run
 // Calibrate Lower Value
 Serial.println("Enter Low Temperature (deg. C)");
 while (!Serial.available()) {                               // Wait for a value
 }
 i = Serial.readStringUntil(13);                             // Get new temperature value
 tl_J = i.toFloat();                                   // Convert to a number
 if (SD.exists("tl_J.csv")) {                               // If a file already exists,
  SD.remove("tl_J.csv");                                 // delete the old file
 }
 File f_tl_J = SD.open("tl_J.csv", FILE_WRITE);                // Create and open a new file
 if (f_tl_J) {                                         // If the file was opened
  f_tl_J.print(tl_J);                                   // Save the new temperature value
  f_tl_J.close();                                      // Close the file
  Serial.print("low temp: ");                               // Print the low temperature value to the
serial
  Serial.println(tl_J);
 }
 Serial.println("Send any key when ready to calibrate low temperature");
 while (!Serial.available()) {                               // Wait for anything to be sent from serial
  t_J = ads1115.readADC_SingleEnded(TypeJ_PIN);              // While waiting, get the
analog value from type J thermocouple
  Serial.println(t_J);                                  // and print the anlog value to the serial
  delay(500);                                          // every half second
 }
```

```
  i = Serial.readStringUntil(13);                    // Read the key that was sent to clear the
serial buffer (or it will just wait there to be read)
  tvl_J = ads1115.readADC_SingleEnded(TypeJ_PIN);          // Get new calibration value
  if (SD.exists("tvl_J.csv")) {                      // If a file already exists,
    SD.remove("tvl_J.csv");                          // delete the old file
  }
  File f_tvl_J = SD.open("tvl_J.csv", FILE_WRITE);         // Create and open a new file
  if (f_tvl_J) {                                     // If the file was opened
    f_tvl_J.print(tvl_J);                            // Save the new calibration value
    f_tvl_J.close();                                 // Close the file
    Serial.print("tvl_J = ");                        // Print the new calibration value to the serial
    Serial.println(tvl_J);
  }

  // Calibrate Higher Value
  Serial.println("Enter High Temperature (deg. C)");
  while (!Serial.available()) {                      // Wait for a value
  }
  i = Serial.readStringUntil(13);                    // Get new temperature value
  th_J = i.toFloat();                                // Convert to a number
  if (SD.exists("th_J.csv")) {                       // If a file alrady exists,
    SD.remove("th_J.csv");                           // delete the old file
  }
  File f_th_J = SD.open("th_J.csv", FILE_WRITE);           // Create and open a new file
  if (f_th_J) {                                      // If the file was opened
    f_th_J.print(th_J);                              // Save the new temperature value
    f_th_J.close();                                  // Close the file
    Serial.print("high temp: ");                     // Print the high temperature value to the
serial
    Serial.println(th_J);
  }
  Serial.println("Send any key when ready to calibrate high temperature");
  while (!Serial.available()) {                      // Wait for anything to be sent from serial
    t_J = ads1115.readADC_SingleEnded(TypeJ_PIN);          // While waiting, get the anlog
value from type J thermocouple
    Serial.println(t_J);                             // and print the analog value to the serial
    delay(500);                                      // every half second
  }
  i = Serial.readStringUntil(13);                    // Read the key that was sent to clear the
serial buffer (or it will just wait there to be read)
  tvh_J = ads1115.readADC_SingleEnded(TypeJ_PIN);          // Get new calibration value
  if (SD.exists("tvh_J.csv")) {                      // If a file already exists,
    SD.remove("tvh_J.csv");                          // delete the old file
  }
  File f_tvh_J = SD.open("tvh_J.csv", FILE_WRITE);         // Create and open a new file
  if (f_tvh_J) {                                     // If the file was opened
```

```
    f_tvh_J.print(tvh_J);                    // Save the new calibration value
    f_tvh_J.close();                         // Close the file
    Serial.print("tvh_J = ");                // Print the new calibration value to the serial
    Serial.println(tvh_J);
  }

  // Calculate new calibration parameters
  tm_J = (th_J - tl_J) / (tvh_J - tvl_J);
  tb_J = th_J - (tm_J * tvh_J);

  Serial.println("Type J Thermocouple Calibration Done");
}

// Type K Thermocouple Calibration
void CaltK() {                               // Triggered when type K thermocouple
calibration function is run
  // Calibrate Lower Value
  Serial.println("Enter Low Temperature (deg. C)");
  while (!Serial.available()) {              // Wait for a value
  }
  i = Serial.readStringUntil(13);            // Get new temperature value
  tl_K = i.toFloat();                        // Convert to a number
  if (SD.exists("tl_K.csv")) {               // If a file already exists,
    SD.remove("tl_K.csv");                   // delete the old file
  }
  File f_tl_K = SD.open("tl_K.csv", FILE_WRITE);          // Create and open a new file
  if (f_tl_K) {                              // If the file was opened
    f_tl_K.print(tl_K);                      // Save the new temperature value
    f_tl_K.close();                          // Close the file
    Serial.print("low temp: ");              // Print the low temperature value to the
serial
    Serial.println(tl_K);
  }
  Serial.println("Send any key when ready to calibrate low temperature");
  while (!Serial.available()) {              // Wait for anything to be sent from serial
    t_K = ads1115.readADC_SingleEnded(TypeK_PIN);          // While waiting, get the
analog value from type K thermocouple
    Serial.println(t_K);                     // and print the anlog value to the serial
    delay(500);                              // every half second
  }
  i = Serial.readStringUntil(13);            // Read the key that was sent to clear the
serial buffer (or it will just wait there to be read)
  tvl_K = ads1115.readADC_SingleEnded(TypeK_PIN);         // Get new calibration value
  if (SD.exists("tvl_K.csv")) {              // If a file already exists,
    SD.remove("tvl_K.csv");                  // delete the old file
  }
```

```
  File f_tvl_K = SD.open("tvl_K.csv", FILE_WRITE);              // Create and open a new file
  if (f_tvl_K) {                                    // If the file was opened
    f_tvl_K.print(tvl_K);                           // Save the new calibration value
    f_tvl_K.close();                                // Close the file
    Serial.print("tvl_K = ");                       // Print the new calibration value to the serial
    Serial.println(tvl_K);
  }

  // Calibrate Higher Value
  Serial.println("Enter High Temperature (deg. C)");
  while (!Serial.available()) {                     // Wait for a value
  }
  i = Serial.readStringUntil(13);                   // Get new temperature value
  th_K = i.toFloat();                               // Convert to a number
  if (SD.exists("th_K.csv")) {                      // If a file alrady exists,
    SD.remove("th_K.csv");                          // delete the old file
  }
  File f_th_K = SD.open("th_K.csv", FILE_WRITE);              // Create and open a new file
  if (f_th_K) {                                     // If the file was opened
    f_th_K.print(th_K);                             // Save the new temperature value
    f_th_K.close();                                 // Close the file
    Serial.print("high temp: ");                    // Print the high temperature value to the
serial
    Serial.println(th_K);
  }
  Serial.println("Send any key when ready to calibrate high temperature");
  while (!Serial.available()) {                     // Wait for anything to be sent from serial
    t_K = ads1115.readADC_SingleEnded(TypeK_PIN);               // While waiting, get the
anlog value from type K thermocouple
    Serial.println(t_K);                            // and print the analog value to the serial
    delay(500);                                     // every half second
  }
  i = Serial.readStringUntil(13);                   // Read the key that was sent to clear the
serial buffer (or it will just wait there to be read)
  tvh_K = ads1115.readADC_SingleEnded(TypeK_PIN);               // Get new calibration value
  if (SD.exists("tvh_K.csv")) {                     // If a file already exists,
    SD.remove("tvh_K.csv");                         // delete the old file
  }
  File f_tvh_K = SD.open("tvh_K.csv", FILE_WRITE);              // Create and open a new file
  if (f_tvh_K) {                                    // If the file was opened
    f_tvh_K.print(tvh_K);                           // Save the new calibration value
    f_tvh_K.close();                                // Close the file
    Serial.print("tvh_K = ");                       // Print the new calibration value to the serial
    Serial.println(tvh_K);
  }
```

```
  // Calculate new calibration parameters
  tm_K = (th_K - tl_K) / (tvh_K - tvl_K);
  tb_K = th_K - (tm_K * tvh_K);

  Serial.println("Type K Thermocouple Calibration Done");
}
```

Appendix E: Description of Multi-Sensor Array Data Logging Code

This code requires the same four libraries as the calibration code (the built-in SD library for the microSD card, the built-in math library for the nitrate and chloride probes, and the built-in wire library and Adafruit's Adafruit_ADS1015 library for the ADS1115 16-bit ADC shield) plus two additional libraries for the GPS. These libraries are the built-in SoftwareSerial library to allow serial communication on pins other than the hardware serial pins and Adafruit's Adafruit_GPS library which handles communication with the GPS and parsing of the GPS data.

At the top of the code, in addition to the libraries, are several variables and definitions that make editing of the code easier. These definitions consist of pin definitions for the LEDs, microSD card, GPS, chloride probe, nitrate probe, type J thermocouple, type K thermocouple, and turbidity sensor. Variables included here include those for controlling the log rate, strings for reading from files or reading from and writing to serials, variables for storing and converting parameters, and various indicators.

The data logging code begins with a setup phase. This phase begins by setting up the pins that will be used to control the three LEDs: one to indicate if the system has power, one to indicate that GPS data were logged, and one to indicate that the other sensor data were logged. Then the power LED is turned on to show the system has power and is running. Next, communication is initialized with the computer, GPS, electrical conductivity EZO™ circuit, pH EZO™ circuit, dissolved oxygen EZO™ circuit, ADS1115 16-bit ADC shield, and microSD card. If the microSD card was not successfully initialized, the GPS and data LEDs are blinked five times and an error message is printed to the serial monitor to let the user know that the system will not work because of the microSD card. Then, the gain of the ADS1115 16-bit ADC shield is set to two to match the gain used for calibration, the GPS settings are adjusted to enable

the RMC and GGA strings and to set the update rate to 1 Hz, and several bytes are reserved for communication with the electrical conductivity, pH, and dissolved oxygen EZO™ circuits. Next, the code checks for the calibration files to calculate the parameters needed for the calibration of the nitrate, chloride, and thermocouple sensors. If the necessary calibration files exist, the code simply calculates the calibration parameters and moves on, however, if one or more calibration files is missing for a sensor an error variable is flagged and an error message is printed indicating that the sensor requires calibration. Once the code has checked for the calibration files of all four of the sensors, the code checks if the error variable has been flagged and if it has, the code will blink all three LEDs ten times to indicate to the user that there were missing calibration files. Using LEDs to indicate an error for both the microSD card and calibration parameters is done to ensure that the user knows if the system will be calibrated and working properly whether or not they are using a computer and will receive the error messages. Finally, the code begins the process of setting up a new file on the microSD card to store the data.

To set up a new data file, the code first waits for a GPS signal and prints a message indicating that the code has reached this point and is waiting for the GPS signal before proceeding. Once a GPS signal is obtained, another message is printing to indicate that the GPS has a signal and the code is now waiting for valid GPS data before using the data to name the new data file. The code evaluates whether or not the GPS data can be used to name the file by checking if either a time variable (HHMMSS.SSS) or a date variable (YYMMDD) are equal to 0 (their default value). Once both of these variables have been changed from 0, the code will run a function that uses the most recent GPS data to name the data file using the following format: DDHHMMSS.csv. Lastly, the code prints headers in the new data file and to the serial monitor

of the computer to indicate which data are included in each column. This completes the setup phase.

Once the setup is complete, data collection occurs as the loop phase continuously repeats. Each loop begins by waiting for new GPS data to be ready. Once new GPS data are received, the GPS data are parsed and the code checks if the new data are valid. This is done very similarly to how it is done in the setup phase. First, the time (HHMMSS.SSS) and date (YYMMDD) variables are determined using the new GPS data. Then the code checks that both of these variables are not equal to 0 and that the time variable is different from the last logged time variable. This checks that the GPS data are actually new data and that the time and date values are valid. If the GPS data are determined to be valid, the code then pulls the GPS data that needs to be logged (year, month, day, hour, minute, seconds, milliseconds, latitude, longitude, altitude, and number of satellites), converts it to the desired format (converts latitude and longitude to decimal degrees), and indicates that there are valid GPS data to be logged. Next, the code gets the raw data from the nitrate probe, chloride probe, type J thermocouple, and type K thermocouple and converts it to the parameter of interest using the calibration parameters. The code also gets the raw data from the turbidity sensor, however, it simply converts this data to a voltage rather than a turbidity value as discussed previously. Finally, the code checks that at least five seconds have occurred since the system was powered on and whether it is time to log new data; the log rate is set so that data will be logged once a second. If both of these conditions are met, a function to log the data from all of the sensors is run. If the function to log the data was completed successfully, the data LED is blinked to indicate that data were saved.

When the function to log data is called in the loop, the code begins the process of logging data to the microSD card. First, the code checks if there are any GPS data available to be logged,

and if there are, it logs the GPS data to the microSD card and prints the data to the serial monitor. The GPS LED is then blinked to indicate that GPS data were successfully logged. If there are no GPS data to be logged, the code instead logs an error value (-9999) to the microSD card, skips printing the GPS data to the serial monitor, and the GPS LED does not blink. Next, the code logs the Teensy 3.6 microcontroller's internal reference time (millis), the nitrate concentration, and the chloride concentration and prints the same values to the serial monitor. Then the code checks if there are available electrical conductivity, pH, and dissolved oxygen data to be logged, and if so, logs the data and prints it to the serial monitor. Like for the GPS data, if there are no available data to be logged, an error value (-9999) is logged instead and that value is skipped in the serial monitor. Next, the type J thermocouple, type K thermocouple, and turbidity data are all logged and printed to the serial monitor. Finally, the code indicates whether or not the function was executed successfully and if data were saved.

SerialEvent functions are run automatically between each loop. These functions check if there are any data available from the electrical conductivity, pH, and dissolved oxygen probes (serialEvent1 – electrical conductivity, serialEvent2 – pH, and serialEvent3 – dissolved oxygen) and if the data contains the correct number of characters and are therefore valid. The code stores any available data that are determined to be valid in variables that are used to log the data to the microSD card when the data logging function is run.

Appendix F: Multi-Sensor Array Data Logging Code

```
// Libraries for GPS
#include <Adafruit_GPS.h>
#include <SoftwareSerial.h>

// Library for SD Card
#include <SD.h>

// Library for Vernier Probes (Nitrate, Chloride, and Thermistor)
#include <math.h>

// Libraries for ADS1115 ADC (for Thermocouples)
#include <Adafruit_ADS1015.h>
#include <Wire.h>

// LEDs
#define powerLight 2                    // Red LED to turn on when system has
power
#define gpsLight 3                      // Blue LED to blink when GPS data was
logged
#define dataLight 4                     // Blue LED to blink when sensor data was
logged

// Pin Definitions
#define SD_PIN 53                       // Digital pin 53
#define GPS_Rx_PIN 10                   // Digital pin 10
#define GPS_Tx_PIN 11                   // Digital pin 11
#define Chloride_PIN A0                 // Analog pin A0
#define Nitrate_PIN A1                  // Analog pin A1
#define TypeJ_PIN 0                     // ADS1115 pin 0
#define TypeK_PIN 1                     // ADS1115 pin 1
#define Turbidity_PIN A3                // Analog pin A3

// Set Up for GPS
SoftwareSerial ssGPS(GPS_Rx_PIN, GPS_Tx_PIN);          // Create a software serial
communication
Adafruit_GPS GPS(&ssGPS);

// Set Up for ADS1115 ADC (for Thermocouples)
Adafruit_ADS1115 ads1115;

String f = "";                          // String for reading from files
int lr = 1000;                          // Log rate in ms
unsigned long ll = 0;                   // Time of last log
char logfilename[13];                   // Char to hold file name for data
```

```
// Variables for GPS
float lt, lg, a;                                    // lt = latitude (ddmm.mmmm), lg = longitude
(dddmm.mmmm), a = altitude (m)
uint8_t y, m, d, h, mn, s, sv;                      // y = year, m = month, d = day, h = hour, mn
= minute, s = second (UTC time), sv = number of satellites
uint16_t ms;                                        // ms = millisecond
boolean GPS_complete = false;                       // Indicator for if GPS data was valid
float timeCheck1;                                   // Variable to check if time data is new
float timeCheck2;                                   // Variable to check if time data is new and
valid
float dateCheck;                                    // Variable to check if date is valid
double mins = 0.0;                                  // Variable to hold minutes when converting
from ddmm.mmmm to decimal degrees
int degs = 0;                                       // Variable to hold degrees when converting from
ddmm.mmmm to decimal degrees
double lt_dd, lg_dd;                                // lt_dd = latitude (deg), lg_dd = longitude
(deg)

// Variables for Nitrate
int n, nvh, nvl;                                    // n = analog value of nitrate, nvh = analog value
of high standard, nvl = analog value of low standard
float nh, nl, nvhmV, nvlmV, nEo, nm;                // Variables for Nitrate Calibration, nh
= high standard concentration (mg/L), nl = low standard concentration (mg/L), nvhmV and
nvlmV = analog values of standard converted to mV, nEo and nm = calibration parameters
float nV, nmV, nC;                                  // Variables for Nitrate Readings, nV = n
converted to V, nmV = nV converted to mV, nC = nmV converted to mg/L

// Variables for Chloride
int c, cvh, cvl;                                    // c = analog value of chloride, cvh = analog value
of high standard, nvl = analog value of low standard
float ch, cl, cvhmV, cvlmV, cEo, cm;                // Variables for Chloride Calibration, ch
= high standard concentration (mg/L), cl = low standard concentration (mg/L), cvhmV and
cvlmV = analog values of standards converted to mV, cEo and cm = calibration parameters
float cV, cmV, cC;                                  // Variables for Chloride Readings, cV = c
converted to V, cmV = cV converted to mV, cC = cmV converted to mg/L

// Variables for Electrical Conductivity Communication
String iEC = "";                                    // String to hold strings sent from serial to EC
String sEC = "";                                    // String to hold strings sent from EC to be read
boolean iEC_complete = false;                       // Indicator for if string is ready to be sent
to EC
boolean sEC_complete = false;                       // Indicator for if EC string is ready to be
read

// Variables for pH Communication
```

```
String ipH = "";                                  // String to hold strings sent from serial to pH
String spH = "";                                  // String to hold strings sent from pH to be read
boolean ipH_complete = false;                     // Indicator for if string is ready to be sent
to pH
boolean spH_complete = false;                     // Indicator for if pH string is ready to be
read

// Variables for Dissolved Oxygen Communication
String iDO = "";                                  // String to hold strings sent from serial to DO
String sDO = "";                                  // String to hold strings sent from DO to be read
boolean iDO_complete = false;                     // Indicator for if string is ready to be sent
to DO
boolean sDO_complete = false;                     // Indicator for if DO string is ready to be
read

// Variables for Type J Thermocouple
int16_t t_J;                                      // t_J = analog value of temperature
float tl_J, th_J, tvl_J, tvh_J, tm_J, tb_J, tC_J;  // tl_J = low calibration temperature,
th_J = high calibration temperature, tvl_J = analog value of low calibration temperature, tvh_J =
analog value of high calibration temperature, tm_J = slope of linear calibration, tb_J = intercept
of linear calibration, tC_J = temperature reading in deg. C

// Variables for Type K Thermocouple
int16_t t_K;                                      // t_K = analog value of temperature
float tl_K, th_K, tvl_K, tvh_K, tm_K, tb_K, tC_K;  // tl_K = low calibration
temperature, th_K = high calibration temperature, tvl_K = analog value of low calibration
temperature, tvh_K = analog value of high calibration temperature, tm_K = slope of linear
calibration, tb_K = intercept of linear calibration, tC_K = temperature reading in deg. C

// Variables for Turbidity
int tbd;                                          // Variable for analog value of turbidity
float tbdV;                                       // Variable for Voltage of turbidity

boolean error;                                    // Variable to indicate if any errors occured
during setup

void setup() {
  // Set up LEDs
  pinMode(powerLight, OUTPUT);                    // Set up LED to indicate if power is
supplied
  pinMode(gpsLight, OUTPUT);                      // Set up LED to indicate if GPS data is
being logged
  pinMode(dataLight, OUTPUT);                     // Set up LED to indicate if data is
being logged
```

```
  digitalWrite(powerLight, HIGH);                    // Turn on power LED to indicate
system has power

  // Begin Communications
  Serial.begin(9600);                               // Begin communication with computer
  GPS.begin(9600);                                  // Begin communication with GPS
  Serial1.begin(9600);                              // Begin communication with EC (Rx - 19; Tx
- 18);
  Serial2.begin(9600);                              // Begin communication with pH (Rx - 17; Tx
- 16);
  Serial3.begin(9600);                              // Begin communication with DO (Rx - 15; Tx
- 14);
  ads1115.begin();                                  // Begin communication with ADS1115 ADC
(for Thermocouples)
  if (!SD.begin(SD_PIN)) {                          // Initiate microSD card
    Serial.println("SD Error");
    for (int i = 0; i < 5; i++) {                   // If there was an error with initiation, blink
blue LEDs 5 times
      digitalWrite(gpsLight, HIGH);
      digitalWrite(dataLight, HIGH);
      delay(250);
      digitalWrite(gpsLight, LOW);
      digitalWrite(dataLight, LOW);
      delay(250);
    }
  }

  // ADS1115 Settings
  ads1115.setGain(GAIN_TWO);                        // Sets gain to 2 (+/- 2.048 V)

  // GPS Settings
  GPS.sendCommand(PMTK_SET_NMEA_OUTPUT_RMCGGA);     // Enables
RMC and GGA strings
  GPS.sendCommand(PMTK_SET_NMEA_UPDATE_1HZ);        // Sets GPS update
rate to 1 Hz
  //  GPS.sendCommand(PGCMD_ANTENNA);               // Enables updates on
Antenna status

  // Reserve Bytes for Communication with EC, pH, and DO
  iEC.reserve(10);
  sEC.reserve(30);
  ipH.reserve(10);
  spH.reserve(30);
  iDO.reserve(10);
  sDO.reserve(30);
```

```
  // If Calibration Data Exists, Calculate Calibration Parameters
  if (SD.exists("nvh.csv") && SD.exists("nvl.csv") && SD.exists("nh.csv") &&
SD.exists("nl.csv")) { // Check if all nitrate calibration files exist, and if they do
    File f_nvh = SD.open("nvh.csv");                    // Open the calibration file for the high
standard nitrate calibration value
    if (f_nvh) {                                  // If the file was opened
      f = f_nvh.readStringUntil(13);                    // Get the high standard calibration value
for nitrate
      nvh = f.toFloat();                          // Convert that value to a number
      f_nvh.close();                              // Close the file
    }
    File f_nvl = SD.open("nvl.csv");                    // Open the calibration file for the low
standard nitrate calibration value
    if (f_nvl) {                                  // If the file was opened
      f = f_nvl.readStringUntil(13);                    // Get the low standard calibration value for
nitrate
      nvl = f.toFloat();                          // Convert that value to a number
      f_nvl.close();                              // Close the file
    }
    File f_nh = SD.open("nh.csv");                      // Open the calibration file for the high
standard nitrate concentration
    if (f_nh) {                                   // If the file was opened
      f = f_nh.readStringUntil(13);                     // Get the high standard concentration for
nitrate
      nh = f.toFloat();                           // Convert that value to a number
      f_nh.close();                               // Close the file
    }
    File f_nl = SD.open("nl.csv");                      // Open the calibration file for the low
standard nitrate concentration
    if (f_nl) {                                   // If the file was opened
      f = f_nl.readStringUntil(13);                     // Get the low standard concentration for
nitrate
      nl = f.toFloat();                           // Convert that value to a number
      f_nl.close();                               // Close the file
    }
    nvhmV = 137.55 * (nvh / 1023.0 * 5.0) - 0.1682;             // Using the nitrate calibration
values, calculate the nitrate calibration parameters
    nvlmV = 137.55 * (nvl / 1023.0 * 5.0) - 0.1682;             // These equations come from the
manual for the vernier nitrate probe
    nm = (nvhmV - nvlmV) / log(nh / nl);
    nEo = nvlmV - (nm * log(nl));
  }
  else {                                          // If no nitrate calibration data, indicate there was an
error
    error = true;
    Serial.println("Need Nitrate Calibration");
```

```
  }
  if (SD.exists("cvh.csv") && SD.exists("cvl.csv") && SD.exists("ch.csv") &&
SD.exists("cl.csv")) { // Check if all chloride calibration files exists, and if they do,
    File f_cvh = SD.open("cvh.csv");                    // Open the calibration file for the high
standard chloride calibration value
    if (f_cvh) {                                  // If the file was opened
      f = f_cvh.readStringUntil(13);                    // Get the high standard calibration value
for chloride
      cvh = f.toFloat();                          // Convert that value to a number
      f_cvh.close();                              // Close the file
    }
    File f_cvl = SD.open("cvl.csv");                    // Open the calibration file for the low
standard chloride calibration value
    if (f_cvl) {                                  // If the file was opened
      f = f_cvl.readStringUntil(13);                    // Get the low standard calibration value for
chloride
      cvl = f.toFloat();                          // Convert that value to a number
      f_cvl.close();                              // Close the file
    }
    File f_ch = SD.open("ch.csv");                      // Open the calibration file for the high
standard chloride concentration
    if (f_ch) {                                   // If the file was opened
      f = f_ch.readStringUntil(13);                     // Get the high standard concentration for
chloride
      ch = f.toFloat();                           // Convert that value to a number
      f_ch.close();                               // Close the file
    }
    File f_cl = SD.open("cl.csv");                      // Open the calibration file for the low
standard chloride concentration
    if (f_cl) {                                   // If the file was opened
      f = f_cl.readStringUntil(13);                     // Get the low standard concentration for
chloride
      cl = f.toFloat();                           // Convert that value to a number
      f_cl.close();                               // Close the file
    }
    cvhmV = 137.55 * (cvh / 1023.0 * 5.0) - 0.1682;             // Using the chloride calibration
values, calculate the chloride calibration parameters
    cvlmV = 137.55 * (cvl / 1023.0 * 5.0) - 0.1682;             // These equations come from the
manual for the vernier chloride probe
    cm = (cvhmV - cvlmV) / log(ch / cl);
    cEo = cvlmV - (cm * log(cl));
  }
  else {                                          // If no chloride calibration data, indicate there was
an error
    error = true;
    Serial.println("Need Chloride Calibration");
```

```
  }
  if (SD.exists("tvl_J.csv") && SD.exists("tvh_J.csv") && SD.exists("tl_J.csv") &&
SD.exists("th_J.csv")) { // Check if all type J thermocouple calibration files exists, and if they do,
    File f_tvl_J = SD.open("tvl_J.csv");                    // Open the calibration file for the lower
temperature calibration value
    if (f_tvl_J) {                                  // If the file was opened
      f = f_tvl_J.readStringUntil(13);                      // Get the lower temperature calibration
value
      tvl_J = f.toFloat();                          // Convert that value to a number
      f_tvl_J.close();                              // Close the file
    }
    File f_tvh_J = SD.open("tvh_J.csv");                    // Open the calibration file for the
higher temperature calibration value
    if (f_tvh_J) {                                  // If the file was opened
      f = f_tvh_J.readStringUntil(13);                      // Get the higher temperature calibration
value
      tvh_J = f.toFloat();                          // Convert that value to a number
      f_tvh_J.close();                              // Close the file
    }
    File f_tl_J = SD.open("tl_J.csv");                      // Open the calibration file for the lower
temperature value
    if (f_tl_J) {                                   // If the file was opened
      f = f_tl_J.readStringUntil(13);                       // Get the lower temperature value
      tl_J = f.toFloat();                           // Convert that value to a number
      f_tl_J.close();                               // Close the file
    }
    File f_th_J = SD.open("th_J.csv");                      // Open the calibration file for the higher
temperature value
    if (f_th_J) {                                   // If the file was opened
      f = f_th_J.readStringUntil(13);                       // Get the higher temperature value
      th_J = f.toFloat();                           // Convert that value to a number
      f_th_J.close();                               // Close the file
    }
    tm_J = (th_J - tl_J) / (tvh_J - tvl_J);                 // Using the temperature calibration and
temperature values, calculate the type J thermocouple calibration parameters
    tb_J = th_J - (tm_J * tvh_J);                           // These equations assume a linear
relationship
  }
  else {                                            // If no type J thermocouple calibration data, indicate
there was an error
    error = true;
    Serial.println("Need Type J  Thermocouple Calibration");
  }
  if (SD.exists("tvl_K.csv") && SD.exists("tvh_K.csv") && SD.exists("tl_K.csv") &&
SD.exists("th_K.csv")) { // Check if all type K thermocouple calibration files exists, and if they
do,
```

```
   File f_tvl_K = SD.open("tvl_K.csv");                    // Open the calibration file for the
lower temperature calibration value
   if (f_tvl_K) {                                        // If the file was opened
     f = f_tvl_K.readStringUntil(13);                      // Get the lower temperature calibration
value
     tvl_K = f.toFloat();                                // Convert that value to a number
     f_tvl_K.close();                                    // Close the file
   }
   File f_tvh_K = SD.open("tvh_K.csv");                    // Open the calibration file for the
higher temperature calibration value
   if (f_tvh_K) {                                        // If the file was opened
     f = f_tvh_K.readStringUntil(13);                      // Get the higher temperature calibration
value
     tvh_K = f.toFloat();                                // Convert that value to a number
     f_tvh_K.close();                                    // Close the file
   }
   File f_tl_K = SD.open("tl_K.csv");                      // Open the calibration file for the lower
temperature value
   if (f_tl_K) {                                         // If the file was opened
     f = f_tl_K.readStringUntil(13);                       // Get the lower temperature value
     tl_K = f.toFloat();                                 // Convert that value to a number
     f_tl_K.close();                                     // Close the file
   }
   File f_th_K = SD.open("th_K.csv");                      // Open the calibration file for the
higher temperature value
   if (f_th_K) {                                         // If the file was opened
     f = f_th_K.readStringUntil(13);                       // Get the higher temperature value
     th_K = f.toFloat();                                 // Convert that value to a number
     f_th_K.close();                                     // Close the file
   }
   tm_K = (th_K - tl_K) / (tvh_K - tvl_K);                 // Using the temperature calibration
and temperature values, calculate the type K thermocouple calibration parameters
   tb_K = th_K - (tm_K * tvh_K);                           // These equations assume a linear
relationship
  }
  else {                                                 // If no type K thermocouple calibration data,
indicate there was an error
   error = true;
   Serial.println("Need Type K Thermocouple Calibration");
  }

  if (error == true) {                                   // If there was an error during set up (missing
calibration data)
   digitalWrite(powerLight, LOW);                          // Turn power light off
   delay(250);
   for (int i = 0; i < 10; i++) {                        // Blink all three LEDs 10 times
```

```
    digitalWrite(powerLight, HIGH);
    digitalWrite(gpsLight, HIGH);
    digitalWrite(dataLight, HIGH);
    delay(250);
    digitalWrite(powerLight, LOW);
    digitalWrite(gpsLight, LOW);
    digitalWrite(dataLight, LOW);
    delay(250);
   }
  }
  digitalWrite(powerLight, HIGH);                          // Turn power light back on

  // Begin a New File and Print Header to the File
  Serial.println("GPS looking for signal...");
  while (!GPS.newNMEAreceived()) {                         // Wait for GPS data before
continuing
   GPS.read();
  }
  Serial.println("GPS waiting for good data...");
  while (timeCheck2 == 0 || dateCheck == 0) {              // Wait until GPS data is valid
before naming file
   GPS.read();
   if (GPS.newNMEAreceived()) {
    GPS.parse(GPS.lastNMEA());
    timeCheck2 = float(GPS.hour) * 10000 + float(GPS.minute) * 100 + float(GPS.seconds) +
float(GPS.milliseconds) / 1000;
    dateCheck = float(GPS.year) * 10000 + float(GPS.month) * 100 + float(GPS.day);
   }
  }
  updateFileName();                                        // Run the function that updates the file name
  File f_dat = SD.open(logfilename, FILE_WRITE);           // Open the new data file
  if (f_dat) {                                             // If the file was opened

f_dat.println("Year,Month,Day,Hour,Minute,Second,Millisecond,Latitude,Longitude,Altitude,Sa
tellites,Millis,Nitrate,Chloride,EC,pH,DO,Type J Thermocouple,Type K
Thermocouple,Turbidity");  // Print the column headers for the data
   f_dat.close();                                          // Close the file
  }
  Serial.print("Year");
  Serial.print('\t');
  Serial.print("Month");
  Serial.print('\t');
  Serial.print("Day");
  Serial.print('\t');
  Serial.print("Hour");
  Serial.print('\t');
```

```
  Serial.print("Minute");
  Serial.print('\t');
  Serial.print("Second");
  Serial.print('\t');
  Serial.print("Millisecond");
  Serial.print('\t');
  Serial.print("Latitude");
  Serial.print('\t');
  Serial.print("Longitude");
  Serial.print('\t');
  Serial.print("Altitude");
  Serial.print('\t');
  Serial.print("Satellites");
  Serial.print('\t');
  Serial.print("Millis");
  Serial.print('\t');
  Serial.print("Nitrate");
  Serial.print('\t');
  Serial.print("Chloride");
  Serial.print('\t');
  Serial.print("EC");
  Serial.print('\t');
  Serial.print("pH");
  Serial.print('\t');
  Serial.print("DO");
  Serial.print('\t');
  Serial.print("Type J");
  Serial.print('\t');
  Serial.print("Type K");
  Serial.print('\t');
  Serial.println("Turbidity");
}

// Get Data from EC
void serialEvent1() {                                   // Triggered if there is an incoming string from
EC
  sEC = "";                                             // Clear the EC sensor string variable
  sEC = Serial1.readStringUntil(13);                    // Store the incoming string in the EC
sensor string variable, to be sent to the serial and/or logged
  sEC_complete = true;                                  // Indicate that there is a string to be sent to
the serial and/or data to be logged
  if (sEC.length() > 5) {                               // If there is an error and the data is too long
(two data points mashed together)
    sEC_complete = false;                               // Indicate that there is no data to be logged
instead
  }
```

```
}

// Get Data from pH
void serialEvent2() {                                   // Triggered if there is an incoming string from
from pH
  spH = "";                                            // Clear the pH sensor string variable
  spH = Serial2.readStringUntil(13);                          // Store the incoming string in the pH
sensor string variable, to be sent to the serial and/or logged
  spH_complete = true;                                  // Indicate that there is a string to be sent to
the serial and/or data to be logged
  if (spH.length() > 6) {                              // If there is an error and the data is too long
(two data points mashed together)
    spH_complete = false;                               // Indicate that there is no data to be logged
instead
  }
}

// Get Data from DO
void serialEvent3() {                                   // Triggered if there is an incoming string from
DO
  sDO = "";                                            // Clear the DO sensor string variable
  sDO = Serial3.readStringUntil(13);                          // Store the incoming string in the DO
sensor string variable, to be sent to the serial and/or logged
  sDO_complete = true;                                  // Indicate that there is a string to be sent to
the serial and/or data to be logged
  if (sDO.length() > 5) {                              // If there is an error and the data is too long
(two data points mashed together)
    sDO_complete = false;                               // Indicate that there is no data to be logged
instead
  }
}

void loop() {
  // Get GPS Data
  while (!GPS.newNMEAreceived()) {                            // Wait for next NMEA string
    GPS.read();
  }
  if (GPS.newNMEAreceived()) {                               // If there is new GPS data to be read
    GPS.parse(GPS.lastNMEA());                               // Parse the new GPS data
    timeCheck2 = float(GPS.hour) * 10000 + float(GPS.minute) * 100 + float(GPS.seconds) +
float(GPS.milliseconds) / 1000; // Determine the new time in HHMMSS.SSS format
    dateCheck = float(GPS.year) * 10000 + float(GPS.month) * 100 + float(GPS.day);  //
Determine the new date in YYMMDD format
    if (timeCheck1 != timeCheck2 && timeCheck2 != 0 && dateCheck != 0) {  // If the new data
is valid
      y = GPS.year;                                     // Get the year (YY)
```

```
    m = GPS.month;                          // Get the month (MM)
    d = GPS.day;                           // Get the day (DD)
    h = GPS.hour;                          // Get the hour (HH)
    mn = GPS.minute;                        // Get the minute (MM)
    s = GPS.seconds;                       // Get the seconds (SS)
    ms = GPS.milliseconds;                      // Get the milliseconds (MMM)
    lt = GPS.latitude;                     // Get the latitude in ddmm.mmmm format
    lg = GPS.longitude;                     // Get the longitude in dddmm.mmmm format
    a = GPS.altitude;                      // Get the altitude in meters
    sv = GPS.satellites;                   // Get the number of satellites
    timeCheck1 = timeCheck2;                        // Set the new time as the last recorded
time

    // Convert latitude and longitude to decimal degrees
    mins = fmod((double)lt, 100.0);                 // Determine the minutes of the latitude
    degs = (int)(lt / 100);                // Determine the degrees of the latitude
    lt_dd = degs + (mins / 60);                 // Compute the latitude in decimal degrees

    mins = fmod((double)lg, 100.0);                  // Determine the minutes of the longitude
    degs = (int)(lg / 100);                // Determine the degrees of the longitude
    lg_dd = degs + (mins / 60);                 // Compute the longitude in decimal degrees

    GPS_complete = true;                    // Indicate that there is valid GPS data to be
logged
   }
  }

  // Get Nitrate Data
  n = analogRead(Nitrate_PIN);                  // Read the analog value from the nitrate
probe
  nV = n / 1023.0 * 5.0;                  // Convert that value to V using 10 bit
conversion from Arduino ADC
  nmV = 137.55 * nV - 0.1682;                   // Convert that value to mV using
equation given in the vernier nitrate probe manual
  double(nval) = ((nmV - nEo) / nm);                   // Using the equation given in the
vernier nitrate probe manual and the calibration parameters,
  nC = exp(nval);                         // Calculate the concentration of nitrate in mg/L

  // Get Chloride Data
  c = analogRead(Chloride_PIN);                   // Read the analog value from the
chloride probe
  cV = c / 1023.0 * 5.0;                  // Convert that value to V using 10 bit
conversion from Arduino ADC
  cmV = 137.55 * cV - 0.1682;                   // Convert that value to mV using
equation given in the vernier nitrate probe manual
```

```
  double(cval) = ((cmV - cEo) / cm);                    // Using the equation given in the vernier chloride probe manual and the calibration parameters,
  cC = exp(cval);                                       // Calculate the concentration of chloride in mg/L

  // Get Type J Thermocouple Data
  t_J = ads1115.readADC_SingleEnded(TypeJ_PIN);         // Read the analog value from the type J thermocouple
  tC_J = tm_J * t_J + tb_J;                             // Using the calibration parameters and an assumed linear relationship, calculate temperature in deg. C from the average analog value

  // Get Type K Thermocouple Data
  t_K = ads1115.readADC_SingleEnded(TypeK_PIN);         // Read the analog value from the type K thermocouple
  tC_K = tm_K * t_K + tb_K;                             // Using the calibration parameters and an assumed linear relationship, calculate temperature in deg. C of the average analog value

  // Get Turbidity Data
  tbd = analogRead(Turbidity_PIN);                      // Read the analog value from the turbidity sensor
  tbdV = tbd * (5.0 / 1023.0);                          // Convert that value to V using 10 bit conversion from Arduino ADC

  // Log Data to SD Card
  if ((ll + lr) <= millis() && millis() >= 5000) {      // Check if it's time for next data point
    if (logData()) {                                    // If data was logged
      digitalWrite(dataLight, HIGH);                    // Blink the data LED to indicate data was logged
      delay(100);
      digitalWrite(dataLight, LOW);
    }
  }
}

// Function that logs the data to the micro SD card and prints it in the serial
byte logData() {                                        // Triggered when function to log the data is run
  File f_dat = SD.open(logfilename, FILE_WRITE);        // Open the data file in write mode
  if (f_dat) {                                          // If the file was opened
    if (GPS_complete == true) {                         // If there is valid GPS data to log
      f_dat.print(y + 2000);                            // Log the year (YYYY)
      f_dat.print(',');
      f_dat.print(m);                                   // Log the month (MM)
      f_dat.print(',');
      f_dat.print(d);                                   // Log the day (DD)
      f_dat.print(',');
```

```
    f_dat.print(h);                          // Log the hour (HH, UTC time)
    f_dat.print(',');
    f_dat.print(mn);                          // Log the minute (MM, UTC time)
    f_dat.print(',');
    f_dat.print(s);                          // Log the seconds (SS, UTC time)
    f_dat.print(',');
    f_dat.print(ms);                          // Log the milliseconds (MMM, UTC time)
    f_dat.print(',');
    f_dat.print(lt_dd, 6);                    // Log latitude in degrees
    f_dat.print(',');
    f_dat.print(lg_dd, 6);                    // Log longitude in degrees
    f_dat.print(',');
    f_dat.print(a, 1);                       // Log altitude in meters
    f_dat.print(',');
    f_dat.print(sv);                         // Log the number of satellites
    f_dat.print(',');
    Serial.print(y + 2000);                   // Log the year (YYYY)
    Serial.print('\t');
    Serial.print(m);                         // Log the month (MM)
    Serial.print('\t');
    Serial.print(d);                         // Log the day (DD)
    Serial.print('\t');
    Serial.print(h);                         // Log the hour (HH, UTC time)
    Serial.print('\t');
    Serial.print(mn);                         // Log the minute (MM, UTC time)
    Serial.print('\t');
    Serial.print(s);                         // Log the seconds (SS, UTC time)
    Serial.print('\t');
    Serial.print(ms);                         // Log the milliseconds (MMM, UTC time)
    Serial.print('\t');
    Serial.print(lt_dd, 6);                   // Log latitude in degrees
    Serial.print('\t');
    Serial.print(lg_dd, 6);                   // Log longitude in degrees
    Serial.print('\t');
    Serial.print(a, 1);                      // Log altitude in meters
    Serial.print('\t');
    Serial.print(sv);                        // Log the number of satellites
    Serial.print('\t');
    GPS_complete = false;                          // Indicate that GPS data was printed to the
serial
    digitalWrite(gpsLight, HIGH);                  // Blink the GPS LED indicating GPS
data was logged
    delay(100);
    digitalWrite(gpsLight, LOW);
  }
```

```
  else {                                    // If there is not valid GPS data to log, log error
value (-9999) for each value missed
    f_dat.print("-9999,-9999,-9999,-9999,-9999,-9999,-9999,-9999,-9999,-9999,-9999,");
    Serial.print('\t');
    Serial.print('\t');
    Serial.print('\t');
    Serial.print('\t');
    Serial.print('\t');
    Serial.print('\t');
    Serial.print('\t');
    Serial.print('\t');
    Serial.print('\t');
    Serial.print('\t');
    Serial.print('\t');
  }
  f_dat.print(millis());                    // Log the Arduino time in millis
  f_dat.print(',');
  f_dat.print(nC);                          // Log the nitrate concentration in mg/L
  f_dat.print(',');
  f_dat.print(cC);                          // Log the chloride concentration in mg/L
  f_dat.print(',');
  Serial.print(millis());                   // Log the Arduino time in millis
  Serial.print('\t');
  Serial.print(nC);                         // Log the nitrate concentration in mg/L
  Serial.print('\t');
  Serial.print(cC);                         // Log the chloride concentration in mg/L
  Serial.print('\t');
  if (sEC_complete == true) {               // If there is a string to read from EC
    f_dat.print(sEC);                       // Log EC string/data
    f_dat.print(',');
    Serial.print(sEC);                      // Log EC string/data
    Serial.print('\t');
    sEC_complete = false;                   // Indicate string from EC was read
  }
  else {                                    // If there is not a string to read from EC
    f_dat.print("-9999,");                  // Log error value (-9999)
    Serial.print('\t');                     // Log nothing
  }
  if (spH_complete == true) {               // If there is a string to read from pH
    f_dat.print(spH);                       // Log pH string/data
    f_dat.print(',');
    Serial.print(spH);                      // Log pH string/data
    Serial.print('\t');
    spH_complete = false;                   // Indicate string from pH was read
  }
  else {                                    // If there is not a string to read from pH
```

```
    f_dat.print("-9999,");                    // Log error value (-9999)
    Serial.print('\t');                       // Log nothing
   }
   if (sDO_complete == true) {                // If there is a string to read from DO
    f_dat.print(sDO);                         // Log DO string/data
    f_dat.print(',');
    Serial.print(sDO);                        // Log DO string/data
    Serial.print('\t');
    sDO_complete = false;                     // Indicate string from DO was read
   }
   else {                                     // If there is not a string to read from DO
    f_dat.print("-9999,");                    // Log error value (-9999)
    Serial.print('\t');                       // Log nothing
   }
   f_dat.print(tC_J);                         // Log the type J thermocouple temperature in
deg. C
   f_dat.print(',');
   f_dat.print(tC_K);                         // Log the type K thermocouple temperature in
deg. C
   f_dat.print(',');
   Serial.print(tC_J);                        // Log the type J thermocouple temperature in
deg. C
   Serial.print('\t');
   Serial.print(tC_K);                        // Log the type K thermocouple temperature in
deg. C
   Serial.print('\t');
   f_dat.println(tbdV);                       // Log the turbidity voltage in V
   Serial.println(tbdV);                      // Log the turbidity voltage in V
   f_dat.close();                             // Close the file
   ll = millis();                             // Save the time as the time of the last log
   return 1;                                  // Return that data was logged successfully
  }
  return 0;                                   // If the file was not opened, return that data was
not logged
}

void updateFileName() {                       // Triggered when function to update the
file name is run
  memset(logfilename, 0, strlen(logfilename));          // Clear the char used to store the file
name
  //  String mon = String(GPS.month);         // Get the month from the GPS and
convert to a string (Can be switched with seconds for MMDDMMHH format)
  String dy = String(GPS.day);                // Get the day from the GPS and convert to
a string
  String hr = String(GPS.hour);               // Get the hour from the GPS and convert
to a string
```

```
  String mnt = String(GPS.minute);                    // Get the minute from the GPS and
convert to a string
  String sec = String(GPS.seconds);                   // Get the second from the GPS and
convert to a string
  //  if (mon.length() < 2)                           // If the month is less than 10, add a '0' as the
first digit
  //  {
  //    mon = "0" + mon;
  //  }
  if (dy.length() < 2)                                // If the day is less than 10, add a '0' as the first
digit
  {
    dy = "0" + dy;
  }
  if (hr.length() < 2)                                // If the hour is less than 10, add a '0' as the first
digit
  {
    hr = "0" + hr;
  }
  if (mnt.length() < 2)                               // If the minute is less than 10, add a '0' as the
first digit
  {
    mnt = "0" + mnt;
  }
  if (sec.length() < 2)                               // If the second is less than 10, add a '0' as the
first digit
  {
    sec = "0" + sec;
  }
  String FileName = dy + hr + mnt + sec + ".csv";     // Combine the day, hour, minute
and second string to DDHHMMSS format and add a '.csv' to create the file name string
  FileName.toCharArray(logfilename, 13);              // Convert the file name string to a
char array
}
```

Appendix G: Description of Seismic Sensor Code

At the top of the code all of the necessary variables are assigned and several definitions are included to make future editing easier. This includes, but is not limited to, variables used for storing GPS and seismic data, variables used for controlling timing and the LED state, definitions of the pins used for the various connections, and settings such as the sample interval and duration for seismic data, number of points of GPS data to collect, and how often to collect a GPS data file. To ensure that a GPS file will be collected at the very beginning, the variable used to indicate the time since the last GPS data were logged is set to a value greater than the GPS interval. In addition, several definitions and variables necessary for controlling the ADS1220 24-bit ADC shield are defined. These include pin definitions, command bytes, addresses, programmable gain, voltage reference, and configuration registers which control settings such as sampling rate, gain, etc. These are described in more detail in the datasheet. In this section, the SdFat library is also included, which is an alternative to the built-in SD library that allows file names longer than eight characters and is used for controlling the microSD card.

The code begins with a setup phase that runs once upon startup and in which any one-time setup that is necessary for any of the components is completed. The setup phase begins by initializing the GPS and sending a command to the GPS which disables all NMEA sentences other than the RMC and GGA sentences used by the code. Then the code sets up the GPS enable pin used to turn the GPS off when not in use to conserve power and immediately turns the GPS off. Next, an interrupt is created to detect the rising edge of the PPS signal. The GPS is turned off before this step to ensure that the interrupt does not run and begin collecting PPS times while setup is still occurring. Once the setup for the GPS is complete, the code then sets up the pin to control the LED and initializes the microSD card. If the microSD card is not initialized

successfully, the code then blinks the LED five times to indicate there was an error. Finally, the code sets up the ADS1220 24-bit ADC shield by running a function which starts and sets up the ADS1220 24-bit ADC shield with the desired settings. This function is described in more detail in the following paragraph. This completes the setup phase and data collection then begins.

To start up the ADS1220 24-bit ADC shield, the procedures described in the data sheet are followed. First, the CS pin is set as an output and the DRDY pin is set as an input. Then, SPI communication is started using the functions of the SPI library and the bit order (MSB first) and data mode are specified. Then the system is reset by setting the CS pin low, high, then low again with 2 ms pauses between transitions, sending the reset command byte to the system, and setting the CS pin high again. This initiates the ADS1220 24-bit ADC shield, however, the desired settings must be set and changed from the defaults before it can be used. To do this, the four configuration registers must be set to the values that correspond to the desired settings. For each register, this is done by setting the CS pin low, sending a command byte which indicates which registers should be changed at which address, then sending the value to set the configuration registers to for the desired settings, and finally setting the CS pin high again. Once this process is repeated for each of the four configuration registers, the ADS1220 24-bit ADC shield is ready to be used.

Data collection occurs in the loop phase, a phase which continuously repeats after the setup phase. At the beginning of each loop the code checks if it is time to collect another file of GPS data. If it’s time to collect GPS data, the GPS is turned on and the code begins the process of naming the new GPS file. First, a variable is set to indicate that GPS data collection for naming the file is not complete. Then, until GPS data have been collected, the code continues to check if GPS data are available and if GPS data have been collected. Whenever GPS data are

available, the code gets the new NMEA sentence from the GPS and runs a function to parse the data.

To parse the data, the code first checks whether the fourth, fifth, and sixth characters in the NMEA sentence are RMC or GGA, which tells the code the type of sentence the new data are contained in, what data needs to be pulled from the sentence, and where in the sentence that data are located. Then, regardless of whether the sentence is an RMC or GGA sentence, the code checks through each character contained in the sentence and finds the location of all the commas. As described earlier, the code then uses the sentence type and the locations of the commas to identify which characters correspond to the GPS parameters of interest and pulls these parameters from the sentence. Finally, the code checks if the GPS data are valid. For an RMC sentence, the GPS data are considered valid if the status is A, the mode is A, D, or E, the UTC time contains no decimal seconds, and the UTC time differs from the last recorded UTC time. If RMC data are valid, the code increases the count of stored RMC data, updates the last recorded UTC time, and changes the state of the LED. For a GGA sentence, the GPS data are considered valid if the fix is either 1 or 2. If GGA data are valid, the code simply increases the count of stored GGA data. As explained previously, the LED state is not changed when new GGA data are stored since the GGA and RMC sentences update at the same time. The last thing the code does when parsing data, regardless of the type of sentence, is clear the variable used to store incoming NMEA sentences so the next sentence can be received.

In addition to checking for new GPS data and whether GPS data have been collected, while waiting for valid GPS data to name the file, the code also checks that the number of stored PPS times, RMC data, and GGA data are not over 190. If there are more than 190 measurements stored for any of these data types, the code resets the count to 0 to prevent the storage arrays,

which can only contain 200 measurements, from overflowing and causing the code to freeze and the system to stop working. This scenario is unlikely to occur as long as the GPS is located in an area with a clear GPS signal, however, it is good practice to include as a safety measure.

Once there is at least one valid set of RMC data, the code determines that the GPS data collection necessary for naming the new GPS file is complete and proceeds to name the file. The code then uses the GPS data from the valid RMC sentence to name the GPS file using a YYMMDDHHMMSS.gps format and prints column headers describing the GPS data that will be logged to the file. This includes the UTC time, date, latitude, longitude, altitude, number of satellites, and the Teensy 3.6 microcontroller's internal reference time when the PPS signal corresponding to the GPS data was detected. Of these, the most important are the PPS time and UTC time which are essential for achieving the desired real time accuracy for recorded seismic signals.

With the GPS file now created, the code resets the variable used to indicate if GPS data collection is complete and the PPS, RMC, and GGA counts, then begins the process of collecting and storing the desired number of GPS data measurements (for our system we chose 150 measurements). Just like when collecting GPS data for naming the GPS file, the code continues to check if new GPS data are available, if GPS data collection is complete (if at least 150 RMC and GGA strings have been stored), and if the number of stored PPS times, RMC data, and GGA data are over 190. When GPS data are available, the code stores the Teensy 3.6 microcontroller's internal reference time at which the GPS data were detected (which will be needed to pair the GPS data to the correct PPS time as discussed in the following paragraphs) and runs the function to parse the data, determine if the data are valid, and store any valid data. If any of the PPS,

RMC, or GGA counts exceed 190 the count is reset to ensure that the storage arrays do not overflow.

In addition to storing GPS data provided from NMEA sentences, the code simultaneously stores the Teensy 3.6 microcontroller's internal reference time whenever a PPS signal is detected. To ensure the time of this PPS signal is accurate, an interrupt is used. This means that whenever a PPS signal is detected, the code pauses whatever else it may be doing, immediately records the time of the PPS signal and increases the count of PPS signals stored, then continues with whatever it was doing before the interrupt occurred.

Once sufficient GPS data have been collected, the code begins pairing the stored GPS data to the corresponding PPS times using another function. This function begins by separately pairing the RMC data and GGA data to the PPS times, then uses the paired PPS times to pair the RMC data and GGA data together. To match the RMC data and GGA data to the PPS times, for each stored set of RMC or GGA data, the code checks if there is a PPS time that occurred 175 to 700 ms before the RMC or GGA data were detected. This length of time corresponds to the delay which occurs between when the PPS signal occurs and the NMEA sentence is received by the system. If a matching PPS time is found, the PPS time is set as the Teensy 3.6 microcontroller's internal reference time corresponding to the RMC or GGA data. If no matching PPS time was found, an error value is used instead and the data will ultimately be rejected. Next, the code goes through the matched PPS times of all the RMC data and checks if there is GGA data with the same matched PPS time. The code saves the indices corresponding to the RMC data and GGA data with the same matched PPS times. Once the function determines the indices of all the GPS data that has both RMC data and GGA data with a matched PPS time, the code uses the saved indices corresponding to this data to print the GPS data to the microSD card. Then the code

resets the GPS timer, resets the PPS, RMC, and GGA counts, turns off the GPS, and indicates that GPS data collection is complete.

After the code completes the GPS portion of the loop phase, the code begins the process of saving a new file of seismic data. First, a function is run to name the new file using a FILE#####.geo format. This function begins by checking file names to see which files exist and what the name of the next sequential file should be. To reduce the time it takes for this function to scan through the existing files, a variable is used to store the number of the last created file and only file numbers that come after are checked. If it is the very first time the function is run, the last file number defaults to 0. Once the file number is determined, the code puts together the file name using the file number to match the FILE#####.geo format. Finally, the code sets up the newly named file by printing the last logged GPS data at the top and creating column headers for the seismic data.

One the file is named, the code begins to collect seismic data to store in the file. First, the number of data points, the counter for the LED, and the timer to indicate how long data should be collected are all reset. Then, until the timer reaches the specified length of time that should be recorded in each file (which was defined at the top of the code), the code continuously stores seismic data at the specified sampling rate. To store data at the specified sampling rate, a second timer is used to indicate when a new sample should be taken. While the code is storing seismic data, it continuously checks if this second timer has reached the specified sample interval (also defined at the top of the code) and if it has, begins the process of storing another data point. First the code gets the seismic data; it stores the Teensy 3.6 microcontroller's current internal reference time and the analog values for the x, y, and z-axis seismic data (obtained from the ADS1220 24-bit ADC shield by running a function described in the following paragraph) in the

location of the storage arrays defined by the current number of data points. Then, to prepare for the next sample, the sample interval timer is reset, the number of data points is increased, and the count for the LED is also increased. At this point, the code also checks if it should change the state of the LED, which it does every time the LED count reaches 50. If the LED count has reached 50, the code checks the LED indicator to see if the LED is currently on or off, changes the LED state to the corresponding opposite state, and then resets the count. This entire process repeats until the specified duration of time has passed.

To obtain the analog values of the x, y, and z-axis seismic data, a function is run following the procedures described in the data sheet to take a reading from one of the analog channels of the ADS1220 24-bit ADC shield. First, variables that will be used to store and convert the data are defined. Next, the CS pin is set low, a command byte is sent to select which analog channel should be used, and the CS pin is set high again. Then, to begin the conversion of the signal to an analog value, the CS pin is again set low, a command byte to start the conversion is sent, and the CS pin is set high once more. The code then waits until the conversion is complete by monitoring the state of the DRDY pin which remains high while the conversion is ongoing then drops low. As soon as the DRDY pin drops low, the code gets the analog value by setting the CS pin low once more, using the functions of the SPI library to pull the data, and setting the CS pin high again. Finally, the data are converted to a 32-bit analog value by bitshifting the bits to the appropriate locations and the 32-bit analog value is returned so it can be stored. Each time a sample is taken this function is run three times, once for each axis and their corresponding analog channels on the ADS1220 32-bit ADC shield.

Once the code has finished storing seismic data, it must print that data to the microSD card in order to ensure the data are saved and to clear memory space for the next stage of data

collection (whether it is GPS or seismic data collection). Using the number of data points stored, the code cycles through each entry in the storage arrays (starting from 0 and increasing by 1 until reaching the last stored data point) to print the Teensy 3.6 microcontroller's internal reference time and the corresponding analog values of the x, y, and z-axis seismic data to the microSD card. Once this is complete, the code has successfully logged a new file of seismic data and completed another cycle of the loop phase. The code then returns to the beginning of the loop phase, checks once again if it is time to collect GPS data, and the entire loop phase continues to repeat endlessly until power is lost.

Appendix H: Seismic Sensor Code

```
// Library for SD card
#include <SdFat.h>

// Library for Protocentral ADS1220 ADC
#include <SPI.h>

// Define GPS serial port
#define gps Serial1

// Define Pin Connections
#define GPS_PPS_pin 2
#define GPS_EN_pin 3
#define LED_pin 29

// Protocentral ADS1220 ADC Definitions
// Teensy Pins
#define CS_PIN 10
#define DRDY_PIN 9
// Multiplexer Pins
#define Ax_pin 0x80
#define Ay_pin 0x90
#define Az_pin 0xA0
// Scale
#define PGA 1 // Programmable Gain = 1
#define VREF 3.3 // Internal reference of 2.048V
#define VFSR VREF/PGA
#define FULL_SCALE   (((int32_t)1<<23)-1)
//Config registers
#define CONFIG_REG0_ADDRESS 0x00
#define CONFIG_REG1_ADDRESS 0x01
#define CONFIG_REG2_ADDRESS 0x02
#define CONFIG_REG3_ADDRESS 0x03
// Other
#define REG_CONFIG0_MUX_MASK 0xF0
#define SPI_MASTER_DUMMY 0xFF
#define RESET 0x06 //Send the RESET command (06h) to make sure the ADS1220 is properly
reset after power-up
#define START 0x08 //Send the START/SYNC command (08h) to start converting in
continuous conversion mode
#define WREG 0x40

////////////////////////////////////////////////////////////////// Settings
//////////////////////////////////////////////////////////////////
#define sample_interval 2400 // Time to wait before taking another measurement (us)
```

```
#define log_length 53600000 // Length of time to record data for (us): maximum value of sample_interval * storage array size (4000 * 21500)
#define gps_interval 3600000 // Time to wait before recording more gps data (ms)
#define num_points 150 // Number of GPS data points to take before storing data to file (must be less than size of storage array)
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////

// Counts
int count_pps = 0; // Count number of detected pps that are recorded
int count_rmc = 0; // Count number of valid RMC sentences recorded
int count_gga = 0; // Count number of valid GGA sentences recorded
int count_match = 0; // Count number of RMC and GGA sentences matched

// Timing Variables
unsigned long bufms; // Time at which a new NMEA sentence is received
unsigned long ppsms[200]; // Time at which pps signals were detected
unsigned long rmcms[200]; // Time at which a new RMC sentence was received
unsigned long rmcms_pps[200]; // Time at which pps signal corresponding to RMC sentence was detected
unsigned long ggams[200]; // Time at which a new GGA sentence was received
unsigned long ggams_pps[200]; // Time at which pps signal corresponding to GGA sentence was detected

// NMEA Receiving Variables
String buf; // Variable to hold new NMEA sentence
String check_rmc; // Variable used to check if RMC sentence contains new data (last time recorded - ddmmss.sss)
String UTC; // Variable used to check if RMC sentence contains new data (new time - ddmmss.sss)

// NMEA Parsing Variables
int j; // Counter used to assign new character to appropriate location
int comma[18]; // Variable used to indicate the position of commas in the newly received NMEA sentence

// Time Pairing Variables
boolean rmc_match = false; // Variable to indicate if the RMC sentence had a matching PPS time
boolean gga_match = false; // Variable to indicate if the GGA sentence had a matching PPS time
int rmc_matched_indices[200]; // Variable used to indicate which RMC sentences had a matching PPS time
int gga_matched_indices[200]; // Variable used to indicate which GGA sentenecs had a matchign PPS time

// RMC Variables: Time and Date
char utctime[11][200]; // Variable to store GPS time (hhmmss.sss)
```

```
char utcdate[7][200]; // Variable to store GPS date (ddmmyy)
char rmcstatus; // Variable to store GPS status ('A' is valid)
char rmcmode; // Variable to store GPS mode ('A', 'D', or 'E' is valid)

// GGA Variables: Location
char lat[10][200]; // Variable to store GPS latitude (ddmm.mmmm)
char ns[200]; // Variable to store GPS latitude direction (N = north, S = south)
char lon[11][200]; // Variable to store GPS longitude (dddmm.mmmm)
char ew[200]; // Variable to store GPS longitude direction (E = east, W = west)
char msl[8][200]; // Variable to store altitude above mean sea level (m)
char nsats[3][200]; // Variable to store number of satellites
char fix; // Variable to store GPS fix ('1' or '2' is valid)

// Accelerometer and Geophone Variables
int p; // Variable to indicate current index of storage array and total number of points
unsigned long us[13400]; // Variable to store microseconds
int32_t Ax[13400]; // Variable to store analog value of x-axis (accelerometer or geophone)
int32_t Ay[13400]; // Variable to store analog value of y-axis (accelerometer or geophone)
int32_t Az[13400]; // Variable to store analog value of z-axis (accelerometer or geophone)

// LED Variables
boolean light = false; // Variable to indicate if LED is on or off (false = off, true = on)
int count = 0; // Variable to indicate when to switch LED on/off while collecting data (switch
every 50 data points, every ~250 ms)

// SD Variables
SdFatSdio SD; // Define name and class type for the SD card
char gpsFileName[17]; // Variable for the file name for the GPS data
(YYMMDDHHMMSS.gps)
char logFileName[13]; // Variable for the file name for the geophone or accelerometer data
(FILE#####.geo or FILE#####.acc)
char suffix[5] = ".acc"; // Variable for the suffix of the geophone or accelerometer file name
(".geo" or ".acc")
int last_file = 0; // Variable to indicate the number corresponding to the last created file

// Protocentral ADS1220 ADC Settings Variables
uint8_t m_config_reg0 = 0x00; //Default settings: AINP=AIN0, AINN=AIN1, Gain 1, PGA
enabled
uint8_t m_config_reg1 = 0xD0; //Default settings: DR=2000 SPS, Mode=Turbo, Conv
mode=single-shot, Temp Sensor disabled, Current Source off
uint8_t m_config_reg2 = 0xC0; //Default settings: Vref Analog Supply, No 50/60Hz rejection,
power open, IDAC off
uint8_t m_config_reg3 = 0x00; //Default settings: IDAC1 disabled, IDAC2 disabled, DRDY pin
only
```

```
// Timing Control Variables (elapsedMicros and elapsedMillis are special variable types that automatically increase as time passes and can be reset)
elapsedMicros log_time; // Variable to indicate time since logging analog data began
elapsedMicros data_time; // Variable to indicate time since last analog data was logged
elapsedMillis gps_time = gps_interval + 1000; // Variable to indicate time since last GPS data was logged (Start with a value greater than gps interval so a GPS file will be taken at the beginning)
boolean gps_complete = false; // Variable to indicate when the desired number of GPS points is logged and GPS data collection is complete

void setup() {
  // Set Up the GPS
  gps.begin(9600); // Start GPS communication
  gps.println( F("$PMTK314,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0*28") ); // Disable all NMEA sentences except RMC and GGA
  pinMode(GPS_EN_pin, OUTPUT); // Set up the pin to turn the GPS on and off
  digitalWrite(GPS_EN_pin, LOW); // Turn GPS off
  attachInterrupt(GPS_PPS_pin, pps, RISING); // Attach interrupt to detect PPS signal

  // Set Up the LED
  pinMode(LED_pin, OUTPUT); // Set up the LED pin

  // Set Up the SD Card
  if (!SD.begin()) { // Start SD card
    for (int i = 0; i < 5; i++) { // If there was an error, blink LED 5 times
      digitalWrite(LED_pin, HIGH);
      delay(250);
      digitalWrite(LED_pin, LOW);
      delay(250);
    }
  }

  // Begin the Protocentral ADS1220 ADC
  ADS1220_Start();
}

void loop() {
  if (gps_time >= gps_interval) { // Record specified number of GPS points every GPS interval
    digitalWrite(GPS_EN_pin, HIGH); // Turn GPS on
    gps_complete = false; // Reset GPS indicator
    while (gps_complete == false) { // Wait for valid GPS data before naming GPS file
      if (gps.available()) { // If GPS data is available
        bufms = micros(); // Record the time the new NMEA sentence was received
        buf = gps.readStringUntil(13); // Get the new NMEA sentence
        parseGPS(); // Parse the NMEA sentence and store it if valid
      }
```

```
    if (count_rmc > 1) { // If there was a valid RMC sentence
      gps_complete = true; // Indicate that the GPS data is valid and the file can be named
    }
    if (count_pps > 190) { // If the pps count gets close to full, reset it to prevent the counter from overfilling the storage array and causing errors
      count_pps = 0;
    }
    if (count_rmc > 190) { // If the RMC count gets close to full, reset it to prevent the counter from overfilling the storage array and causing errors
      count_rmc = 0;
    }
    if (count_gga > 190) { // If the GGA count gets close to full, reset it to prevent the counter from overfilling the storage array and causing errors
      count_gga = 0;
    }
  }
  updateGpsFileName(); // Name the GPS file
  gps_complete = false; // Reset GPS indicator
  count_pps = 0; // Reset the PPS counter
  count_rmc = 0; // Reset the RMC counter
  count_gga = 0; // Reset the GGA counter
  while (gps_complete == false) { // Wait until specified number of GPS points is collected and logged
    if (gps.available()) { // If GPS data is available
      bufms = micros(); // Record the time the new NMEA sentence was received
      buf = gps.readStringUntil(13); // Get the new NMEA sentence
      parseGPS(); // Parse the NMEA sentence and store it if valid
    }
    if (count_rmc > num_points && count_gga > num_points) { // If the specified number of GPS points has been met for both sentences
      pairTime(); // Pair the GPS data with the PPS times
      File gpsFile = SD.open(gpsFileName, FILE_WRITE); // Log the GPS data, open the file
      if (gpsFile) { // If the file was opened
        for (int e1 = 0; e1 < count_match; e1++) { // For every entry in storage array that was matched to a PPS time
          for (int e2 = 0; e2 < 10; e2++) { // Print the characters associated with GPS time
            gpsFile.print(utctime[e2][rmc_matched_indices[e1]]);
          }
          gpsFile.print(',');
          for (int e2 = 0; e2 < 6; e2++) { // Print the characters associated with GPS date
            gpsFile.print(utcdate[e2][rmc_matched_indices[e1]]);
          }
          gpsFile.print(',');
          for (int e2 = 0; e2 < 9; e2++) { // Print the characters associated with GPS latitude
            gpsFile.print(lat[e2][gga_matched_indices[e1]]);
          }
```

```
          gpsFile.print(',');
          gpsFile.print(ns[gga_matched_indices[e1]]); // Print the character associated with GPS
latitude direction
          gpsFile.print(',');
          for (int e2 = 0; e2 < 10; e2++) { // Print the characters associated with GPS longitude
            gpsFile.print(lon[e2][gga_matched_indices[e1]]);
          }
          gpsFile.print(',');
          gpsFile.print(ew[gga_matched_indices[e1]]); // Print the character associated with GPS
longitude direction
          gpsFile.print(',');
          for (int e2 = 0; e2 < 7; e2++) {
            gpsFile.print(msl[e2][gga_matched_indices[e1]]); // Print the characters associated with
GPS altitude
          }
          gpsFile.print(',');
          for (int e2 = 0; e2 < 2; e2++) {
            gpsFile.print(nsats[e2][gga_matched_indices[e1]]); // Print the characters associated
with the number of satellites
          }
          gpsFile.print(',');
          gpsFile.println(rmcms_pps[rmc_matched_indices[e1]]); // Print the PPS detection time
corresponding to the GPS data
        }
        gpsFile.close(); // Close the file
        gps_time = 0; // Reset the gps timer
        count_pps = 0; // Reset the PPS counter
        count_rmc = 0; // Reset the RMC counter
        count_gga = 0; // Reset the GGA counter
        gps_complete = true; // Indicate GPS data logging is complete
        digitalWrite(GPS_EN_pin, LOW); // Turn GPS off
      }
    }
    if (count_pps > 190) { // If the pps count gets close to full, reset it to prevent the counter from
overfilling the storage array and causing errors
      count_pps = 0;
    }
    if (count_rmc > 190) { // If the RMC count gets close to full, reset it to prevent the counter
from overfilling the storage array and causing errors
      count_rmc = 0;
    }
    if (count_gga > 190) { // If the GGA count gets close to full, reset it to prevent the counter
from overfilling the storage array and causing errors
      count_gga = 0;
    }
  }
```

```
  }
  updateLogFileName(); // Name the geophone or accelerometer data file
  p = 0; // Reset the index/number of points indicator
  count = 0; // Reset the LED counter
  log_time = 0; // reset the log timer to start data collection
  while (log_time <= log_length) { // Store data for 1 minute
   if (data_time >= sample_interval) { // Take a measurement every 5 ms
    us[p] = micros(); // Save the current time (us)
    Ax[p] = ADS1220_Read(Ax_pin); // Save the x-axis analog value
    Ay[p] = ADS1220_Read(Ay_pin); // Save the y-axis analog value
    Az[p] = ADS1220_Read(Az_pin); // Save the z-axis analog value
    data_time = 0; // Reset the sampling interval timer
    p = p + 1; // Increase the index/number of points
    count = count + 1; // Increase the LED counter
    if (count > 50) { // If the LED counter reaches 50
     if (light == false) { // Turn the LED on/off to indicate data was logged
      digitalWrite(LED_pin, HIGH);
      light = true;
     }
     else {
      digitalWrite(LED_pin, LOW);
      light = false;
     }
     count = 0; // Reset the LED counter
    }
   }
  }
  File logFile = SD.open(logFileName, FILE_WRITE); // Log the analog data to the SD card
  if (logFile) {
   for (int e = 0; e < p; e++) { // For each measurement stored
    logFile.print(us[e]); // Print the time (us)
    logFile.print(',');
    logFile.print(Ax[e]); // Print the x-axis analog value
    logFile.print(',');
    logFile.print(Ay[e]); // Print the y-axis analog value
    logFile.print(',');
    logFile.println(Az[e]); // Print the z-axis analog value
   }
   logFile.close(); // Close the file
  }
}

void parseGPS() { // Function to parse the GPS data
  if (buf.charAt(4) == 'R' && buf.charAt(5) == 'M' && buf.charAt(6) == 'C') { // If the NMEA
sentence is an RMC sentence
```

```
   rmcms[count_rmc] = bufms; // Save the time the NMEA sentence was received as the RMC
sentence time
   j = 0; // Reset the character index
   for (int i = 0; i < buf.length(); i++) { // For each character in the NMEA sentence
    if (buf.charAt(i) == ',') { // If the character is a comma
     comma[j] = i; // Record the location of the comma
     j = j + 1; // Increase the character index
    }
   }
   j = 0; // Reset the character index
   for (int i = 0; i < buf.length(); i++) { // For all characters in the NMEA sentence
    if (i > comma[0] && i < comma[1]) { // If the character falls between the 1st and 2nd
commas
     utctime[j][count_rmc] = buf.charAt(i); // Record the character as part of the UTC time
     j = j + 1; // Increase the character index
     if (i == comma[1] - 1) { // If the character is the last of the UTC time
      j = 0; // Reset the character index for the next parameter
     }
    }
    else if (i > comma[1] && i < comma[2]) { // If the character falls between the 2nd and 3rd
commas
     rmcstatus = buf.charAt(i); // Record the character as the RMC status
    }
    else if (i > comma[8] && i < comma[9]) { // If the character falls between the 9th and 10th
commas
     utcdate[j][count_rmc] = buf.charAt(i); // Recrod the character as part of the UTC date
     j = j + 1; // Increase the character index
     if (i == comma[9] - 1) { // If the character is the last of the UTC date
      j = 0; // Reset the character index for the next parameter
     }
    }
    else if (i == (comma[11] + 1)) { // If the character is the character immediately after the last
(12th) comma
     rmcmode = buf.charAt(i); // Record the character as the RMC mode
    }
   }
   if (rmcstatus == 'A') { // If the RMC status was valid
    if (rmcmode == 'A' || rmcmode == 'D' || rmcmode == 'E') { // And the RMC mode was valid
     if (utctime[7][count_rmc] == '0' && utctime[8][count_rmc] == '0' &&
utctime[9][count_rmc] == '0') { // And the UTC time was valid (no decimal seconds)
      UTC = ""; // Clear the new UTC time variable
      for (int e = 0; e < 10; e++) { // For each character of the UTC time
       UTC = UTC + String(utctime[e][count_rmc]); // Add that character to the UTC time
string
      }
```

```
        if (check_rmc != UTC) { // If the new UTC time is different from the last recorded UTC
time
          count_rmc = count_rmc + 1; // Increase the RMC counter
          check_rmc = UTC; // And set the new UTC time as the last recorded UTC time
          if (light == false) { // Turn the LED on/off to indicate data was stored
            digitalWrite(LED_pin, HIGH);
            light = true;
          }
          else {
            digitalWrite(LED_pin, LOW);
            light = false;
          }
        }
      }
    }
  }
 if (buf.charAt(4) == 'G' && buf.charAt(5) == 'G' && buf.charAt(6) == 'A') { // If the NMEA
sentence is a GGA sentence
   ggams[count_gga] = bufms; // Save the time the NMEA sentence was received as the GGA
sentence time
   j = 0; // Reset the character index
   for (int i = 0; i < buf.length(); i++) { // For each character in the NMEA sentence
    if (buf.charAt(i) == ',') { // If the character is a comma
      comma[j] = i; // Record the location of the comma
      j = j + 1; // Increase the character index
    }
   }
   j = 0; // Reset the character index
   for (int i = 0; i < buf.length(); i++) { // For all characters in the NMEA sentence
    if (i > comma[1] && i < comma[2]) { // If the character falls between the 2nd and 3rd
commas
      lat[j][count_gga] = buf.charAt(i); // Record the character as part of the latitude
      j = j + 1; // Increase the character index
      if (i == comma[2] - 1) { // If the character is the last of the latitude
        j = 0; // Reset the character index for the next parameter
      }
    }
    else if (i > comma[2] && i < comma[3]) { // If the character falls between the 3rd and 4th
commas
      ns[count_gga] = buf.charAt(i); // Record the character as the latitidue direction
    }
    else if (i > comma[3] && i < comma[4]) { // If the character falls between the 4th and 5th
commas
      lon[j][count_gga] = buf.charAt(i); // Record the character as part of the longitude
      j = j + 1; // Increase the character index
```

```
      if (i == comma[4] - 1) { // If the character is the last of the longitude
        j = 0; // Reset the character index for the next parameter
      }
    }
    else if (i > comma[4] && i < comma[5]) { // If the character falls between the 5th and 6th
commas
      ew[count_gga] = buf.charAt(i); // Record the character as the longitude direction
    }
    else if (i > comma[5] && i < comma[6]) { // If the character falls between the 6th and 7th
commas
      fix = buf.charAt(i); // Record the character as the GPS fix
    }
    else if (i > comma[6] && i < comma[7]) { // If the character falls between the 7th and 8th
commas
      nsats[j][count_gga] = buf.charAt(i); // Record the character as part of the number of
satellites
      j = j + 1; // Increase the character index
      if (i == comma[7] - 1) { // If the character is the last of the number of satellites
        j = 0; // Reset the character index for the next parameter
      }
    }
    else if (i > comma[8] && i < comma[9]) { // If the character falls between the 9th and 10th
commas
      msl[j][count_gga] = buf.charAt(i); // Record the character as part of the altitude
      j = j + 1; // Increase the character index
      if (i == comma[9] - 1) { // If the character is the last of the altitude
        j = 0; // Reset the character index for the next parameter
      }
    }
  }
  if (fix == '1' || fix == '2') { // If the GPS was valid
    count_gga = count_gga + 1; // Increase the GGA counter
  }
 }
 buf = ""; // Clear NMEA sentence string so it can read the next value
}

void pairTime() { // Pair RMC and GGA sentences to PPS times
 // Pair RMC and GGA times to PPS times separately
 for (int e1 = 0; e1 < num_points; e1++) { // For the desired number of GPS data points (may
want to change to use counts instead)
   for (int e2 = 0; e2 < count_pps; e2++) { // And for all recorded PPS times (may want to
change to size of storage array)
    if (ppsms[e2] > rmcms[e1]) { // If the PPS time is greater than the RMC time, assume the
micros function rolled over between times (RMC time should occur after PPS)
```

```
      if (((2 ^ 32) - 1 - ppsms[e2] + rmcms[e1]) < 700000 && ((2 ^ 32) - 1 - ppsms[e2] +
rmcms[e1]) > 175000) { // If the RMC sentence arrived 175 to 700 ms after the PPS
        rmcms_pps[e1] = ppsms[e2]; // Pair the RMC sentence to the PPS time
        rmc_match = true; // Indicate that the RMC sentence was paired with a PPS time
       }
     }
     else if ((rmcms[e1] - ppsms[e2]) < 700000 && (rmcms[e1] - ppsms[e2]) > 175000) { //
Otherwise, if the RMC time is greater than the PPS time as expected and the RMC sentence
arrived 175 to 700 ms after the PPS
      rmcms_pps[e1] = ppsms[e2]; // Pair the RMC sentence to the PPS time
      rmc_match = true; // Indicate that the RMC sentence was paired with a PPS time
     }
     if (ppsms[e2] > ggams[e1]) { // If the PPS time is greater than the GGA time, assume the
micros function rolled over between times (GGA time should occur after PPS)
      if (((2 ^ 32) - 1 - ppsms[e2] + ggams[e1]) < 700000 && ((2 ^ 32) - 1 - ppsms[e2] +
ggams[e1]) > 175000) { // If the GGA sentence arrived 175 to 700 ms after the PPS
        ggams_pps[e1] = ppsms[e2]; // Pair the GGA sentence to the PPS time
        gga_match = true; // Indicate that the GGA sentence was paired with a PPS time
       }
     }
     else if ((ggams[e1] - ppsms[e2]) < 700000 && (ggams[e1] - ppsms[e2]) > 175000) { //
Otherwise, if the GGA time is greater than the PPS time as expected and the GGA sentence
arrived 175 to 700 ms after the PPS
      ggams_pps[e1] = ppsms[e2]; // Pair the GGA sentence to the PPS time
      gga_match = true; // Indicate that the GGA sentence was paired with a PPS time
     }
    }
    if (rmc_match == true) { // If the RMC sentence was paired with a PPS time
     rmc_match = false; // Reset the RMC match indicator
    }
    else { // Otherwise, if the RMC sentence was not paired with a PPS time
     rmcms_pps[e1] = 99999; // Set the PPS time equal to an error value (99999)
    }
    if (gga_match == true) { // If the GGA sentence was paired with a PPS time
     gga_match = false; // Reset the GGA match indicator
    }
    else { // Otherwise, if the GGA sentence was not paired with a PPS time
     ggams_pps[e1] = 99999; // Set the PPS time equal to an error value(99999)
    }
   }

   // Pair RMC and GGA times together
   count_match = 0; // Reset the count of the number of paired RMC and GGA sentences (that
were also paired with PPS times)
   for (int e1 = 0; e1 < num_points; e1++) { // For all of the stored RMC sentences
    for (int e2 = 0; e2 < num_points; e2++) { // And for all the stored GGA sentences
```

```
      if (rmcms_pps[e1] == 99999) { // If the RMC sentence was not paired with a PPS time
        // Ignore the sentence (do nothing)
      }
      else if (ggams_pps[e2] == 99999) { // If the GGA sentence was not paired with a PPS time
        // Ignore the sentence (do nothing)
      }
      else if (rmcms_pps[e1] == ggams_pps[e2]) { // Otherwise, if both the RMC and GGA
sentences were paired to the same PPS time
        rmc_matched_indices[count_match] = e1; // Save the RMC sentence as fully paired, valid
GPS data
        gga_matched_indices[count_match] = e2; // Saev the GGA sentence as fully paired, valid
GPS data
        count_match = count_match + 1; // Increase the number of paired RMC and GGA sentences
      }
    }
  }
}

void pps() { // Function that runs when a PPS signal is detected
  ppsms[count_pps] = micros(); // Store the current time as the time the PPS was detected
  count_pps = count_pps + 1; // Increase the count of the number of detected PPS signals
}

void updateGpsFileName() { // Update the file name to new YYMMDDHHMMSS.gps
  memset(gpsFileName, 0, strlen(gpsFileName)); // Clear the char to store the file name
  // Use the most recent GPS data to name the GPS file
  gpsFileName[0] = utcdate[4][0]; // Pull the first digit of the year from utcdate
  gpsFileName[1] = utcdate[5][0]; // Pull the second digit of the year from utcdate
  gpsFileName[2] = utcdate[2][0]; // Pull the first digit of the month from utcdate
  gpsFileName[3] = utcdate[3][0]; // Pull the second digit of the month from utcdate
  gpsFileName[4] = utcdate[0][0]; // Pull the first digit of the day from utcdate
  gpsFileName[5] = utcdate[1][0]; // Pull the second digit of the day from utcdate
  gpsFileName[6] = utctime[0][0]; // Pull the first digit of the hour from utctime
  gpsFileName[7] = utctime[1][0]; // Pull the second digit of the hour from utctime
  gpsFileName[8] = utctime[2][0]; // Pull the first digit of the minute from utctime
  gpsFileName[9] = utctime[3][0]; // Pull the second digit of the minute from utctime
  gpsFileName[10] = utctime[4][0]; // Pull the first digit of the second from utctime
  gpsFileName[11] = utctime[5][0]; // Pull the second digit of the second from utctime
  strcat(gpsFileName, ".gps"); // Add the ".gps" suffix to the file name
  File gpsFile = SD.open(gpsFileName, FILE_WRITE); // Print headers for GPS data file
  if (gpsFile) {
    gpsFile.print("Time");
    gpsFile.print(',');
    gpsFile.print("Date");
    gpsFile.print(',');
    gpsFile.print("Latitude");
```

```
    gpsFile.print(',');
    gpsFile.print("NS");
    gpsFile.print(',');
    gpsFile.print("Longitude");
    gpsFile.print(',');
    gpsFile.print("EW");
    gpsFile.print(',');
    gpsFile.print("Altitude (m)");
    gpsFile.print(',');
    gpsFile.print("Satellites");
    gpsFile.print(',');
    gpsFile.println("PPS");
    gpsFile.close();
  }
}

void updateLogFileName() {
  for (int i = last_file; i < 32768; i++) { // For all possible positive values of int
    memset(logFileName, 0, strlen(logFileName)); // Clear the char to store the file name
    if (i < 10) {
      sprintf(logFileName, "%s%d%s", "FILE0000", i, suffix);
    }
    else if (i < 100) {
      sprintf(logFileName, "%s%d%s", "FILE000", i, suffix);
    }
    else if (i < 1000) {
      sprintf(logFileName, "%s%d%s", "FILE00", i, suffix);
    }
    else if (i < 10000) {
      sprintf(logFileName, "%s%d%s", "FILE0", i, suffix);
    }
    else {
      sprintf(logFileName, "%s%d%s", "FILE", i, suffix);
    }
    if (!SD.exists(logFileName)) {
      last_file = i;
      break;
    }
  }
  File logFile = SD.open(logFileName, FILE_WRITE); // Print GPS data and headers for
geophone or accelerometer data file
  if (logFile) {
    int e1 = count_match - 1; // For the last entry in the storage array that was matched to a PPS
time
    for (int e2 = 0; e2 < 10; e2++) { // Print the characters associated with GPS time
      logFile.print(utctime[e2][rmc_matched_indices[e1]]);
```

```
    }
    logFile.print(',');
    for (int e2 = 0; e2 < 6; e2++) { // Print the characters associated with GPS date
      logFile.print(utcdate[e2][rmc_matched_indices[e1]]);
    }
    logFile.print(',');
    for (int e2 = 0; e2 < 9; e2++) { // Print the characters associated with GPS latitude
      logFile.print(lat[e2][gga_matched_indices[e1]]);
    }
    logFile.print(',');
    logFile.print(ns[gga_matched_indices[e1]]); // Print the character associated with GPS latitude
direction
    logFile.print(',');
    for (int e2 = 0; e2 < 10; e2++) { // Print the characters associated with GPS longitude
      logFile.print(lon[e2][gga_matched_indices[e1]]);
    }
    logFile.print(',');
    logFile.print(ew[gga_matched_indices[e1]]); // Print the character associated with GPS
longitude direction
    logFile.print(',');
    for (int e2 = 0; e2 < 7; e2++) {
      logFile.print(msl[e2][gga_matched_indices[e1]]); // Print the characters associated with GPS
altitude
    }
    logFile.print(',');
    for (int e2 = 0; e2 < 2; e2++) {
      logFile.print(nsats[e2][gga_matched_indices[e1]]); // Print the characters associated with the
number of satellites
    }
    logFile.print(',');
    logFile.println(rmcms_pps[rmc_matched_indices[e1]]); // Print the PPS detection times
corresponding to the GPS data
  }
  logFile.println("micros,Ax,Ay,Az"); // Print headers for geophone or accelerometer data
  logFile.close();
}

void ADS1220_Start() {
  // Start ADS1220
  pinMode(CS_PIN, OUTPUT);
  pinMode(DRDY_PIN, INPUT);
  SPI.begin();
  SPI.setBitOrder(MSBFIRST);
  SPI.setDataMode(SPI_MODE1);
  // Reset
  delay(100);
```

```
  digitalWrite(CS_PIN, LOW);
  delay(2);
  digitalWrite(CS_PIN, HIGH);
  delay(2);
  digitalWrite(CS_PIN, LOW);
  SPI.transfer(RESET);
  digitalWrite(CS_PIN, HIGH);
  delay(100);
  // Set Configuration Register 0 to Selected Settings
  digitalWrite(CS_PIN, LOW);
  SPI.transfer(WREG | (CONFIG_REG0_ADDRESS << 2));
  SPI.transfer(m_config_reg0);
  digitalWrite(CS_PIN, HIGH);
  delay(100);
  // Set Configuration Register 1 to Selected Settings
  digitalWrite(CS_PIN, LOW);
  SPI.transfer(WREG | (CONFIG_REG1_ADDRESS << 2));
  SPI.transfer(m_config_reg1);
  digitalWrite(CS_PIN, HIGH);
  delay(100);
  // Set Configuration Register 2 to Selected Settings
  digitalWrite(CS_PIN, LOW);
  SPI.transfer(WREG | (CONFIG_REG2_ADDRESS << 2));
  SPI.transfer(m_config_reg2);
  digitalWrite(CS_PIN, HIGH);
  delay(100);
  // Set Configuration Register 3 to Selected Settings
  digitalWrite(CS_PIN, LOW);
  SPI.transfer(WREG | (CONFIG_REG3_ADDRESS << 2));
  SPI.transfer(m_config_reg3);
  digitalWrite(CS_PIN, HIGH);
  delay(100);
}

int32_t ADS1220_Read(uint8_t Mux_Pin) {
  static byte SPI_Buff[3];
  int32_t adc_data = 0;
  int32_t bit24;
  // Select Channel
  m_config_reg0 &= ~REG_CONFIG0_MUX_MASK;
  m_config_reg0 |= Mux_Pin;
  digitalWrite(CS_PIN, LOW);
  SPI.transfer(WREG | (CONFIG_REG0_ADDRESS << 2));
  SPI.transfer(m_config_reg0);
  digitalWrite(CS_PIN, HIGH);
  // Start Conversion
```

```
  digitalWrite(CS_PIN, LOW);
  SPI.transfer(START);
  digitalWrite(CS_PIN, HIGH);
  // Wait for Conversion to Complete
  while (digitalRead(DRDY_PIN) == HIGH) {}
  // Get the Converted Data
  digitalWrite(CS_PIN, LOW);
  for (int i = 0; i < 3; i++) {
    SPI_Buff[i] = SPI.transfer(SPI_MASTER_DUMMY);
  }
  digitalWrite(CS_PIN, HIGH);
  bit24 = SPI_Buff[0];
  bit24 = (bit24 << 8) | SPI_Buff[1];
  bit24 = (bit24 << 8) | SPI_Buff[2];
  bit24 = (bit24 << 8);
  adc_data = (bit24 >> 8);

  return adc_data;
}
```

Appendix I: MatLab Code for Compiling Seismic Data

```
%% Fresh start
clear; clc; close all; fclose all;

dataName = 'SusieCube1';

%% Determine the names of any existing GPS files.

% First, get the file information for all files with .gps extension.
files = dir('*.gps');
% Then pull the names from the structure containing the file information.
gpsNames = extractfield(files,'name');

%% Get the GPS data

time = cell(length(gpsNames),1);
date = cell(length(gpsNames),1);
lat = cell(length(gpsNames),1);
lon = cell(length(gpsNames),1);
alt = cell(length(gpsNames),1);
sats = cell(length(gpsNames),1);
pps = cell(length(gpsNames),1);
tag = cell(length(gpsNames),1);

for i = 1:length(gpsNames)-1
    fileID = fopen(gpsNames{i});
    M1 = textscan(fileID,'%q');
    M1 = M1{1}(3:end);
    fclose(fileID);

    fileID = fopen(gpsNames{i+1});
    M2 = textscan(fileID,'%q');
    M2 = M2{1}(3:end);
    fclose(fileID);

    M = [M1; M2];

    time{i} = zeros(length(M),1);
    date{i} = zeros(length(M),1);
    lat{i} = zeros(length(M),1);
    lon{i} = zeros(length(M),1);
    alt{i} = zeros(length(M),1);
    sats{i} = zeros(length(M),1);
    pps{i} = zeros(length(M),1);
    tag{i} = M1{end};

    for j = 1:length(M)
        char = M{j};
        time{i}(j) = str2double(char(1:10));
        date{i}(j) = str2double(char(12:17));
        if char(29) == 'N'
            lat{i}(j) = str2double(char(19:27));
```

```
        elseif char(29) == 'S'
            lat{i}(j) = -str2double(char(19:27));
        else
            lat{i}(j) = 99999;
        end
        if char(42) == 'E'
            lon{i}(j) = str2double(char(31:40));
        elseif char(42) == 'W'
            lon{i}(j) = -str2double(char(31:40));
        else
            lon{i}(j) = 99999;
        end
        alt{i}(j) = str2double(char(44:48));
        sats{i}(j) = str2double(char(52:53));
        pps{i}(j) = str2double(char(55:end));
    end
end

fileID = fopen(gpsNames{end});
M1 = textscan(fileID,'%q');
M1 = M1{1}(3:end);
fclose(fileID);

M = M1;

time{length(gpsNames)} = zeros(length(M),1);
date{length(gpsNames)} = zeros(length(M),1);
lat{length(gpsNames)} = zeros(length(M),1);
lon{length(gpsNames)} = zeros(length(M),1);
alt{length(gpsNames)} = zeros(length(M),1);
sats{length(gpsNames)} = zeros(length(M),1);
pps{length(gpsNames)} = zeros(length(M),1);
tag{length(gpsNames)} = M1{end};

for j = 1:length(M)
    char = M{j};
    time{length(gpsNames)}(j) = str2double(char(1:10));
    date{length(gpsNames)}(j) = str2double(char(12:17));
    if char(29) == 'N'
        lat{length(gpsNames)}(j) = str2double(char(19:27));
    elseif char(29) == 'S'
        lat{length(gpsNames)}(j) = -str2double(char(19:27));
    else
        lat{length(gpsNames)}(j) = 99999;
    end
    if char(42) == 'E'
        lon{length(gpsNames)}(j) = str2double(char(31:40));
    elseif char(42) == 'W'
        lon{length(gpsNames)}(j) = -str2double(char(31:40));
    else
        lon{length(gpsNames)}(j) = 99999;
    end
    alt{length(gpsNames)}(j) = str2double(char(44:48));
    sats{length(gpsNames)}(j) = str2double(char(52:53));
    pps{length(gpsNames)}(j) = str2double(char(55:end));
```

```
end

clear char fileID files M M1 M2 Names

%% Convert GPS time to real time.

time_real = cell(length(time),1);
time_adj = cell(length(time),1);

for i = 1:length(time)
    % Separate HHMMSS.SSS into separate variables for hour, min, sec, and cs.
    rh = mod(time{i}, 10000);
    h = (time{i} - rh) / 10000;
    rm = mod(time{i}, 100);
    m = (time{i} - rm) / 100 - h * 100;
    s = time{i} - h * 10000 - m * 100;

    % Combine the separate hour, min, sec, and cs into real time in hours.
    time_real{i} = h + m/60 + s/60/60;

    % Adjust GPS time so time continually increases (resets every 24 hours).
    % Each time GPS time resets (value of next GPS time is less than current
    % GPS time) add an additional 24 hours to each GPS time value.
    adj = 0;
    time_adj{i} = time_real{i};
    for j = 2:length(time_real{i})
        if time_real{i}(j) < time_real{i}(j-1)
            adj = adj + 24;
        end
        time_adj{i}(j) = time_real{i}(j) + adj;
    end
end

%% Adjust real times so that time continually increases between cells as
% (not just within each cell, as done previously). If the first GPS time of
% one cell is less than the first GPS time of the previous cell, the time
% has reset between the cells and an addtional 24 hours must be added to
% all values of the cell.

adj2 = 0;
time_adj2 = time_adj;
for i = 1:length(time_adj)-1
    if time_adj{i+1}(1) < time_adj{i}(1)
        adj2 = adj2 + 24;
    end
    time_adj2{i+1} = time_adj{i+1} + adj2;
end

%% Adjust recorded micros so time continually increases (resets every
% 4294967295 us). Each time micros resets (value of next micros is less
% than current micros) add an additional 4294967295 us to each micros
% value.

pps_adj = cell(length(pps),1);
```

```
for i = 1:length(pps)
    adj = 0;
    pps_adj{i} = pps{i};
    for j = 2:length(pps{i})
        if pps{i}(j) < pps{i}(j-1)
            adj = adj + 4294967295;
        end
        pps_adj{i}(j) = pps{i}(j) + adj;
    end
end

%% Use the 2 min segments to determine the linear trend between PPS times
% and real GPS time between each pair of segments.

line = cell(length(pps_adj),1);
S = cell(length(pps_adj),1);
mu = cell(length(pps_adj),1);
time_calc = cell(length(pps_adj),1);
dif = cell(length(pps_adj),1);
time_fix = cell(length(pps_adj),1);
pps_good = cell(length(pps_adj),1);
time_good = cell(length(pps_adj),1);

for i = 1:length(pps_adj)
    [line{i},S{i},mu{i}] = polyfit(pps_adj{i},time_adj2{i},1);
    time_calc{i} = polyval(line{i},pps_adj{i},S{i},mu{i});
    dif{i} = (time_calc{i} - time_adj2{i})*60*60*1000;

    %     subplot(1,3,1)
    %     plot(time_calc{i},dif{i},'.')
    %     xlabel('Time (hours)')
    %     ylabel('Difference (ms)')
    %     title(['Original: ' num2str(i)])

    % If any of the differences are approximately 1 sec off, the GPS data is
    % likely incorrectly paired. Adjust the data by adding 1 sec.

    check = dif{i} > 500;
    time_fix{i} = time_adj2{i};
    time_fix{i}(check) = time_fix{i}(check) + 1/60/60;

    % Recalculate the linear trends between PPS times and real GPS times.

    [line{i},S{i},mu{i}] = polyfit(pps_adj{i},time_fix{i},1);
    time_calc{i} = polyval(line{i},pps_adj{i},S{i},mu{i});
    dif{i} = (time_calc{i} - time_fix{i})*60*60*1000;

    %     subplot(1,3,2)
    %     plot(time_calc{i},dif{i},'.')
    %     xlabel('Time (hours)')
    %     ylabel('Difference (ms)')
    %     title(['Correct Pairing: ' num2str(i)])
```

```
    % Remove any GPS data that has a difference greater than 1 ms.

    check = abs(dif{i}) > 1;
    pps_good{i} = pps_adj{i}(~check);
    time_good{i} = time_fix{i}(~check);

    % Recalculate the linear trends between PPS times and real GPS times.

    [line{i},S{i},mu{i}] = polyfit(pps_good{i},time_good{i},1);
    time_calc{i} = polyval(line{i},pps_good{i},S{i},mu{i});
    dif{i} = (time_calc{i} - time_good{i})*60*60*1000;

    %     subplot(1,3,3)
    %     plot(time_calc{i},dif{i},'.')
    %     xlabel('Time (hours)')
    %     ylabel('Difference (ms)')
    %     title(['Filter Bad Data: ' num2str(i)])

    %     set(gcf,'Units','inches')
    %     set(gcf,'Position',[0 1 13.33 7.5])

    %     pause(2) % Use this to wait 2 seconds before moving to next plot
    %     input('') % Use this to wait until enter key is pressed
end

close all;

%% Remove unnecessary variables.

clear time date lat lon alt sats pps
clear pps_adj pps_good time_adj time_fix time_good time_calc time_real dif
clear adj ans check gpsNames h m s rh rm i j

%% Determine the names of any existing Geophone files.

% First, get the file information for all files with .geo extension.
files = dir('*.geo');
% Then pull the names from the structure containing the file information.
geoNames = extractfield(files,'name')';

%% Using the GPS data printed at the top of each file, determine which hour
% interval each files falls into (e.g. find out which GPS data to use to
% convert micros to real time).

% First, pull the GPS data identifier from the top of each file.
geoID = cell(length(geoNames),1);
for i = 1:length(geoNames)
    fileID = fopen(geoNames{i});
    M = textscan(fileID,'%q');
    geoID{i} = M{1}{1};
    fclose(fileID);
end
```

```
% Then, match those identifiers to the GPS tags (the last GPS data
% recorded in each file).
tagLine = zeros(length(geoID),1);
for i = 1:length(tag)
    check = strcmp(geoID,tag{i});
    if any(check) == 1
        tagLine(check) = i;
    end
end

% Remove any files that don't have any matching GPS files.
check = tagLine == 0;
tagLine(check) = [];
geoNames = geoNames(~check);

%% Load the Analog Data and separate the data into the hour long segments
% that fall between the 2 min GPS data segments.

micros = cell(length(tag),1);
Ax = cell(length(tag),1);
Ay = cell(length(tag),1);
Az = cell(length(tag),1);

for i = 1:length(geoNames)
    M = csvread(geoNames{i},2,0);
    micros{tagLine(i)} = [micros{tagLine(i)}; M(:,1)];
    Ax{tagLine(i)} = [Ax{tagLine(i)}; M(:,2)];
    Ay{tagLine(i)} = [Ay{tagLine(i)}; M(:,3)];
    Az{tagLine(i)} = [Az{tagLine(i)}; M(:,4)];
end

%% Adjust recorded micros so time continually increases (resets every
% 4294967295 us). Each time micros resets (value of next micros is less
% than current micros) add an additional 4294967295 us to each micros
% value.

micros_adj = cell(length(micros),1);

for i = 1:length(micros)
    adj = 0;
    micros_adj{i} = micros{i};
    for j = 2:length(micros{i})
        if micros{i}(j) < micros{i}(j-1)
            adj = adj + 4294967295;
        end
        micros_adj{i}(j) = micros{i}(j) + adj;
    end
end

%% Use the linear trends to convert micros to GPS time

micros_real = cell(length(micros_adj),1);

for i = 1:length(micros_adj)
```

```
    micros_real{i} = polyval(line{i},micros_adj{i},S{i},mu{i});
end

%% Save the data.

save(dataName,'micros_real','Ax','Ay','Az')
```

Appendix J: Description of Electrical Resistivity Code

At the top of the code are several variables that control different settings that may be adjusted by the user, including the parameters that are needed by Res2Dinv at the top of the data file. Adjustable variables include the name of the data file, the name of the survey line, the smallest electrode spacing, the array type, and the number of decimals to record for the x-location, electrode spacing, n integer (Dipole-Dipole and Schlumberger arrays only), and apparent resistivity of each measurement. In addition, there are two variables that correspond to the duration of time the signal should be averaged over and the resistance of the resistor used to measure current. The duration of time over which the signal should be averaged is related to the frequency of the AC signal supplied to the current electrodes. For our purposes, we chose a duration corresponding to 20 cycles. The actual resistance of the resistor used to measure current, as opposed to the reported resistance, should be measured with a multimeter and adjusted in these settings as necessary. Other variables are used to define the pins used for various connections and the settings and addresses needed for the ADS1220 24-bit ADC shield. These variables and definitions are included at the top of the code to allow the user to easily adjust the code in one location, rather than having to change the same variable or definition multiple times throughout the code.

In addition, the necessary libraries are also included at the top of the code. This code requires the inclusion of two built-in libraries: the SD library for communication with and storing data on the microSD card and the SPI library for communication with and controlling the ADS1220 24-bit ADC shield.

The setup phase begins with the usual steps necessary to setup the different components of the system to work correctly: serial communication with a laptop computer is established, the

microSD card and ADS1220 24-bit ADC shield are initialized, the digital pins that will be used to control the multiplexers are set as outputs, the multiplexers are all closed, and the digital pin used to send the digital low signal through the multiplexers and turn on the relays is set as an output and set to low. The multiplexers are closed initially and always closed before the electrodes are switched to prevent the supplied current and ground lines from being connected directly to the same electrode and short circuiting the system. The multiplexers are opened or closed by setting the digital pins connected to the multiplexers' EN pins to low or high, respectively. To ensure the user knows whether the microSD card was successfully initialized, the system will print either "SD ERROR" or "SD READY" to the serial monitor of the connected laptop after initialization is attempted. The ADS1220 24-bit ADC shield is initialized using the same function described for the seismic array to start up and apply the necessary settings to the ADS1220 24-bit ADC shield. This concludes the setup necessary for the system.

Next, the system will print the name of the array type that it will run to the serial monitor of the connected laptop so the user can verify that the array type is correct. The system then runs one of three functions that corresponds to the array type selected in the settings. If an invalid array type was selected, the system will instead print "Invalid array configuration." to the serial monitor of the connected laptop. In each of the three functions, a data file is created, named according to the settings, and the header for the Res2Dinv file format is printed to the file. This includes the survey name, smallest electrode spacing, and array type selected in the settings, the number of data points corresponding to the electrode combinations included for that array (see Appendix L through Appendix N), and two zeros indicating that x-locations will be the location of the first electrode and that only resistivity data will be included. We decided to limit the code to resistivity data only to reduce the complexity necessary in the code. If desired, several data

files can be collected using the same array with different AC signal frequencies and reformatted manually after data collection is complete to create an IP data set. After the header is printed, the file is closed and the system begins to take electrical resistivity measurements.

For each electrical resistivity measurement, the system sets which electrodes to use as the current and potential electrodes (the electrode combinations in the appendix determine which electrodes are set as the current and potential electrodes for each measurement) and then opens the multiplexers by setting the various digital pins used to control the multiplexers LOW or HIGH. The digital pins connected to the multiplexers' S0, S1, S2, and S3 pins are used to control which channel the multiplexers' input is connected to. The combinations of LOW and HIGH values the S0, S1, S2, and S3 pins must be set to in order to connect to each channel can be found in Table J.1. Note that the multiplexer channel does not necessarily correspond directly to the same number electrode. The multiplexer channels are all offset by one to prevent short circuiting. To see which channels correspond to each electrode for the different multiplexers see Figure 6.1.1. The multiplexers are opened by setting the digital pin connected to the multiplexers' EN pin LOW.

Before continuing, the system pauses for 20 ms to ensure the relays are fully connected before proceeding. Then, to allow the signals to stabilize before beginning the measurement, the code waits for the duration corresponding to 20 cycles of the AC signal (specified in the settings) before continuing. The system then records the potential at four different locations for the next 20 cycles and records the voltages in storage arrays. These locations correspond to the potential immediately before and after the 100-Ω resistor, which can be used to calculate the current supplied to the subsurface, and the potential at the two potential electrodes, so the potential difference can be calculated. These voltages are determined using a function to pull the values

Table J.1: Combinations of LOW and HIGH values that the S0, S1, S2, and S3 pins must be set to in order to connect to each channel.

Channel	S0	S1	S2	S3
0	LOW	LOW	LOW	LOW
1	HIGH	LOW	LOW	LOW
2	LOW	HIGH	LOW	LOW
3	HIGH	HIGH	LOW	LOW
4	LOW	LOW	HIGH	LOW
5	HIGH	LOW	HIGH	LOW
6	LOW	HIGH	HIGH	LOW
7	HIGH	HIGH	HIGH	LOW
8	LOW	LOW	LOW	HIGH
9	HIGH	LOW	LOW	HIGH
10	LOW	HIGH	LOW	HIGH
11	HIGH	HIGH	LOW	HIGH
12	LOW	LOW	HIGH	HIGH
13	HIGH	LOW	HIGH	HIGH
14	LOW	HIGH	HIGH	HIGH
15	HIGH	HIGH	HIGH	HIGH

from the ADS1220 24-bit ADC shield that is nearly identical to the function described for the seismic array, however, with one additional step which converts the 32-bit analog value to a voltage. The system stores a new set of four potential values approximately every 2.15 ms, which is the fastest sampling rate the system is capable of. Once the specified recording duration is over, the system closes the multiplexers and then begins the process of calculating the apparent resistivity.

First, the system sums all the voltages for each of the four locations and divides by the total number of points to determine the mean. The system then subtracts the means from all of the recorded voltages to normalize the signals. Once the signals have been normalized, the system calculates the RMS voltage by summing the square of each of the normalized voltages and dividing by the total number of points. The system then uses the RMS voltages from before and after the 100-Ω resistor to calculate the current and the RMS voltages at the two potential

electrodes to calculate the potential difference. Finally, the system determines the x-location and spacing of the measurement and uses them, along with the current and potential difference, to calculate the apparent resistivity using the appropriate equation.

Once the apparent resistivity is calculated, the system prints a line to the data file recording the measurement. As shown in the data file format, this measurement includes the x-location, electrode spacing, n integer (Dipole-Dipole and Schlumberger arrays only), and apparent resistivity. At this point, the system prints a counter to the serial monitor of the connected laptop to indicate that data collection is continuing to occur. This also acts as a progress indicator which tells the user which measurement the system is currently on. In addition, the system pauses for another 20 ms to ensure the relays are fully closed before switching to the new electrodes and beginning the next measurement.

This concludes the steps the system takes for each measurement. After a measurement is complete, the system repeats this process with the next electrode combination in the series. Once all measurements for a series are complete, the system prints four additional lines of zeros at the end of the file which is required by the Res2Dinv software's data file format. Lastly, the system will print "Done." to the serial monitor of the connected laptop so the user knows that data collection is complete. The system will continue to wait after data collection is complete for the user to reupload the code before running again.

Appendix K: Electrical Resistivity Code

```
////////////////////////////////////////////////////////////////// Settings
//////////////////////////////////////////////////////////////////

// File Header Settings
char logfilename[13] = "wPc.dat"; // Name of the data file, 8.3 format: cccccccc.ccc (can have less than 8 characters before period)
char surveyname[51] = "Wenner Resistivity of Circuit"; // Name of the survey line, 50 character limit
float spacing = 0.02; // Electrode spacing (m), include at least one decimal
int arraytype = 1; // 1 = Wenner, 3 = Dipole-Dipole, 7 = Schlumberger

// Code Settings
int l = 1076397; // Duration of time to average data over (microseconds)
int R_ref = 98; // Resistance of resistor used to measure current (Ohms), determined using multimeter

// Number of Decimals to Print
int x_dec = 3; // Decimals for x-location
int s_dec = 3; // Decimals for electrode spacing
int n_dec = 4; // Decimals for n integer for Dipole-Dipole and Sclumberger Arrays
int R_dec = 4; // Decimals for resistivity

////////////////////////////////////////////////////////////////// Main Code
//////////////////////////////////////////////////////////////////

// Library for SD card
#include <SD.h>

// Library for Protocentral ADS1220 ADC
#include <SPI.h>

// Current Electrode (+,power)
#define C1_0 3
#define C1_1 2
#define C1_2 1
#define C1_3 0
#define C1_en 4

// Current Electrode (-,ground)
#define C2_0 8
#define C2_1 7
#define C2_2 6
#define C2_3 5
#define C2_en 24
```

```
// Potential Electrode (+)
#define P1_0 28
#define P1_1 27
#define P1_2 26
#define P1_3 25
#define P1_en 29

// Potential Electrode (-)
#define P2_0 33
#define P2_1 32
#define P2_2 31
#define P2_3 30
#define P2_en 34

// Signal pin to send to multiplexers to open relays
#define sig 35

// Protocentral ADS1220 ADC Definitions
// Teensy Pins
#define CS_PIN 10
#define DRDY_PIN 9
// Multiplexer Pins
#define C1_pin 0x80
#define C2_pin 0x90
#define P1_pin 0xA0
#define P2_pin 0xB0
// Scale
#define PGA 1 // Programmable Gain = 1
#define VREF 3.3 // Internal reference of 2.048V
#define VFSR VREF/PGA
#define FULL_SCALE   (((int32_t)1<<23)-1)
//Config registers
#define CONFIG_REG0_ADDRESS 0x00
#define CONFIG_REG1_ADDRESS 0x01
#define CONFIG_REG2_ADDRESS 0x02
#define CONFIG_REG3_ADDRESS 0x03
// Other
#define REG_CONFIG0_MUX_MASK 0xF0
#define SPI_MASTER_DUMMY 0xFF
#define RESET 0x06 //Send the RESET command (06h) to make sure the ADS1220 is properly
reset after power-up
#define START 0x08 //Send the START/SYNC command (08h) to start converting in
continuous conversion mode
#define WREG 0x40
```

```
// Protocentral ADS1220 ADC Settings Variables
uint8_t m_config_reg0 = 0x00; //Default settings: AINP=AIN0, AINN=AIN1, Gain 1, PGA enabled
uint8_t m_config_reg1 = 0xD0; //Default settings: DR=2000 SPS, Mode=Turbo, Conv mode=single-shot, Temp Sensor disabled, Current Source off
uint8_t m_config_reg2 = 0xC0; //Default settings: Vref Analog Supply, No 50/60Hz rejection, power open, IDAC off
uint8_t m_config_reg3 = 0x00; //Default settings: IDAC1 disabled, IDAC2 disabled, DRDY pin only

void setup() {
  // Begin serial communication.
  Serial.begin(9600);
  delay(2000);

  // Set up Teensy's built-in SD card
  if (!SD.begin(BUILTIN_SDCARD)) {
    Serial.println("SD ERROR");
    while (1) { // If there was an error, uses an infinite loop to stop the code from proceeding
    }
  }
  else {
    Serial.println("SD READY");
  }

  // Begin the Protocentral ADS1220 ADC
  ADS1220_Start();

  // Set all digital pins needed to control the four multiplexers as outputs.
  pinMode(C1_0, OUTPUT);
  pinMode(C1_1, OUTPUT);
  pinMode(C1_2, OUTPUT);
  pinMode(C1_3, OUTPUT);
  pinMode(C2_0, OUTPUT);
  pinMode(C2_1, OUTPUT);
  pinMode(C2_2, OUTPUT);
  pinMode(C2_3, OUTPUT);
  pinMode(P1_0, OUTPUT);
  pinMode(P1_1, OUTPUT);
  pinMode(P1_2, OUTPUT);
  pinMode(P1_3, OUTPUT);
  pinMode(P2_0, OUTPUT);
  pinMode(P2_1, OUTPUT);
  pinMode(P2_2, OUTPUT);
  pinMode(P2_3, OUTPUT);
  pinMode(C1_en, OUTPUT);
```

```
  pinMode(C2_en, OUTPUT);
  pinMode(P1_en, OUTPUT);
  pinMode(P2_en, OUTPUT);

  // Turn all multiplexers off.
  digitalWrite(C1_en, HIGH);
  digitalWrite(C2_en, HIGH);
  digitalWrite(P1_en, HIGH);
  digitalWrite(P2_en, HIGH);

  // Set up the signal pin, when this signal is sent to the relays it will open them
  pinMode(sig, OUTPUT);
  digitalWrite(sig, LOW);

  // Run the correct array configuration
  if (arraytype == 1) {
    Serial.println("Wenner:");
    wenner();
    Serial.println();
  }
  else if (arraytype == 3) {
    Serial.println("Dipole-Dipole:");
    dipole_dipole();
    Serial.println();
  }
  else if (arraytype == 7) {
    Serial.println("Schlumberger:");
    schlumberger();
    Serial.println();
  }
  else {
    Serial.println("Invalid array configuration.");
  }
}

void loop() {
}

void wenner() {
  // Create file and print headers
  File file = SD.open(logfilename, FILE_WRITE);
  if (file) {
    file.println(surveyname);
    file.println(spacing);
    file.println(arraytype);
    file.println("35"); // Total number of data points (for 16 electrodes)
```

```
  file.println("0"); // X-location begins at location of 1st electrode
  file.println("0"); // Resistivity only, no IP (can be reformatting later if desired)
  file.close();
}

int n = 1;

for (int j = 1; j < 6; j++) {
  for (int i = 0; i < (16 - (3 * j)); i++) {

    // Set the correct electrodes
    set_C1(i);
    set_C2(i + 3 * j);
    set_P1(i + j);
    set_P2(i + 2 * j);

    // Turn the electrodes on
    digitalWrite(C1_en, LOW);
    digitalWrite(C2_en, LOW);
    digitalWrite(P1_en, LOW);
    digitalWrite(P2_en, LOW);

    // Wait to make sure relays are opened
    delay(20);

    int p = 0;
    float C1[600];
    float C2[600];
    float P1[600];
    float P2[600];
    elapsedMicros t = 0;

    while (t <= l) { // Wait for signals to stabilize
    }
    t = 0;

    while (t <= l) {
      // Take a measurement
      C1[p] = ADS1220_Read(C1_pin);
      C2[p] = ADS1220_Read(C2_pin);
      P1[p] = ADS1220_Read(P1_pin);
      P2[p] = ADS1220_Read(P2_pin);

      p = p + 1;
    }
```

```
// Turn the electrodes off
digitalWrite(C1_en, HIGH);
digitalWrite(C2_en, HIGH);
digitalWrite(P1_en, HIGH);
digitalWrite(P2_en, HIGH);

// Calculate the mean voltages
float C1_sum = 0;
float C2_sum = 0;
float P1_sum = 0;
float P2_sum = 0;
for (int e = 0; e < p; e++) {
  C1_sum = C1_sum + C1[e];
  C2_sum = C2_sum + C2[e];
  P1_sum = P1_sum + P1[e];
  P2_sum = P2_sum + P2[e];
}
float C1_mean = C1_sum / p;
float C2_mean = C2_sum / p;
float P1_mean = P1_sum / p;
float P2_mean = P2_sum / p;

// Normalize the signals by subtracting the mean
float C1_n[600];
float C2_n[600];
float P1_n[600];
float P2_n[600];
for (int e = 0; e < p; e++) {
  C1_n[e] = C1[e] - C1_mean;
  C2_n[e] = C2[e] - C2_mean;
  P1_n[e] = P1[e] - P1_mean;
  P2_n[e] = P2[e] - P2_mean;
}

// Sum the squares of all the voltages at each electrode
float C1_sum_sq = 0;
float C2_sum_sq = 0;
float P1_sum_sq = 0;
float P2_sum_sq = 0;
for (int e = 0; e < p; e++) {
  C1_sum_sq = C1_sum_sq + sq(C1_n[e]);
  C2_sum_sq = C2_sum_sq + sq(C2_n[e]);
  P1_sum_sq = P1_sum_sq + sq(P1_n[e]);
  P2_sum_sq = P2_sum_sq + sq(P2_n[e]);
}
```

```
      // Divide by total number of points to get Vrms of each electrode
      float C1_rms = sqrt(C1_sum_sq) / p;
      float C2_rms = sqrt(C2_sum_sq) / p;
      float P1_rms = sqrt(P1_sum_sq) / p;
      float P2_rms = sqrt(P2_sum_sq) / p;

      // Calculate Resistivity
      float I = abs(C1_rms - C2_rms) / R_ref; // Current supplied to the current electrodes
(determined using a known 1000 Ohm resistor).
      float V = abs(P1_rms - P2_rms); // Measured voltage from the potential electrodes.
      float x = i * spacing; // X-location (m)
      float s = j * spacing; // Spacing (m)
      float R = 2 * PI * s * V / I; //Apparent Resistivity
      //      float R = V / I;

      // Record the measurement
      file = SD.open(logfilename, FILE_WRITE);
      if (file) {
        file.print(x, x_dec);
        file.print('\t');
        file.print(s, s_dec);
        file.print('\t');
        file.println(R, R_dec);
        file.close();
      }

      Serial.println(n);
      n = n + 1;

      // Wait to make sure relays are closed
      delay(20);
    }
  }
  file = SD.open(logfilename, FILE_WRITE);
  if (file) {
    file.println("0");
    file.println("0");
    file.println("0");
    file.println("0");
    file.close();
  }
  Serial.println("Done.");
}

void dipole_dipole() {
  // Create file and print headers
```

```
File file = SD.open(logfilename, FILE_WRITE);
if (file) {
  file.println(surveyname);
  file.println(spacing);
  file.println(arraytype);
  file.println("157"); // Total number of data points (for 16 electrodes)
  file.println("0"); // X-location begins at location of 1st electrode
  file.println("0"); // Resistivity only, no IP (can be reformatting later if desired)
  file.close();
}

int n = 1;

for (int k = 1; k < 3; k++) {
  for (int j = 0; j < (15 - k * 2); j++) {
    for (int i = 1; (j + 2 * k + i) < 16; i++) {

      // Set the correct electrodes
      set_C1(j);
      set_C2(j + k);
      set_P1(j + k + i);
      set_P2(j + 2 * k + i);

      // Turn the electrodes on
      digitalWrite(C1_en, LOW);
      digitalWrite(C2_en, LOW);
      digitalWrite(P1_en, LOW);
      digitalWrite(P2_en, LOW);

      // Wait to make sure relays are opened
      delay(20);

      int p = 0;
      float C1[600];
      float C2[600];
      float P1[600];
      float P2[600];
      elapsedMicros t = 0;

      while (t <= l) { // Wait for signals to stabilize
      }
      t = 0;

      while (t <= l) {
        // Take a measurement
        C1[p] = ADS1220_Read(C1_pin);
```

```
  C2[p] = ADS1220_Read(C2_pin);
  P1[p] = ADS1220_Read(P1_pin);
  P2[p] = ADS1220_Read(P2_pin);

  p = p + 1;
}

// Turn the electrodes off
digitalWrite(C1_en, HIGH);
digitalWrite(C2_en, HIGH);
digitalWrite(P1_en, HIGH);
digitalWrite(P2_en, HIGH);

// Calculate the mean voltages
float C1_sum = 0;
float C2_sum = 0;
float P1_sum = 0;
float P2_sum = 0;
for (int e = 0; e < p; e++) {
  C1_sum = C1_sum + C1[e];
  C2_sum = C2_sum + C2[e];
  P1_sum = P1_sum + P1[e];
  P2_sum = P2_sum + P2[e];
}
float C1_mean = C1_sum / p;
float C2_mean = C2_sum / p;
float P1_mean = P1_sum / p;
float P2_mean = P2_sum / p;

// Normalize the signals by subtracting the mean
float C1_n[600];
float C2_n[600];
float P1_n[600];
float P2_n[600];
for (int e = 0; e < p; e++) {
  C1_n[e] = C1[e] - C1_mean;
  C2_n[e] = C2[e] - C2_mean;
  P1_n[e] = P1[e] - P1_mean;
  P2_n[e] = P2[e] - P2_mean;
}

// Sum the squares of all the voltages at each electrode
float C1_sum_sq = 0;
float C2_sum_sq = 0;
float P1_sum_sq = 0;
float P2_sum_sq = 0;
```

```
      for (int e = 0; e < p; e++) {
        C1_sum_sq = C1_sum_sq + sq(C1_n[e]);
        C2_sum_sq = C2_sum_sq + sq(C2_n[e]);
        P1_sum_sq = P1_sum_sq + sq(P1_n[e]);
        P2_sum_sq = P2_sum_sq + sq(P2_n[e]);
      }

      // Divide by total number of points to get Vrms of each electrode
      float C1_rms = sqrt(C1_sum_sq) / p;
      float C2_rms = sqrt(C2_sum_sq) / p;
      float P1_rms = sqrt(P1_sum_sq) / p;
      float P2_rms = sqrt(P2_sum_sq) / p;

      // Calculate Resistivity
      float I = abs(C1_rms - C2_rms) / R_ref; // Current supplied to the current electrodes
(determined using a known 1000 Ohm resistor).
      float V = abs(P1_rms - P2_rms); // Measured voltage from the potential electrodes.
      float x = j * spacing; // X-location (m)
      float s = k * spacing; // Spacing (m)
      float m = (float)i / (float)k; // Spacing m*a between electrode pairs (already used n)
      float R = PI * s * m * (m + 1) * (m + 2) * V / I; // Apparent Resistivity
      //       float R = V / I;

      // Record the measurement
      file = SD.open(logfilename, FILE_WRITE);
      if (file) {
        file.print(x, x_dec);
        file.print('\t');
        file.print(s, s_dec);
        file.print('\t');
        file.print(m, n_dec);
        file.print('\t');
        file.println(R, R_dec);
        file.close();
      }

      Serial.println(n);
      n = n + 1;

      // Wait to make sure relays are closed
      delay(20);
    }
  }
}
file = SD.open(logfilename, FILE_WRITE);
if (file) {
```

```
    file.println("0");
    file.println("0");
    file.println("0");
    file.println("0");
    file.close();
  }
  Serial.println("Done.");
}

void schlumberger() {
  // Create file and print headers
  File file = SD.open(logfilename, FILE_WRITE);
  if (file) {
    file.println(surveyname);
    file.println(spacing);
    file.println(arraytype);
    file.println("91"); // Total number of data points (for 16 electrodes)
    file.println("0"); // X-location begins at location of 1st electrode
    file.println("0"); // Resistivity only, no IP (can be reformatting later if desired)
    file.close();
  }

  int n = 1;

  for (int k = 1; k < 3; k++) {
    for (int j = 0; j < (14 - k); j ++) {
      for (int i = 1; (j + 2 * i + k) < 16; i ++) {

        // Set the correct electrodes
        set_C1(j);
        set_C2(j + 2 * i + k);
        set_P1(j + i);
        set_P2(j + i + k);

        // Turn the electrodes on
        digitalWrite(C1_en, LOW);
        digitalWrite(C2_en, LOW);
        digitalWrite(P1_en, LOW);
        digitalWrite(P2_en, LOW);

        // Wait to make sure relays are opened
        delay(20);

        int p = 0;
        float C1[600];
        float C2[600];
```

```
float P1[600];
float P2[600];
elapsedMicros t = 0;

while (t <= l) { // Wait for signals to stabilize
}
t = 0;

while (t <= l) {
  // Take a measurement
  C1[p] = ADS1220_Read(C1_pin);
  C2[p] = ADS1220_Read(C2_pin);
  P1[p] = ADS1220_Read(P1_pin);
  P2[p] = ADS1220_Read(P2_pin);

  p = p + 1;
}

// Turn the electrodes off
digitalWrite(C1_en, HIGH);
digitalWrite(C2_en, HIGH);
digitalWrite(P1_en, HIGH);
digitalWrite(P2_en, HIGH);

// Calculate the mean voltages
float C1_sum = 0;
float C2_sum = 0;
float P1_sum = 0;
float P2_sum = 0;
for (int e = 0; e < p; e++) {
  C1_sum = C1_sum + C1[e];
  C2_sum = C2_sum + C2[e];
  P1_sum = P1_sum + P1[e];
  P2_sum = P2_sum + P2[e];
}
float C1_mean = C1_sum / p;
float C2_mean = C2_sum / p;
float P1_mean = P1_sum / p;
float P2_mean = P2_sum / p;

// Normalize the signals by subtracting the mean
float C1_n[600];
float C2_n[600];
float P1_n[600];
float P2_n[600];
for (int e = 0; e < p; e++) {
```

```
  C1_n[e] = C1[e] - C1_mean;
  C2_n[e] = C2[e] - C2_mean;
  P1_n[e] = P1[e] - P1_mean;
  P2_n[e] = P2[e] - P2_mean;
 }

 // Sum the squares of all the voltages at each electrode
 float C1_sum_sq = 0;
 float C2_sum_sq = 0;
 float P1_sum_sq = 0;
 float P2_sum_sq = 0;
 for (int e = 0; e < p; e++) {
  C1_sum_sq = C1_sum_sq + sq(C1_n[e]);
  C2_sum_sq = C2_sum_sq + sq(C2_n[e]);
  P1_sum_sq = P1_sum_sq + sq(P1_n[e]);
  P2_sum_sq = P2_sum_sq + sq(P2_n[e]);
 }

 // Divide by total number of points to get Vrms of each electrode
 float C1_rms = sqrt(C1_sum_sq) / p;
 float C2_rms = sqrt(C2_sum_sq) / p;
 float P1_rms = sqrt(P1_sum_sq) / p;
 float P2_rms = sqrt(P2_sum_sq) / p;

 // Calculate Resistivity
 float I = abs(C1_rms - C2_rms) / R_ref; // Current supplied to the current electrodes
(determined using a known 1000 Ohm resistor).
 float V = abs(P1_rms - P2_rms); // Measured voltage from the potential electrodes.
 float x = j * spacing; // X-location (m)
 float s = k * spacing; // Spacing (m)
 float m = (float)i / (float)k; // Spacing m*a between current and potential electrodes (already
used n)
 float R = PI * m * (m + 1) * s * V / I; //Apparent Resistivity
 //       float R = V / I;

 // Record the measurement
 file = SD.open(logfilename, FILE_WRITE);
 if (file) {
  file.print(x, x_dec);
  file.print('\t');
  file.print(s, s_dec);
  file.print('\t');
  file.print(m, n_dec);
  file.print('\t');
  file.println(R, R_dec);
  file.close();
```

```
        }

        Serial.println(n);
        n = n + 1;

        // Wait to make sure relays are closed
        delay(20);
      }
    }
  }
  file = SD.open(logfilename, FILE_WRITE);
  if (file) {
    file.println("0");
    file.println("0");
    file.println("0");
    file.println("0");
    file.close();
  }
  Serial.println("Done.");
}

// Functions to set multiplexer pins.

void set_C1(int channel) {
  int controlPin[] = {C1_0, C1_1, C1_2, C1_3};
  int muxChannel[16][4] = {
    {0, 0, 0, 0}, //electrode 0 - channel 0
    {1, 0, 0, 0}, //electrode 1 - channel 1
    {0, 1, 0, 0}, //electrode 2 - channel 2
    {1, 1, 0, 0}, //electrode 3 - channel 3
    {0, 0, 1, 0}, //electrode 4 - channel 4
    {1, 0, 1, 0}, //electrode 5 - channel 5
    {0, 1, 1, 0}, //electrode 6 - channel 6
    {1, 1, 1, 0}, //electrode 7 - channel 7
    {0, 0, 0, 1}, //electrode 8 - channel 8
    {1, 0, 0, 1}, //electrode 9 - channel 9
    {0, 1, 0, 1}, //electrode 10 - channel 10
    {1, 1, 0, 1}, //electrode 11 - channel 11
    {0, 0, 1, 1}, //electrode 12 - channel 12
    {1, 0, 1, 1}, //electrode 13 - channel 13
    {0, 1, 1, 1}, //electrode 14 - channel 14
    {1, 1, 1, 1}  //electrode 15 - channel 15
  };
  //loop through the 4 sig
  for (int i = 0; i < 4; i ++) {
    digitalWrite(controlPin[i], muxChannel[channel][i]);
```

```
  }
}

void set_C2(int channel) {
  int controlPin[] = {C2_0, C2_1, C2_2, C2_3};
  int muxChannel[16][4] = {
    {1, 1, 1, 1}, //electrode 0 - channel 15
    {0, 0, 0, 0}, //electrode 1 - channel 0
    {1, 0, 0, 0}, //electrode 2 - channel 1
    {0, 1, 0, 0}, //electrode 3 - channel 2
    {1, 1, 0, 0}, //electrode 4 - channel 3
    {0, 0, 1, 0}, //electrode 5 - channel 4
    {1, 0, 1, 0}, //electrode 6 - channel 5
    {0, 1, 1, 0}, //electrode 7 - channel 6
    {1, 1, 1, 0}, //electrode 8 - channel 7
    {0, 0, 0, 1}, //electrode 9 - channel 8
    {1, 0, 0, 1}, //electrode 10 - channel 9
    {0, 1, 0, 1}, //electrode 11 - channel 10
    {1, 1, 0, 1}, //electrode 12 - channel 11
    {0, 0, 1, 1}, //electrode 13 - channel 12
    {1, 0, 1, 1}, //electrode 14 - channel 13
    {0, 1, 1, 1}  //electrode 15 - channel 14
  };
  //loop through the 4 sig
  for (int i = 0; i < 4; i ++) {
    digitalWrite(controlPin[i], muxChannel[channel][i]);
  }
}

void set_P1(int channel) {
  int controlPin[] = {P1_0, P1_1, P1_2, P1_3};
  int muxChannel[16][4] = {
    {0, 1, 1, 1}, //electrode 0 - channel 14
    {1, 1, 1, 1}, //electrode 1 - channel 15
    {0, 0, 0, 0}, //electrode 2 - channel 0
    {1, 0, 0, 0}, //electrode 3 - channel 1
    {0, 1, 0, 0}, //electrode 4 - channel 2
    {1, 1, 0, 0}, //electrode 5 - channel 3
    {0, 0, 1, 0}, //electrode 6 - channel 4
    {1, 0, 1, 0}, //electrode 7 - channel 5
    {0, 1, 1, 0}, //electrode 8 - channel 6
    {1, 1, 1, 0}, //electrode 9 - channel 7
    {0, 0, 0, 1}, //electrode 10 - channel 8
    {1, 0, 0, 1}, //electrode 11 - channel 9
    {0, 1, 0, 1}, //electrode 12 - channel 10
    {1, 1, 0, 1}, //electrode 13 - channel 11
```

```
    {0, 0, 1, 1}, //electrode 14 - channel 12
    {1, 0, 1, 1}  //electrode 15 - channel 13
  };
  //loop through the 4 sig
  for (int i = 0; i < 4; i ++) {
    digitalWrite(controlPin[i], muxChannel[channel][i]);
  }
}

void set_P2(int channel) {
  int controlPin[] = {P2_0, P2_1, P2_2, P2_3};
  int muxChannel[16][4] = {
    {1, 0, 1, 1}, //electrode 0 - channel 13
    {0, 1, 1, 1}, //electrode 1 - channel 14
    {1, 1, 1, 1}, //electrode 2 - channel 15
    {0, 0, 0, 0}, //electrode 3 - channel 0
    {1, 0, 0, 0}, //electrode 4 - channel 1
    {0, 1, 0, 0}, //electrode 5 - channel 2
    {1, 1, 0, 0}, //electrode 6 - channel 3
    {0, 0, 1, 0}, //electrode 7 - channel 4
    {1, 0, 1, 0}, //electrode 8 - channel 5
    {0, 1, 1, 0}, //electrode 9 - channel 6
    {1, 1, 1, 0}, //electrode 10 - channel 7
    {0, 0, 0, 1}, //electrode 11 - channel 8
    {1, 0, 0, 1}, //electrode 12 - channel 9
    {0, 1, 0, 1}, //electrode 13 - channel 10
    {1, 1, 0, 1}, //electrode 14 - channel 11
    {0, 0, 1, 1}  //electrode 15 - channel 12
  };
  //loop through the 4 sig
  for (int i = 0; i < 4; i ++) {
    digitalWrite(controlPin[i], muxChannel[channel][i]);
  }
}

void ADS1220_Start() {
  // Start ADS1220
  pinMode(CS_PIN, OUTPUT);
  pinMode(DRDY_PIN, INPUT);
  SPI.begin();
  SPI.setBitOrder(MSBFIRST);
  SPI.setDataMode(SPI_MODE1);
  // Reset
  delay(100);
  digitalWrite(CS_PIN, LOW);
  delay(2);
```

```
  digitalWrite(CS_PIN, HIGH);
  delay(2);
  digitalWrite(CS_PIN, LOW);
  SPI.transfer(RESET);
  digitalWrite(CS_PIN, HIGH);
  delay(100);
  // Set Configuration Register 0 to Selected Settings
  digitalWrite(CS_PIN, LOW);
  SPI.transfer(WREG | (CONFIG_REG0_ADDRESS << 2));
  SPI.transfer(m_config_reg0);
  digitalWrite(CS_PIN, HIGH);
  delay(100);
  // Set Configuration Register 1 to Selected Settings
  digitalWrite(CS_PIN, LOW);
  SPI.transfer(WREG | (CONFIG_REG1_ADDRESS << 2));
  SPI.transfer(m_config_reg1);
  digitalWrite(CS_PIN, HIGH);
  delay(100);
  // Set Configuration Register 2 to Selected Settings
  digitalWrite(CS_PIN, LOW);
  SPI.transfer(WREG | (CONFIG_REG2_ADDRESS << 2));
  SPI.transfer(m_config_reg2);
  digitalWrite(CS_PIN, HIGH);
  delay(100);
  // Set Configuration Register 3 to Selected Settings
  digitalWrite(CS_PIN, LOW);
  SPI.transfer(WREG | (CONFIG_REG3_ADDRESS << 2));
  SPI.transfer(m_config_reg3);
  digitalWrite(CS_PIN, HIGH);
  delay(100);
}

float ADS1220_Read(uint8_t Mux_Pin) {
  static byte SPI_Buff[3];
  int32_t adc_data = 0;
  int32_t bit24;
  // Select Channel
  m_config_reg0 &= ~REG_CONFIG0_MUX_MASK;
  m_config_reg0 |= Mux_Pin;
  digitalWrite(CS_PIN, LOW);
  SPI.transfer(WREG | (CONFIG_REG0_ADDRESS << 2));
  SPI.transfer(m_config_reg0);
  digitalWrite(CS_PIN, HIGH);
  // Start Conversion
  digitalWrite(CS_PIN, LOW);
  SPI.transfer(START);
```

```
  digitalWrite(CS_PIN, HIGH);
  // Wait for Conversion to Complete
  while (digitalRead(DRDY_PIN) == HIGH) {}
  // Get the Converted Data
  digitalWrite(CS_PIN, LOW);
  for (int i = 0; i < 3; i++) {
    SPI_Buff[i] = SPI.transfer(SPI_MASTER_DUMMY);
  }
  digitalWrite(CS_PIN, HIGH);
  bit24 = SPI_Buff[0];
  bit24 = (bit24 << 8) | SPI_Buff[1];
  bit24 = (bit24 << 8) | SPI_Buff[2];
  bit24 = (bit24 << 8);
  adc_data = (bit24 >> 8);

  float Vout = (float)((adc_data * VFSR) / FULL_SCALE * 2 + 6.56);

  return Vout;
}
```

Appendix L: Electrode Combinations Included for Wenner Array.

a spacing	C1	P1	P2	C2
1	0	1	2	3
	1	2	3	4
	2	3	4	5
	3	4	5	6
	4	5	6	7
	5	6	7	8
	6	7	8	9
	7	8	9	10
	8	9	10	11
	9	10	11	12
	10	11	12	13
	11	12	13	14
	12	13	14	15
2	0	2	4	6
	1	3	5	7
	2	4	6	8
	3	5	7	9
	4	6	8	10
	5	7	9	11
	6	8	10	12
	7	9	11	13
	8	10	12	14
	9	11	13	15
3	0	3	6	9
	1	4	7	10
	2	5	8	11
	3	6	9	12
	4	7	10	13
	5	8	11	14
	6	9	12	15
4	0	4	8	12
	1	5	9	13
	2	6	10	14
	3	7	11	15
5	0	5	10	15

Appendix M: Electrode Combinations Included for Dipole-Dipole Array

a spacing	n separation	C1	C2	P1	P2
1	1	0	1	2	3
		1	2	3	4
		2	3	4	5
		3	4	5	6
		4	5	6	7
		5	6	7	8
		6	7	8	9
		7	8	9	10
		8	9	10	11
		9	10	11	12
		10	11	12	13
		11	12	13	14
		12	13	14	15
	2	0	1	3	4
		1	2	4	5
		2	3	5	6
		3	4	6	7
		4	5	7	8
		5	6	8	9
		6	7	9	10
		7	8	10	11
		8	9	11	12
		9	10	12	13
		10	11	13	14
		11	12	14	15
	3	0	1	4	5
		1	2	5	6
		2	3	6	7
		3	4	7	8
		4	5	8	9
		5	6	9	10
		6	7	10	11
		7	8	11	12
		8	9	12	13
		9	10	13	14
		10	11	14	15
	4	0	1	5	6
		1	2	6	7
		2	3	7	8
		3	4	8	9
		4	5	9	10
		5	6	10	11

		6	7	11	12
		7	8	12	13
	4	8	9	13	14
		9	10	14	15
		0	1	6	7
		1	2	7	8
		2	3	8	9
		3	4	9	10
	5	4	5	10	11
		5	6	11	12
		6	7	12	13
		7	8	13	14
		8	9	14	15
		0	1	7	8
		1	2	8	9
		2	3	9	10
		3	4	10	11
	6	4	5	11	12
		5	6	12	13
		6	7	13	14
		7	8	14	15
		0	1	8	9
		1	2	9	10
1		2	3	10	11
	7	3	4	11	12
		4	5	12	13
		5	6	13	14
		6	7	14	15
		0	1	9	10
		1	2	10	11
		2	3	11	12
	8	3	4	12	13
		4	5	13	14
		5	6	14	15
		0	1	10	11
		1	2	11	12
	9	2	3	12	13
		3	4	13	14
		4	5	14	15
		0	1	11	12
	10	1	2	12	13
		2	3	13	14
		3	4	14	15
		0	1	12	13
	11	1	2	13	14
		2	3	14	15

1	12	0	1	13	14
	12	1	2	14	15
	13	0	1	14	15
2	0.5	0	2	3	5
		1	3	4	6
		2	4	5	7
		3	5	6	8
		4	6	7	9
		5	7	8	10
		6	8	9	11
		7	9	10	12
		8	10	11	13
		9	11	12	14
		10	12	13	15
	1	0	2	4	6
		1	3	5	7
		2	4	6	8
		3	5	7	9
		4	6	8	10
		5	7	9	11
		6	8	10	12
		7	9	11	13
		8	10	12	14
		9	11	13	15
	1.5	0	2	5	7
		1	3	6	8
		2	4	7	9
		3	5	8	10
		4	6	9	11
		5	7	10	12
		6	8	11	13
		7	9	12	14
		8	10	13	15
	2	0	2	6	8
		1	3	7	9
		2	4	8	10
		3	5	9	11
		4	6	10	12
		5	7	11	13
		6	8	12	14
		7	9	13	15
	2.5	0	2	7	9
		1	3	8	10
		2	4	9	11
		3	5	10	12
		4	6	11	13

		5	7	12	14
	2.5	6	8	13	15
		0	2	8	10
		1	3	9	11
	3	2	4	10	12
		3	5	11	13
		4	6	12	14
		5	7	13	15
		0	2	9	11
		1	3	10	12
	3.5	2	4	11	13
		3	5	12	14
2		4	6	13	15
		0	2	10	12
		1	3	11	13
	4	2	4	12	14
		3	5	13	15
		0	2	11	13
	4.5	1	3	12	14
		2	4	13	15
	5	0	2	12	14
		1	3	13	15
	5.5	0	2	13	15

Appendix N: Electrode Combinations Included for Schlumberger Array

a spacing	n separation	C1	P1	P2	C2
1	1	0	1	2	3
		1	2	3	4
		2	3	4	5
		3	4	5	6
		4	5	6	7
		5	6	7	8
		6	7	8	9
		7	8	9	10
		8	9	10	11
		9	10	11	12
		10	11	12	13
		11	12	13	14
		12	13	14	15
	2	0	2	3	5
		1	3	4	6
		2	4	5	7
		3	5	6	8
		4	6	7	9
		5	7	8	10
		6	8	9	11
		7	9	10	12
		8	10	11	13
		9	11	12	14
		10	12	13	15
	3	0	3	4	7
		1	4	5	8
		2	5	6	9
		3	6	7	10
		4	7	8	11
		5	8	9	12
		6	9	10	13
		7	10	11	14
		8	11	12	15
	4	0	4	5	9
		1	5	6	10
		2	6	7	11
		3	7	8	12
		4	8	9	13
		5	9	10	14
		6	10	11	15
	5	0	5	6	11
		1	6	7	12

1	5	2	7	8	13
		3	8	9	14
		4	9	10	15
	6	0	6	7	13
		1	7	8	14
		2	8	9	15
	7	0	7	8	15
2	0.5	0	1	3	4
		1	2	4	5
		2	3	5	6
		3	4	6	7
		4	5	7	8
		5	6	8	9
		6	7	9	10
		7	8	10	11
		8	9	11	12
		9	10	12	13
		10	11	13	14
		11	12	14	15
	1	0	2	4	6
		1	3	5	7
		2	4	6	8
		3	5	7	9
		4	6	8	10
		5	7	9	11
		6	8	10	12
		7	9	11	13
		8	10	12	14
		9	11	13	15
	1.5	0	3	5	8
		1	4	6	9
		2	5	7	10
		3	6	8	11
		4	7	9	12
		5	8	10	13
		6	9	11	14
		7	10	12	15
	2	0	4	6	10
		1	5	7	11
		2	6	8	12
		3	7	9	13
		4	8	10	14
		5	9	11	15
	2.5	0	5	7	12
		1	6	8	13
		2	7	9	14

		3	8	10	15
2	3	0	6	8	14
		1	7	9	15