Mobile Waterway Monitor

Kody Stribrny

Computer Systems Engineering

Arizona State University

Tempe, AZ 85253, USA

April 14, 2017

Chapter 1 Project Introduction

Chapter 1.1 Introduction

Collecting water quality measurements is critical to environmental protection. Shifts in pH value and other water quality characteristics not only affects plant life, but also wildlife populations and water usage. Abnormal pH values affect how pollutants are absorbed into water. Highly acidic or basic water will kill normally present wildlife. Invasive species can be better suited to abnormal acidity which further exacerbates the problem of wildlife management. Areas such as the Southwestern United States face a larger problem. The dramatically reduced water supply of the southwest means that managing wildlife water supplies are just as critical as managing drinkable water. Most states like Arizona, Nevada, and California use large canals which stretch for hundreds of miles through low desert valleys to transport water to their populations. These canals, like rivers, are susceptible to pollution even while under constant supervision. To make problems worse, a river, or canal, is not a perfectly mixed solution. Introducing a single monitoring point can miss downstream trends and areas of increased pollution. Pockets of heightened pollutants can exist, especially in the case of a pollutant leeching into the water at a specific point, throughout the entire length of a waterway. Traditional water quality measurements have been inherently static, focusing on a single point along the water channel [1].

The Waterway Monitor was inspired from these loopholes in waterway quality testing. Instead of collecting water data at a single point the Mobile Waterway Monitor (MWM) aims to collect environmental data from flowing water in an unexplored way. The device flows with the current collecting data at a predefined interval. Collecting data along the length of the river instead of a single point creates granularity amongst data points. These precise measurement points can reflect waterway trends on a smaller scale than previously possible. The waterway monitor is a

standalone system which can be used to augment the capabilities of already installed water management systems to provide extremely accurate data. The MWM is both small and buoyant allowing the device to be successfully set adrift in the river. The small size of the waterway monitor reduces snags along weedy rivers. Coordinate specific data points allow for river trends to be seen in regions of a river instead of broadly implying average values for a whole river in the traditional static data collection approach.

The device also provides more detailed and real-time results than manual measurements. Manually testing water quality on fast moving rivers is dangerous for water scientist and land management workers. Flooding after large rains, spring ice melts, or water releases from upstream dams are typically difficult to monitor because of the unpredictable flow and water level. The erratic behavior represents a danger to scientists who work in these areas. The Mobile Waterway Monitor does not require human interaction outside of the initial release and retrieval of the device. The abilities of the Mobile Waterway Monitor are unique to the measurement systems. This thesis presents a new monitoring solution to flowing water.

Chapter 1.2 Problem Importance

The Waterway Monitor can help society in its challenge in preserving nature. Nature preserves and National Parks, both of which are open to the public and cheap to visit, were once a uniquely American establishment. National Parks are unlike man-made urban parks which offer no more than a small playground and a grass field with a small picnic table. Instead, America's often underused and underfunded federally protected national parks showcase naturally occurring spectacles. My original goal was to create something which helps monitor the effects of mankind on these government nature preserves. A new device needs to have several traits to see park service use. Cost is a significant issue to the park service whose budget has not grown to offset park maintenance costs. The Mobile Waterway Monitor is a cheap tool for river monitoring, but it does not have to be restricted to just flowing water. Monitoring bodies of water like lakes, pools, or ponds can also be completed by this prototype. Monitoring water quality affects many areas of the national park system including drinkable water, protection of endangered animals, as well as sediment and erosion. Keeping parks and monuments from being destroyed by pollutants or keeping locally occurring, endangered wildlife healthy is as important as a goal to the National Park Service as providing public access to the parks themselves. National Parks like the Grand Canyon would not have nearly as many activities like whitewater rafting, fishing, or hiking if the mighty Colorado River was highly acidic. The Mobile Waterway Monitor can help the Park Service monitor several key characteristics of rivers like the Colorado River at an affordable cost. Reducing monitoring costs will affect our society by preserving national parks for generations to come, ensuring public access to American treasures. Flood effects can also be measured by government agencies using the MWM. Direction, water content, and speed are all critical factors in emergency flood response. By not relying on land-based infrastructure such as electricity,

telephone lines, or internet, the Waterway Monitor will remain operational during harsh conditions unlike stationary monitors. The monitor's flexibility lies in its low cost and independence to meet public water sensing needs.

Not all benefits of the Mobile Waterway Monitor are found in the public sector. The cheap cost and open source upbringing of the MWM allow individuals the ability to monitor private water sources with more precision than before. Making the MWM as cheap as possible was a goal established to create project value within public services, but the low cost MWM can be used for several home uses like monitoring a fish pond or pool. With only cellular infrastructure demands the MWM is easy to begin using for home owners and government agencies alike. The simple architecture and expandable capabilities allow individual users to customize the Mobile Waterway Monitor to their needs. These needs can include additional measurements like temperature, oxygen content, or salinity.

With such a large list of measurable water characteristics, an important question to ask is: Does pH mean anything? Yes, pH is a fundamental measurement of chemical solutions. Acidity affects many characteristics of water. For instance, pollutant solubility can increase with heightened pH. Aquatic wildlife can be killed if the water which they live in rises above or below their pH threshold. Varying pH does not always have to cause issues. Changes in pH can stem from different plant life or increases in nearby human activity, both recreational and industrial. Pollutants which change river pH can both be directly released into the waterway or can be a product of increased airborne pollution produced by factories or automotive vehicles. This might sound like a strange occurrence in the United States because the Environmental Protection Agency exists, but high pollution is being discovered in many American rivers.

A perfect example of increased pollution content is Detroit's Rouge river which has been found to be one of the most polluted waterways in the United States per one US Geological Survey report summarized by the Detroit Free Press [2]. Central to Detroit's once dominant automotive industry, the Rouge River was found to contain pollutants such as automotive exhaust compound, tar, and detergents. The study mentions how these chemicals will disrupt the hormone production in fish. Austin Baldwin, the hydrologist who led the study also stated that the chemicals "can act like the hormone estrogen and lead to reproductive disruption, including causing male fish to turn into females". These same chemical compounds wreak havoc on smaller fish who are killed off in large numbers. The large fish, who survive by consuming smaller fish, will then see a population decrease as their food supply dwindles away.. These chemicals undoubtedly change the pH content of the river. Changes in the pH value due to pollution could also spell disaster. Swinging to a more basic or acidic water will damage plant life along the river. If the water becomes far too basic or acidic it could simply mean the death of all aquatic wildlife. The infographic below shows the pH ranges of water dependent life forms. Collecting data from the entire Rouge River would allow water scientists the ability to pinpoint problem locations and take preventative measures to predict aquatic wildlife before starvation or irreparable environmental damage occurs.

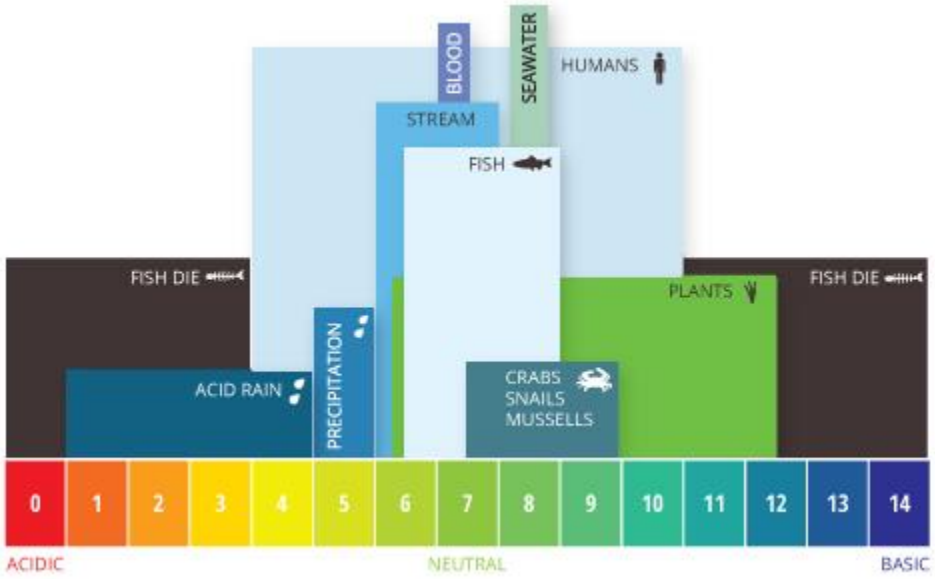


Figure 1: Ecological pH Range [3]

Fluctuating water characteristics due to pollution has a snowball effect which, if not properly handled, will further degrade waterways. One cumulative characteristic is pollutant solubility. Unbalanced water pH will accept pollutants at a much faster rate than in balanced water. Pollutants are accepted into the unbalanced solution in the form of a chemical reaction, which often reduces the extreme pH back towards the center point, with a pH value of 7. In the process of a reaction occurring, a byproduct is produced. This side effect can damage wildlife as significantly as in the unbalanced water. Pollution can also increase waterborne bacteria. Increased algae growth is a common reaction to increased nutrients. Algae is dependent on oxygen from the water it is in. Increased algae due to pollution can additionally strip more oxygen from the water which will disrupt animal and plant life. Surface algae reduces water clarity and sunlight penetration. The reduced sunlight at the river bottom further destroys naturally occurring plant life.

Water quality, at all depths, is highly important. Traits like acidity, clarity, and naturally occurring lifeforms can be reduced or decimated by fluctuations. These fluctuations are disastrous

not only to nature but also to humans who use the water. Industries like fishing, tourism, and boating depend on clean surface water. Ignoring water quality can cost not only financial damage, but irreparable damage to nature.

Chapter 1.3 Scope

The Mobile Waterway Monitor aligns with mobility in the classic mobility-firepower tradeoff. This duality between mobility and firepower suggests that lightweight solutions achieve mobility by reducing capability. Obviously, the small stature of the MWM dictates that every sensor imaginable will not fit; however, the majority of significant characteristics can be measured by a just a few sensors. The MWM's mobile footprint both allows for additional measurements as well as important sensors which read crucial waterway characteristics, minimizing any capability downsides. Taking commonplace measurements like pH value, the unit's data collection capabilities are different because of how they are taken, not by the data type recorded. The MWM records data points along a river. Following the path of a river, the MWM sets a new paradigm in water monitoring. Previous measuring systems are stationary and often require supporting infrastructure. Infrastructures like on grid electrical power and roads for easy maintenance. The MWM does away with these traits and instead turns to battery and solar power. Energy independence is not the only differentiating feature. Siding with mobility, the Waterway Monitor's small form factor leads to versatility. The unit can be used in both rivers or standing water like lakes and ponds. Easy to transport and fitting in nearly any water sources, the MWM can take water quality measurements nearly anywhere.

Water quality measurements that the current MWM records include information like GPS coordinates, elevation, and water pH value. The monitor supports, but does not implement, several other measurements. For example, sensors that record temperature, oxygen content, and salinity are available. Adding these sensors to the Mobile Waterway Monitor would remove much of the firepower tradeoff mentioned previously. Appending sensors is a simple process due to the open source software and hardware throughout the Mobile Waterway Monitor.

Duplication and modification are two hallmarks of the open source and the system building communities. However, most other water monitoring systems do not take advantage of the large number of tinkerers who only want to improve products. Creating the Mobile Waterway Monitor from easily purchased parts ensures that anyone with a passion for water quality monitoring will be able to tweak the MWM implementation to fit their needs. By providing open source code and documentation the Mobile Waterway Monitor is more likely to see growth than closed source equivalents. Open source growth can lead to both additional features or direction changes. One possible direction change for the Mobile Waterway Monitor is features which would expand monitoring capabilities to placid waters like lakes. Some examples of stagnant water where a modified Mobile Waterway Monitor would be a powerful tool include monitoring pool water for pH, reporting on ammonia content in fish ponds, or studying nutrient changes in a hydroponics reservoir as the suns position changes. Instead of trying to implement all possibilities, I have tried to select some of the most universally important parameters.

The Mobile Waterway Monitor features a web backend which is unlike any water quality system. After recording the current battery voltage, the pH value of the water, and the GPS location, the node offloads this data using a cellular connection to a backend server. The server, hosted by Adafruit [21], stores and displays the data in real time. Graphs, maps, and data streams represent only a fraction of the visualization capabilities of the sever. From here the data can be additionally processed through a Python dashboard to display calculated values like speed and pH trends.

The US Geological Service effectively sets the standard for water monitoring. Years of knowledge paired up with practical experience in monitoring waterways means that a lot can be gleaned from the USGS. The government agency collects waterway data in two distinct ways

[4][5]. The first is through long-term stations which are constructed in a permanent location near the surface water. Manual collection is the second, and likely less preferred, method for collection. Long-term stations are extremely powerful, containing dozens of sensors. Stationary setups can process their own data before transmission, and generally, have a reliable and dedicated communication channel. Base stations do require constant power. There are a handful of downsides for long-term stations, but the most important downside is the static location. Permanent stations can only monitor a single point of a river. It is completely possible that upstream of the station is more acidic water which is being masked by an equally basic tributary. Downstream the same effects can occur. Manual testing can avoid these issues if testing is completed by a thorough water scientist who walks the entire length of a river while using either an electronic probe to measure values or collecting water samples. In either case, the water scientist must traverse the river bed to take measurements.

In either case, the Mobile Waterway Monitor augments traditional methods to produce real-time and more accurate water quality measurements. The MWM avoids the single location issue by traversing the entire river length while collecting data. By traversing the entire length, the Waterway Monitor ensures that strange water trends are recorded. Permanent stations are costly and with one in place along a river, a second long-term station downstream is unrealistic. By complementing the stationary outpost with a Mobile Waterway Monitor many of the same water trend can be discovered for a fraction of the cost of a second long-term station. In the event that a base station was destroyed by flooding, the MWM will be able to cover the duties of the damaged station. The Waterway Monitor also augments manual testing. Manual testing is inherently intrusive to wildlife. Deploying the MWM is safer for both wildlife and water scientists. Deploying the MWM node will also mitigate the intrusion on nature by water quality testing. The MWM is

also less costly for park services as a river path can be traversed by an automated system freeing up park rangers for other duties.

Thanks to more powerful and energy efficient computer designs a recent explosion of internet connected controllers have taken place. Revolutionizing concepts like home automation and network capabilities, the Internet of Things (IoT) tidal wave has even affected automated data collection. IoT advancements are allowing technologies and users to take a more active role in various, often ignored, aspects in life. Services like home security, which was traditionally provided by companies, can now be implemented by a handful of sensors, cameras, and a website. The same trends hold for scientific devices. In the remainder of this Chapter, I will cover what I have found to be the state of Internet-connected, semi-mobile waterway monitoring in more recent years. Three projects will be covered.

The first case study, "IoT pH flow controls and automation with Costa Farms" [6] written by Blain Barton, details an experiment conducted by Costa Farms and Microsoft. Costa Farms, an agriculture business focusing on typical house and fruit-bearing plants, needed the ability to monitor pH in their hydroponics system remotely and automatically. Around the clock, autonomous data collection replaced the need for an employee to manually capture and analyze water samples. Costa Farms partnered with Microsoft to work out a solution. The solution was a Wi-Fi enabled microcontroller, recording data from a connected pH probe, much like the probe used in the Mobile Waterway Monitor. The microcontroller was connected to a Microsoft server which would both store data and send updating texts to a mobile device. Blain Barton's article is published in November 2016, only a handful of months before the publication of this thesis. Adopting IoT solutions for businesses is an emerging trend which many companies are racing to profit from. In house IoT solutions such as the Amazon Echo, a product reordering button, and

business to business solutions like the Costa Farms/Microsoft pairing will become increasingly common. Barton's article helps show the current state of IoT development in industry, and his coverage of the Costa Farms solution highlights important similarities and differences between the Mobile Waterway Monitor and the Microsoft implementation. Mobility is the most polarizing difference. The Microsoft solution for Costa Farms was not mobile. For hydroponic systems, recording pH data somewhere along the inlet is fine, but for the MWM this solution does not make sense. Recording from a single point reproduces current river monitors capabilities. Additionally, removing mobility from design constraints also removes the problem of power efficiency and uptime. The Mobile Waterway Monitor must ration power from the battery and solar panel input as sparingly as possible. Microsoft's solution was powered directly from an outlet to constant power. These different characteristics are not surprising considering the unique usage scenarios. What is similar between the Mobile Waterway Monitor and the Costa Farms monitor is that both devices report measurements wirelessly to a centralized server for permanent data storage. Cost was also very minimal for Microsoft's solutions, like the Waterway Monitor. Websites are supported for both the MWM and the Microsoft solution which hosts real time data.

Industry also fostered another connected solution to waterway pH monitoring. Sensorex's SAM-1 [7] is a smartphone adapted pH, oxygen, or purity sensor. All three sensors are discrete and must be purchased separately. With the initial kit price of around $250 and additional probes ranging from $70 to $140, the SAM-1 kit does not come cheap. Relying upon a connected smartphone running either Google's Android or Apple's IOS operating system, the probe is not truly independent. Connecting through a standard audio jack, the SAM-1 transmits through the custom smartphone app which packages data in an email. Sharing data between multiple people requires forwarding the autogenerated email. Other differences include the need for a physically

present phone, sensor, and person who is to operate both devices. Sensorex's solution seems more pseudo-automated than truly autonomous. As mentioned earlier, having a water scientist traversing fast moving water is dangerous. The Mobile Waterway Monitor enhances safety by being independent and not requiring human control while traversing waterways. Cost is also prohibitive for the SAM-1. With a starting price of $250 and with additional cost for sensors averaging at $100, the MWM drastically undercuts the Sensorex product while providing more capability. Not every design choice is radically different though. For instance, both the Sensorex design and the Mobile Waterway Monitor utilize the cellular network to transmit data. Using an omnipresent service like mobile networks ensures that a communication channel can always be established, even if traditional infrastructure is not available. The SAM-1, as well as the MWM, face size and weight constraints in order to be portable, and both are implemented within these constraints. Extensibility is also a concern of both designs. The Waterway Monitor is extensible through additional, unpopulated pins on the Arduino board. The MWM's highly modifiable, and open source, code make expansion cheap and easy. Sensorex instead created different adapters which their proprietary mobile app detects.

Mobile applications and IoT devices show the current commercial directions of water monitoring. For Costa Farms, the ability to monitor acidity can directly affect plant yield, which in turn affects profits. Having a stationary, connected monitor which tests incoming water can be the difference between thousands of dollars in revenue. The Sensorex SAM-1 is a more open ended product which allows the user to monitor pH both using live data displayed on their phone or transmitted data. The Mobile Waterway Monitor can take some design cues from these projects like having multiple connection interfaces (SAM-1) or a more capable server which handles more types of messaging (Feather IoT device). Both of these solutions share many similarities with the

Mobile Waterway Monitor but they do not accurately reflect the state of academic research into waterway monitoring. Academia largely gave birth to IoT research and colleges across the country could benefit by crossing the paths of water monitoring with connected devices.

Enter the University of California Berkeley Floating Sensor Network (FSN) project [8]. The FSN was discovered late after creating the Mobile Waterway Monitor. A collaborative effort between several Berkeley graduate students, the FSN is comprised of many 'drifters'. These drifters are analogous to the Waterway Monitor. There have been three evolution of the drifters. The current generation has capabilities that include cellular communication, pH sensing, and GPS capabilities. The Mobile Waterway Monitor shares all three of these characteristics with the drifter. Additional capabilities such as short wave radio communications, salinity testing, GPS directed movement, and the ability to dive distinguish the individual drifter from the MWM. Several drifters also have the capability to construct a mesh network, hence the name Floating Sensor Network, to foster more efficient communication and ensure optimal coverage of a river. Started around 2007, the FSN has gone through several revisions from a simple phone in a waterproof case to the third-generation drifter which contains the features mentioned above. The project website has not featured an update since 2012, and I believe that many of the team members likely graduated from the PhD program. Project progress seems to have halted.

Berkeley's stopped Floating Sensor Network shares several similarities with the Mobile Waterway Monitor. Both projects focus on essentially the same project scope. The FSN was focused on emergency water monitoring and extended to non-emergency sensing. Derived backwards, the MWM was created to help with normal day to day water quality testing. The side effect that the Water Monitor is emergency capable stems from project scope. Buoyancy is common between both devices which float downriver. Communicating data over cellular

connection is common between both projects. Full autonomy is a characteristic of both the drifter and the MWM river node. Battery life for both devices is sufficiently long with drifters able to run 24 hours before recharging and the river node capable of around 14 hours. Many of the measurement types are overlapping such as pH, GPS location, and battery life. The FSN and MWM both report data to a centralized server which supports live data visualization. Simply put, both the Mobile Waterway Monitor and the Floating Sensor Network are very similar.

Similar, but not the same, the MWM and FSN have a number of differences. Solar power generation is unique to the Mobile Waterway Monitor. Because of the vertical orientation of the FSN drifter, mounting a capable solar panel is impossible. The boat-like shape of the MWM allows for a sizable solar panel to both protect electronics from direct sunlight and recharge the battery. Lacking solar energy, the FSN requires a much larger battery than the MWM to maintain similar run time. At a staggering 17 inches tall, the FSN drifter towers over the 3-inch-tall MWM node. Shallower floating depth will allow the MWM to traverse more waterways. The FSN is more advanced than the current state of the MWM. Mesh networking capabilities, controlled movement and diving, and additional sensors are backed up by software which has been developed over the course of 5 years. The MWM has not had this much development time. Overall, the UC Berkeley Floating Sensor Network is a great solution to the scope of mobile water quality testing and shows how capable a future Mobile Waterway Monitor could be with further research.

Chapter 2 System Components

Chapter 2.1 System Overview

So, what exactly is the Mobile Waterway Monitor? The MWM is a three-stage system which consists of a floating node, server, and dashboard which work together to produce near real-time waterway data. The waterway node represents much of this project goals, and because of this, will be the first stage covered.

The MWM node is the fundamental motivator behind this entire project. This small, floating system will (see below) was designed to float down rivers while collecting both pH and positional data. Data is then transmitted to a receiving server to be processed in near real-time. The node has supporting hardware to record pH, transmit cellular data, and harvest solar energy through a solar panel. Specification and usage of these major components will be covered in the following section.

Figure 2: Mobile Waterway Monitor Node

Software enables all the Mobile Waterway Monitor components to work properly. Software such as the Arduino sketch and libraries power the MWM node as well as Python files which power a dashboard. With an Arduino at heart, the Waterway Monitor node uses several libraries and files to enable cellular data, pH collection, and positional data. Several of the protocols used by the project, including the Arduino sketch, will be described in following sections. The Adafruit server and website require no programming which has saved a tremendous amount of effort from being diverted to server functions. Python, and several its wonderful packages, provide dashboard functionality on a user level. The dashboard provides a simple user interface which allows plotting of data in near real-time. All three of these components must be connected to properly function. Luckily, the path is a one-way street with data originating at the node and flowing to the server and Python dashboard. Directionality makes the program both simpler to run and use than other architectures. The Mobile Waterway Monitor software architecture is shown below and showcases the directional data flow.

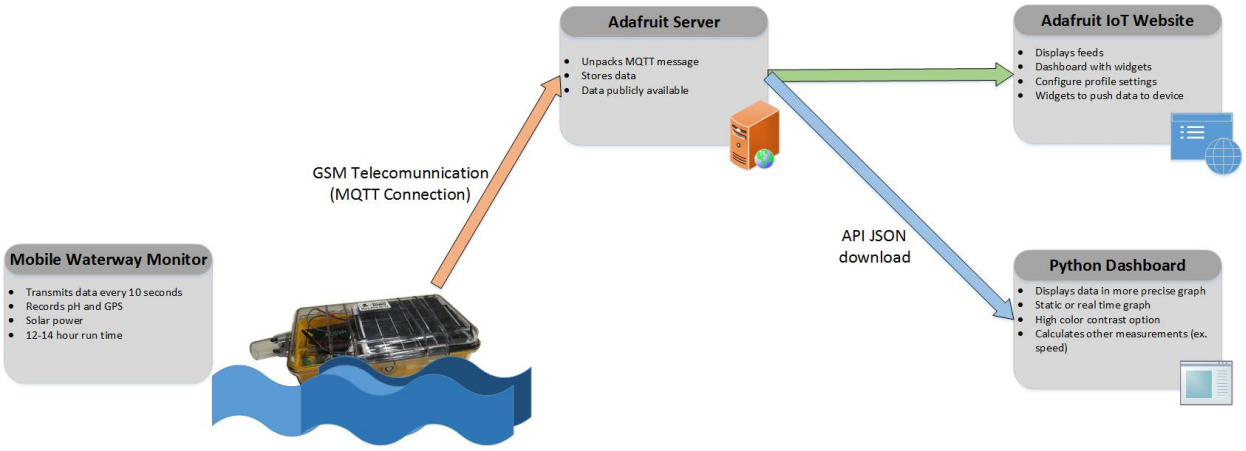


Figure 3: Mobile Waterway Monitor Software Architecture

In summary, the monitoring node uses its collection of sensors to gather measurements and transmit data to a receiving server. This server, which stores data, is accessible through a website as well as the Python dashboard. The Python dashboard and website both aim to

visualize waterway recordings for users. Along this route, which is oversimplified, the Mobile Waterway Monitor uses several software protocols and hardware features to make the daunting journey possible. Later chapters of this paper will cover this journey in greater detail. Sections will showcase a demonstration, results from the demonstration, software design, and communication schemes as well as hardware components of the waterway node. Node hardware is discussed in the following section in greater detail.

Chapter 2.2 Hardware Components

The Mobile Waterway Monitor consists of several significant components. In this chapter, I will first introduce a table which lists the major components, cost, and usage. I will omit trivial detail such as the amount of wiring used or solder. After the table I will then discuss each component in more detail. A parts list is provided in the Appendix with links to where each part was purchased and further documentation resources for each component. Assembly information can also be found in the document appendix.

Major System Components

Part	Cost (USD)	Use	Vendor
Arduino Uno R3	$24.94	Microcontroller	Amazon
Adafruit FONA 808	$49.95	Cellular communication and GPS coordinate receiver	Adafruit
Gravity: Analog pH Sensor	$56.95	Records pH data	DFRobot
Pelican 1060 (yellow/clear)	$20.71	Encloses project	CaseTech
USB DC Solar Charger	$17.50	Converts solar power to useable power which can be diverted to battery	Adafruit
Lithium Ion Battery	$29.50	Holds charge	Adafruit
6V, 2W Solar Panel	$29.00	Converts solar energy to usable power	Adafruit

Arduino Uno:

The Arduino Uno (R3) is central to this project. As the main microcontroller, the Uno facilitates communication between expansion boards. The Uno is equipped with analog and digital

inputs as well as digital and PWM outputs. Using a common microprocessor, the ATmega328P, lends several features to the Arduino. Sporting a RISC architecture, the UNO is both powerful (at 131 instructions) and fast, with most instructions executing in a single cycle. The chip has 32KB of self-programmable flash program memory, 2KB of SRAM, and 1KB of EEPROM. The clock speed is 16Hz The Uno has 14 input/output pins. All of which are digital and operate at 5 volts. The pins operate at a maximum of 5V and can provide 40mA of current. PWM (8-bit) is supported on 6 of the pins. Other supported pin features include SPI, TWI, and even I2C communication. The Arduino can both be powered by a USB from a computer when programming or a 2.1mm power jack or even a battery, if directly wired in. The physical (2D) size of the Uno is about the dimensions of a credit card. The Arduino website reports that the board weighs 25 grams. The Arduino's power doesn't exclusively lie in the hardware, instead comes with a minimal, but expandable, software suite.

Programming the Arduino is largest strength of the otherwise weak computer. The Arduino IDE is both free and intuitive. Extensive library support for the Uno make adding sensors and features streamlined. The Arduino coding language takes cues from C++ and allows for compatible C++ libraries to be used. The Arduino programming platform is based off the Wiring programming platform. These characteristics combine to make programming the Arduino easy to learn for beginners and powerful for long time programmers.

Adafruit FONA 808:

Adafruit's FONA 808 breakout board handles the cellular communication of the MWM. The breakout features both GSM cellular communication and GPS tracking. The board supports

voice, text and SMS, and cellular data. The FONA uses a SIM808 chip which can handle quad-band (850/900/1800/1900MHz) transmission. The FONA requires a SIM card, and T-Mobile is the recommended provider in the United States as the GSM network. The key ability that the river monitor takes advantage of is the GPRS data (cellular data). Having cellular data allows transmission of data through the MQTT protocol, which will be discussed in the following section, which is designed for Internet-of-Things projects. The MT3337 chipset provides GPS which is quick to establish a fix and accurate. GPS is accurate to approximately 2.5 meters which will make a clear outline of any flowing waterway. The GPS chip has 22 tracking and 66 acquisition channels and has tracking sensitivity of -165 dBm. Both GSM and GPS antennas are connected over their own designated uFL ports. The board does require power to be supplied through a JST 2-pin connection or MicroUSB. The physical size of the board is 1.7 inches' square.

Controlling the breakout is simple thanks to Adafruit's FONA library. This library contains all the functions required for voice, text, and data. Additional libraries exist to build features off the FONA board such as an MQTT library.

Gravity: Analog pH Probe:

Monitoring pH is just one of the possible measurements for the waterway monitor. It is one of the more important measurements in water quality. That is why a pH probe was selected to be the primary measurement for this project. The Haoshi H-101 pH electrode is mated with a small DFRobot PCB which accepts the probe's coaxial connection. The probe electrode is housed in a glass membrane, which has a low impedance for accurate readings. Four plastic prongs protect the glass electrode. The pH sensor lays in the waterway case parallel to the length of the case with

only the tip sticking out to take measurements. This configuration should protect the sensor from most damage when floating down river. Haoshi's pH meter is accurate to +/-0.1pH. The output voltage of the probe is linear, which is helpful when debugging. The measuring range is 0-14pH, the full spectrum, and the response time is significantly less than 1 minute. The meter is large, with a diameter of slightly larger than ¾ of an inch. The electrode itself is around 6 inches long, so it does end up taking up a significant amount of the small enclosure. The DFRobot PCB is 1.7*1.26 inches. The MWM uses the PCB's gain adjustment potentiometer to calibrate pH readings to save additional space for future software and data. Both software and hardware calibration instructions are available. A bright blue power LED lights up when the PCB and meter are powered.

Pelican 1060:

The clear polycarbonate Pelican 1060 has been ideal for this project. Highly rugged and watertight, the case meets the IP67 standard which guarantees 1 meter deep submersion for 30 minutes without issue. Several variants of this case are available. These variants come in solid colors or the clear polycarbonate case with a different colored rubber liner. The combination decided upon for this project was clear polycarbonate with a yellow rubber liner. The clear top allows the solar panel to reside inside of the case without reducing effectiveness and the yellow liner is highly visible, especially in the water or down a river. Having an eye-catching case reduces the likelihood of the device being lost. The interior dimensions are 8.25" long by 4.25" wide by 2.25" deep. Interior volume is reduced slightly when accounting for the rubber liner. Having a large amount of interior volume easily allows the solar panel, Arduino, power management, and communications hardware to be safely stored inside. The outer dimensions are 9.37 x 5.56 x 2.62 inches (length x width x height). The overall small form factor will allow the case to slip down a

river with ease. The case is buoyant up to 2.49 pounds which is more than enough for the contents. It is also stable within a temperature range of -10°F to +199°F.

USB DC Solar Charger

Formally titled "USB / DC / Solar Lithium Ion/Polymer charger - v2" by Adafruit, the little board manages the charging of the battery pack with powering the Arduino, breakout boards, and sensors. The board is designed to draw as much current from the solar panel as possible to power the load before resorting to the battery. Most often the solar panel doesn't produce enough current to power the complete device. Based on this setup, it is best to describe the solar power as augmenting the battery. In the off case that the solar panel produces more than enough current to power the circuitry, the battery will be charged with the remaining power. The solar panel inputs to the solar charge controller though a 2.1mm jack and the battery connects through a JST 2-pin connection. The load also uses a JST 2-pin connection. The included 4700µF capacitor is used to buffer the input voltage. This board requires a 6V solar panel. Charge rate for this controller is variable. The standard rate, as set from the factory, is 500mA as the maximum charge rate. This can be adjusted between down to 50mA and up to 1A by soldering on a resistor. I did not elect to change the charge rate as the battery pack itself is not recommended to charge at more than 1 Ampere. For a more power efficient future revision, soldering in the resistor which allows for a 1A current would be ideal. Increasing the charging current would allow for quicker charging times and less waster energy. Modifying the charge rate would need to be done only for a larger solar panel, as the one used in this project only provides a peak current of 378mA. The solar charge controller is buffered by a voltage regulator because of the high input and output voltages of the

charge controller. This charge controller can output the same voltage as the solar panel (6V) which could damage the Arduino.

Lithium Ion Battery

A 6600mAh, +3.7V battery pack was selected for this project. Purposely below the Arduino's voltage needs, to ensure no circuitry damage, the lithium ion battery must be paired up with a voltage regulator to boost input voltage. Conveniently, the same voltage regulator which was used for the solar charge controller covers to the battery. Dual coverage is achieved because the battery's terminals are connected to the charge controller to provide power when solar input is not enough. Maximum current draw for this battery pack is 3.3A. The recommended constant current draw is around 1.3 amps. Protection circuitry connects each of the three balanced 2200mAh cells. The battery's protection circuitry safeguards against over-voltage, over-current, and under-voltage conditions. Thermal protection is not included. Overheating dangers are mitigated as the case will be partially submerged in cool water. Thankfully, water transfers heat at a much quicker rate than air so the enclosure should always be much cooler than the contained air. The battery connects through a JST 2-pin connector. Some battery modification was needed for this project. Both the voltage regulator that power the Arduino and the FONA require a JST connection to a power supply. Two JST connections were wired in parallel to each other to allow for both components to draw from the same power source. No further modifications were needed.

Solar Panel (6V, 2W)

The final major component is the solar panel. The power rating is 2 Watts. The wattage is provided at a rate of 6V, 330mA. Monocrystalline cells provide an overall panel efficiency of 19%. The solar cells are encased inside of a plastic composite which strengthens the panels significantly. Plastic potting makes the panels waterproof, scratch resistant, and UV resistant. The dimensions of this cell are 5.4 inches long by 4.4 inches wide. Size was an important consideration, as a panel this small will fit entirely in the lid of the Pelican case. Being contained inside of the case allows the panel to be out of the way of most the components while still being completely protected from damage when floating down river. It is a tight fit and the panel does need to be angled slightly to fit. Angling the solar panel slightly will not create any problems. The panel weighs in at 3 ounces which hardly impacts the overall buoyancy of the Mobile Waterway Monitor.

Chapter 2.3 Software Protocols

The backbone of the Mobile Waterway Monitor's communication is two software standards – GSM and MQTT. Global System for Mobile Communications, or GSM, is a cellular communication scheme which handles transporting the data from the node to the server. MQ Telemetry Transport, also known as MQTT, wraps messages sent across the network. GSM is the most common cellular system worldwide and MQTT has taken off in the world of Arduinos and connected systems which only benefits users. This section will cover some background on GSM and MQTT protocols and usage.

As described on its Wikipedia page [9], GSM is a second-generation wireless communication standard which was expanded to include data. Data is handled through GPRS, or General Packet Radio Services. Initially defined in 1987 by the European Telecommunications Standards Institute, GSM cellular networks are now the most common network type available. Some estimates, one of which is cited by Wikipedia, has the network scheme holding 90% of the market share. This market share is defined globally. Typically, GSM providers issue SIM cards to identify network subscribers. The Mobile Waterway Monitor requires an activated SIM card to communicate. Relying upon a widespread technology allow the MWM to be used globally where GSM networks can be found. Being so common, GSM networks are highly standardized which means connected devices typically require no configuration.

The MWM was one of the lucky devices which didn't need programming. Requiring no GSM configuration to use, the Mobile Waterway Monitor can quickly establish its connection. In usage, I have found knowing the AT commands is immensely helpful when debugging. All AT network commands are shown when debugging is enabled on the Arduino sketch. Fortunately, understanding the MWM network state only requires the knowledge of a few AT commands.

Below is a capture of the Arduino serial monitor. This interface can only be seen when the Uno is connected to a computer by the USB B type connection. The serial monitor requires physical connection between the computer and the Uno. Network AT commands will be displayed on the serial monitor throughout execution. Several sources documenting the AT commands and their meaning are available, however the handiest has been SIMCom's AT command manual for the SIM800 chip. The SIM800 chip is used in the FONA 808 board so the commands correspond perfectly. The commonality of the SIM800 chip also benefits the MWM through the implementation of MQTT libraries.

```
Disabling GPRS
        ---> AT+CIPSHUT
        <--- SHUT OK
        ---> AT+SAPBR=0,1
        <--- +CME ERROR: operation not allowed
Enabling GPRS
        ---> AT+CIPSHUT
        <--- SHUT OK
        ---> AT+CGATT=1
        <--- OK
        ---> AT+SAPBR=3,1,"CONTYPE","GPRS"
        <--- OK
        ---> AT+SAPBR=3,1,"APN","epc.tmobile.com"
        <--- OK
        ---> AT+CSTT="epc.tmobile.com"
        <--- OK
        ---> AT+SAPBR=1,1
        <--- OK
        ---> AT+CIICR
        <--- OK
Connected to Cellular!
        ---> AT+CIPSHUT
        <--- SHUT OK
        ---> AT+CIPMUX=0
        <--- OK
        ---> AT+CIPRXGET=1
        <--- OK
AT+CIPSTART1="TCP","io.adafruit.com","1883"
        <--- OK
        <--- CONNECT OK
        ---> AT+CIPSTATUS
        <--- OK
        <--- STATE: CONNECT OK
AT+CIPSEND=61
 0x10 0x3B 0x0 0x4 0x4D 0x51 0x54 0x54 0x4 0xFFFFFFC2 0x1 0x2C 0x0 0x0 0x0 0xB 0x73 0x70 0x65 0x63 0x69 0x61 0x6C 0x4B 0x6F 0x64 0x79 0x0 0x20 0x37 0x32 0x65 0x65 0x39 0x36 0x38 0x37 0x63
        <--- >
        <--- SEND OK
        ---> AT+CIPSTATUS
        <--- OK
        <--- STATE: CONNECT OK
        ---> AT+CIPSTATUS
        <--- OK
        <--- STATE: CONNECT OK
        ---> AT+CIPRXGET=4
        <--- +CIPRXGET: 4,0
0 bytes available
        ---> AT+CIPSTATUS
        <--- OK
        <--- STATE: CONNECT OK
        ---> AT+CIPRXGET=4
        <--- +CIPRXGET: 4,0
```

Figure 4: Mobile Waterway Monitor Serial Monitor Output

MQTT, the common name for Message Queue Telemetry Transport, focuses on creating a reliable messaging protocol from unreliable, constricted networks [10]. While GSM is reliable

across the world, when the river node goes sailing down a stream the jerky movement paired up with shallow water dives create a difficult setting for guaranteed transmission. Lightweight and recommended as the interface to communicate with both Adafruit's and Sparkfun's IOT servers, using the MQTT protocol is an obvious choice for the Mobile Waterway Monitor. MQTT builds off the TCP/IP stack. Running above the TCP layer, helps abstract away many of TCP's many confusing acknowledgments sequences. MQTT is also flexible enough to be ported successfully to boards such as the ZigBee series, which use radio communication instead of the TCP stack.

Adafruit has maintained a MQTT library [11] for the FONA series of products as well as the CC3000 based products. This well-maintained library helped curb production time significantly. Both the FONA library and MQTT library can be installed through the Arduino IDE to make importation even easier. This library is described more in detail in the following section. This section will focus on how the MQTT transmission is handled.

Sending a message using the MQTT protocol is a straightforward series of steps. These steps can be broken up into two parts of the code: the setup and the loop. The setup is concerned with establishing the MWM-server connection. The running loop handles sending messages to the server. In the current implementation, the main loop never closes the MQTT connection to the Adafruit server. Keeping an open connection was chosen because of the relatively small time interval between transmissions as well as the high-power cost of establishing a connection. The steps used in the MWM setup code are shown below. They will be described in more detail afterwards.

```
1.      Adafruit_FONA fona = Adafruit_FONA(FONA_RST);
2.      Adafruit_MQTT_FONA   mqtt(&fona,   AIO_SERVER,   AIO_SERVERPORT,
        AIO_USERNAME, AIO_KEY);
3.      Adafruit_MQTT_Publish             location_ph_feed             =
        Adafruit_MQTT_Publish(&mqtt,       AIO_USERNAME        "/feeds/"
        LOCATION_PH_FEED_NAME "/csv");
```

```
4.      Adafruit_MQTT_Publish battery_feed = Adafruit_MQTT_Publish(&mqtt,
        AIO_USERNAME "/feeds/battery");
5.      fona.begin(fonaSS)
6.      fona.getNetworkStatus()
7.      fona.enableGPS(true)
8.      fona.setGPRSNetworkSettings(F(FONA_APN));
9.      fona.enableGPRS(false);
10.     fona.enableGPRS(true);
11.     int8_t ret = mqtt.connect();
```

It might seem a little strange to start discussing sending MQTT data with a FONA call, however, the MQTT is highly dependent on the FONA library [12]. The MQTT library simply wraps many of the lower level FONA calls while appending the required MQTT header information. Consisting of mostly Adafruit_FONA library calls, the setup establishes the MQTT connection. Step one of the MWM connection setup is declaring a FONA object. The instance, named *fona*, must be initialized with a defined reset pin. From here the MQTT FONA object can be initialized. The *mqtt* object must point to a FONA object instance, the server destination, the server port, along with an Adafruit IO username and key. The server is simply "io.adafruit.com" while the port for incoming MQTT communication is 1883. The username and key is specific to the users account. Transmission calls, shown later, require the *mqtt* instance. Steps three and four are almost identical. These steps setup the data feeds to be pushed to. These steps require a MQTT instance reference and the feed name. The *location_ph_feed* has "/csv" appended unlike the battery feed because more than a single measurement is transmitted. This will be seen later. The call *fona.begin(fonaSS)* seen in step 5 establishes the serial connection between the FONA808 board and the Arduino Uno. Steps 1-5 begin defining parameters which are ultimately used to establish the wireless MQTT connection. These steps also establish the serial connection between the cellular breakout board and the Arduino itself. Steps six through 11 configure the FONA to handle the MQTT transmission. Line 6 checks the network status to make sure that a successful

connection to the cellular network has occurred. Step 7 enables the GPS capabilities of the FONA board. The eighth step defines the network GPRS parameters. These parameters are APN username, password, and the network name. Steps 9 and 10 reset the GPRS module so that it reports clean readings. The final step makes the MQTT connection to the Adafruit server. Now messages can be sent.

Sending messages is straightforward now that the connection has been established, network and connection settings defined, and the GPS module has been started. GPS isn't a required for data transmission, but as a defining data point of the Mobile Waterway Monitor it is included. Transmissions should be defined as functions. Having a higher-level, abstracted function call which transmits information increases code readability. One such example of a transmission function will be described below. The function *log_ph_loc* handles transmitting pH value and location parameters to the Adafruit server. The basic structure of the function is given below.

```
1.        char sendBuffer[120];
2.        memset(sendBuffer, 0, sizeof(sendBuffer));
3.        int index = 0;
4.        dtostrf(phVal, 4, 3, &sendBuffer[index]);
5.        index += strlen(&sendBuffer[index]);
6.        sendBuffer[index++] = ',';
7.        publishFeed.publish(sendBuffer);
```

Lines 1-3 declare an empty and cleared character buffer. This buffer allows strings of characters to be concatenated into a single string. The iterator, *index*, is initialized to the beginning of the character buffer. Lines 4-6 show how value is converted. In line 4 the *dtostrf* method takes a floating-point value and trims this down to the width specified. Width is specified in this example is 4, or more generally by the second parameter. The following integer value, 2, represents the number of digits which may succeed the decimal point (example: 2 would look like 0.98 not 0.981). The final argument is the address of memory with which to begin storing the string into. Each successive character is stored at the next memory address. The memory addresses need to be

contiguous which explains the need for a character array titled *sendBuffer*. After storing the string of text the index is incremented by the number of characters in the string to point to the next available memory address. Following the CSV format, line 6 appends a comma to the string of character and then increments the index by one to account for the commas additional length. Appending strings and commas can be repeated several times until the buffer is filled. The last string appended to the buffer does not need to be followed by a comma. Also, remaining buffer space doesn't need to be filled with non-zero values. From here the buffer can be sent using the *publish* function of the Adafruit_MQTT_Publish object. The message is now pushed to the server (if the publish call was successful).

The message transmission is now complete. The MQTT protocol has successfully sent the message over the GSM network.

Chapter 2.4 Software Libraries

Using libraries when programming is a guaranteed way to increase production speed and remove extra potential for errors. To accelerate up the creation of the Mobile Waterway Monitor, and safeguard against typos and other small logical errors, several programming libraries were used. The libraries fell into two regions of the implementation: the river node and the Python script. It is worthwhile to talk about these libraries functions and why they were used in implementing the MWM. I will not detail every single function in each library but I will be going over the functions I found important to this project. The Arduino libraries used assisted with data transmission and automated resets. The first library, titled "Adafruit_FONA" contains all the function calls needed to transmit data over the GSM connected FONA 808 breakout. The "Adafruit_MQTT" library likewise implements all the MQTT wrapping needed for the protocol to work over TCP. The "Adafruit_SleepyDog" library implement a generic watchdog timer which was used to increase the reliability of the node. Basemap, Matplotlib, Pandas, and TkInter were used to aid in data visualization and creating a graphical user interface in the Python dashboard. This section will begin with Arduino and then discuss Python.

One of the most important Arduino libraries used, the Adafruit_FONA library, is mostly abstracted away by the MQTT library. Still, a couple of functions exist which are critical to the program. The first is the FONA constructor function *Adafruit_FONA(int8_t r)*. This function constructs the FONA object. Executed part way into the Arduino sketch, the FONA object, *fona,* is used to establish the MQTT connection over the FONA board. Functions *enableGPRS* and *enableGPS* are used in the beginning of the setup to reset the GPRS and GPS modules to allow a clean GPS and cellular connection later. The FONA library also contains a TCP connection check, *TCPconnected(),* which indicates whether a connection has been established through the board.

Other needed functions include *getGPS()* and *getBattPercent()* which both do what they sound like. Overall, the Adafruit_FONA library is critical to all data transmission through the FONA 808 GSM breakout. While I have mentioned a few direct calls above, there are many more calls which are available to include services like text, real time clock, calling, and HTTP methods. The Adafruit_FONA library is also needed for the Adafruit_MQTT library, which abstracts many of the TCP calls into a simpler format.

Adafruit's MQTT library, Adafruit_MQTT, handles everything required by the IOT inspired MQTT protocol. This library provides functions which are widely used in the MWM and serve as the basis of the MQTT transmission. Functions which are called include *Adafruit_MQTT*, *Adafruit_MQTT_Publish*, and *connect*. These three functions are used to handle much of the communication needs. The constructor, *Adafruit_MQTT*, establishes the characteristics of the connection by taking input parameters such as the server, port, client ID, username, and password. The *fona* object created by the *Adafruit_FONA* constructor mentioned in the above section is passed in as the Server. The *Adafruit_MQTT* constructor is used to initialize an object instance titled *mqtt*. This instance is used in many places. Publishing data strings, completed by the *Adafruit_MQTT_Publish* method, is one example of where the *mqtt* object is used. The publish constructor is used to create a MQTT feed. Data is then pushed to the feed over the MQTT connection to the Adafruit IO feed (in CSV form). The *Adafruit_MQTT_Publish* method takes two mandatory parameters with a third optional parameter. In my programming, I have not needed to set the third parameter, *qos*. The first parameter, *mqttserver* is a reference the *mqtt* object instance created above. The second input is *feed*. This feed parameter specifies the feed name to be pushed to. These feeds are reflected on the Adafruit IO website. Feed objects are then used to write the data across the MQTT connection by a *publish* call contained by the feed object. The final function

I described as necessary is *connect*. This aptly named function establishes the MQTT connection to the Adafruit IO server.

Both the Adafruit_FONA and Adafruit_MQTT libraries are used to establish the wireless connection for data offloading and work together, in distinctly different manners, to implement the full transmission link. The FONA library contains functions which directly manipulates the FONA board itself to implement the cellular connection. The MQTT library wraps data in a MQTT header to establish a stable, simple, and low powered TCP connection. The MQTT library does not write the actual bits to the FONA 808 board, but instead relies upon the FONA library for the intricacies of communication. Data transmission isn't the only complex component to the Mobile Waterway Monitor. Self-righting code drastically mitigates the problem of stalls and infinite loops in software. For this reason, the Adafruit_SleepyDog library is used in the MWM.

The Adafruit_SleepyDog library [13] implements a software watchdog time. In layman's terms, a watchdog is a software defined reset switch which goes off in timed intervals. Wikipedia defines the watchdog timer as "an electronic timer that is used to detect and recover from computer malfunctions" [14]. System recovery for the Mobile Waterway Monitor is handled by a watchdog timer. Automated recovery via a software timer helps catch possibly futile attempts at cellular connection, breakout software connection faults, and GPS malfunction. Automated resets were implemented because of the simple fact that manual reset isn't possible in almost any MWM error scenario. The header file, *Adafruit_SleepyDog.h*, is composed of preprocessor directives which define the board type for the watchdog timer. In the case of the Arduino Uno, the watchdog type is selected to be *WatchdogAVR*. This is defined by the Arduino core architecture. Simple and compact, the five functions of the watchdog timer are necessary for the MWM. The watchdog timer's *enable* function starts the watchdog timer count down. This function takes an integer input

parameter to specify the number of milliseconds in the watchdog count down. The antithesis of this function is *disable* which simply disables the watchdog timer and subsequently, the countdown. Called regularly in various sections of the program, the *reset* function restores the countdown to the original value and allows the code to progress as expected. In the case of the MWM, the watchdog interval is set to 10 seconds. If a portion of code between two resets takes longer than 10 seconds, which would indicate a stall, the program will reset. Calling *reset* is also much more succinct then disabling and reenabling the watchdog timer which improves readability. The final function, *sleep*, might seem redundant as the Arduino platform already has a sleep function, but it performs wildly different. Calling a non-watchdog sleep function while a watchdog enabled would result in an interrupted software sleep. The sleep would be interrupted by the program resetting. Instead, a watchdog sleep timer is used to place the Arduino chip into a deeper, more energy efficient sleep while simultaneously pausing the watchdog count down for the duration of the sleep. This completes the coverage of the Arduino libraries.

The Python dashboard takes advantage of the large number of scientific libraries available. Libraries used in the dashboard include Matplotlib, Basemap, Pandas, and TkInter. The first three listed are for data analytics and visualization. TkInter is used for GUI creation. First covered will be the Matplotlib library followed by the Basemap, Pandas, and TkInter libraries.

All line plots in the Python dashboard are being displayed by the Matplotlib (MPL) library [15]. MPL provides both much needed graphing capabilities as well as a large list of features for nearly identical work as other libraries. The dashboard plots seen in this paper were graphed using Matplotlib. MPL graphs are highly customizable. For my dashboard prototype I almost exclusively used the generic MPL plot. The *plot* function is well documented on the Matplotlib website which hosts installation instructions, usage, and API documentation. Compatible with several input

formats, the MPL plot can take two dimensional arrays as input. All Pandas series are naturally 2 dimensional which makes data storage and plotting as simple as using the same series. Pandas series were used in the Python dashboard as the sole input parameter to the MPL plot with great success. The Matplotlib website hosts a long list of other graphs, charts, and customizations available. Breaking away from the MPL graph used, I would like to explain why Matplotlib is especially powerful for the Mobile Waterway Monitor's Python dashboard.

The Matplotlib plot houses an enormous number of features which come standard. Contained in the plot figure is a toolbar, as you can see in the plot below of the pH recordings, which has 8 icons to select from. The home option, signified by the small house, resets the entire plot to its original state after being most recently drawn. The back and forward arrows, second and third from the left, allow the current operation to be reverted or the current plot to be changed back to the next most recent change. The arrow cross button turns the mouse into a panning pointer in the viewing window. The pointer will now move the graph in any direction dragged to further expose that region of data. Combined with magnification, the panning pointer becomes incredibly powerful for precisely viewing data. The magnifying glass zooms in, for closer inspection. The green arrowed graph logo (6^{th} from the left) allows configuration of the plot itself. The save button exports the graph into a PNG file. The final option, a green checked checkbox allows configuration of the line, its colors, and other parameters like line scaling and line type. Overall these powerful functions allow more control in interfacing with the Python dashboard's many graphs.

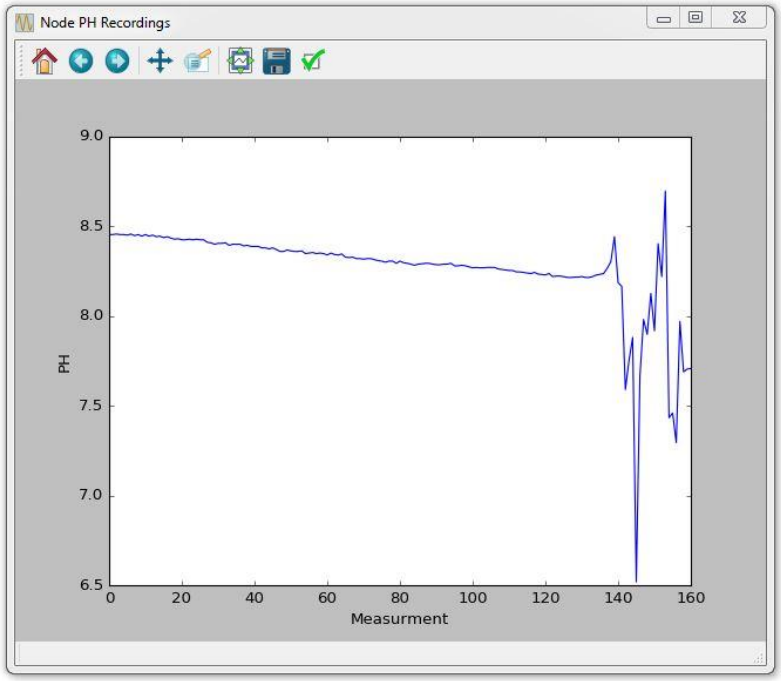


Figure 5: Matplotlib Example Window

Figure 6: Matplotlib Options

Basemap, which is built off Matplotlib, also shares the toolbar feature [16]. The Basemap toolkit was chosen for its ease of use and configurability as well as compatibility with Matplotlib. Basemap follows a similar plotting pattern as Matplotlib. A Basemap object is created and named and later that object calls the *plot* function. This is almost the same as for the simple MPL plot. The most significant difference is in how the Basemap instance is created. The *Basemap* constructor function takes several parameters. In the MWM dashboard it was necessary to supply seven arguments. These arguments specify the size and location of the map as well as the type of projection. For the projection type a simple Mercator projection was used. Mercator projections

map the earth cylindrically to produce a clean rectangular map when laid out. Mercator maps produce distortion at very high latitudes but are easy to read and are visually appealing compared to other graphs when drawn rectangularly. Distortion shouldn't come into play when mapping a smaller area, like a state or region when compared to the world. The map area is specified by the lower left and upper right corner longitude and latitude values. These are passed in as four parameters: *llcrnrlon, llcrnrlat, urcrnrlon, urcrnrlat*. The *area_thresh* parameter is used to plot bodies of water which have a surface area greater than or equal to the parameter in square kilometers. The final parameter *resolution* specifies the overall map resolution. For the color-coded pH map, 'h', or high, was to be the best balance between a highly accurate map and quickly rendered map. There are several settings from low to high fidelity. And once again the Pandas library helped with mapping data.

The Mobile Waterway Monitor dashboard takes advantage of two major pieces of functionality from the Pandas library [17]. The first, and possibly most important function, was the *read_json* function which downloads the JSON file from an Adafruit IO data feed. Pulling the completed pH location data from Adafruit into a Pandas object made parsing and manipulations more convenient throughout the entire program. The Pandas series can also be directly plotted by Matplotlib. The Pandas series itself was the second major feature which helped in implementing the graphing utilities. The series contains a set of user configurable indexes, in the dashboards case the raw index, which is paired with each data point. This means that a Pandas series is a two-column list. One column being the indexes, the other being the data itself. This data can be anything from elevation, pH value, location, speed, to strings. The *Series* function can even convert from a Python dictionary to a Pandas series which helps with plotting interpolated data such as speed.

The Pandas library as well as the Matplotlib and Basemap toolkit provide data visualization functions for the dashboard, but provide no tools for GUI creation. Enter TkInter. TkInter is a GUI toolkit which allows for rapid user interface creation [18]. It is backed by the Python community extensively and has basic components which build off each other to create flexibility. All TkInter applications follow a similar structure. The Tk main loop runs indefinitely until exited. All functions are called form GUI components or timers within this loop. The downside of this single loop architecture is that any form of threading is incredibly difficult. The positive aspects are that the TkInter elements can be updated using function calls and the program design simplifies call trees. Buttons, checkboxes, and labels are some of the basic elements of TkInter. The Tk library supports many window layouts including grid, packed, and place. Grid allows the user to specify the number of rows and columns to divide the screen into. The window elements are then arranged in the column and row entries. The packed layout is simpler, but much less customizable, and only allows elements to be placed relative to each other. Spacing, size, and location of elements is otherwise decided by the manager. The final layout manager is place which requires explicit placement directions within the Tk window. This allows for serious flexibility but takes too much time to get working properly without issue. The Python dashboard is based upon the grid layout. The button and checkbox instantiation is very similar. For a button instantiation is *Button(window, text, command)* where are checkboxes are *Checkbutton(window, text, variable)*. The button parameters define which window, what button text, and what function to call respectively. The checkbox is almost exactly alike but instead of a function call, a TkInter variable is set. These then must be positioned using the *grid* function which specifies the row, column, row span, column span, and sticky directions. The row and column decide the row and column to place the element. The row span and column span declare how wide and tall the element will be. The sticky directions

tell which cardinal directions the element should stretch to. North is defined as the top of the window. Besides running only, the main loop, TK has a function called *after* which will call a function after a specified time. The format for this function call is *after(time, function_name)*. This proved handy in my code as it allowed me to pull from the Adafruit server without blocking the GUI from loading. Self-referencing functions can also be used to implement a sort of pseudo parallelism. Parallelism through self-referencing functions was attempted, but the results were lackluster and so the attempt was scrapped.

Programming libraries greatly helped speed along development of the river node and the Python dashboard. Without libraries, the Mobile Waterway Monitor would be comprised of software packages which would both be highly unstable and non-portable. Code libraries for the Arduino helped ensure reliable data transmission without exact hardware specification knowledge. The Python libraries allow simple use of complex data visualization functions and analytics. The Python and Arduino communities are singlehandedly the reason behind the wealth of libraries available. The MWM builds off these platforms because of the enduring support and development of the Python and Arduino communities. The consistent expansion of these platforms will only lead to a more powerful Mobile Waterway Monitor in the future.

Chapter 3 Design and Performance

Chapter 3.1 Design Independence

The Mobile Waterway Monitor system is composed of three connected, but not fully interdependent, components as mentioned in the overview. These pieces consist of the river node, the hosting server, and the Python script. The current MWM architecture only requires two of these components, the floating node and the server, to work together. A connection between the floating node and the supporting server must be established to transmit data. The Python dashboard is currently optional. The server offloading architecture, shown in figure 3, was selected in this project because of the fast-paced growth of IOT technologies and real-time processing which is currently underway. The MWM isn't locked into this design. The flexibility of the cellular breakout board means data can be sent over text messages, send TCP packets without MQTT wrapping, using HTTP POST and GET methods, or could have a simple SD card for data storage connected to its extra input/output pins. A server-client design was chosen to showcase emerging IoT technologies, like the Adafruit service, as well as increase data visualization options. Along with architecture requirements, each component does have its own set of limiting requirements.

Sadly, the river node does have a few requirements to run. First, the battery cannot be fully drained. Battery dependency, which will be seen in the following section, is due to the large, highly short burst of current draw needed for data transmission. The solar panel is not able to fulfil the 1.7-amp current draw which is needed when sending the MQTT message. Sending the message also requires a cellular connection. These are the only two node specific requirements to run. These requirements are very minor, a theme which is repeated in the server. The current MWM prototype does require a server to offload data. This server dependency could easily be removed. The idea of adding external, but physically present, storage is interesting. Adding a simple SD card through

an expansion board would lead to interesting benefits such as reduced power consumption and higher sampling rate. Instead of transmitting every data point, as it currently does, several points could be stored in the Arduino flash memory and averaged and then the calculated mean could be transmitted. Each individual point could then be stored to the SD card. Local storage to an SD card is likely much more power efficient than constant transmission. Another way around the server dependency is to use text messages to transmit data to another device. This is more a work around and reduces some of the feature set like globally available data. While texting data may be more energy efficient (unlikely) it adds another step to data processing. Converting the data transmissions from a text message to something like a text file or common spreadsheet format is another step for which additional tools would need to be used, and for little gain. Sending the data in a more usable form to a service like the IOT Adafruit website for hosting, parsing, and organization removes many steps. Additionally, the Adafruit service is likely more reliable than another cell phone which must be constantly charged.

The Adafruit server does not have any requirements to run. It simply waits for data to be pushed to it. Accessing and storing data does require an account. The account is free and takes about 5 minutes to create.

The Python dashboard also has few requirements. If properly installed with all supporting libraries the only requirement is the server to pull data from. A significant redesign would be required if the Mobile Waterway Monitor node was reconfigured to text data or use an SD card.

Chapter 3.2 Performance

Battery life and performance are critical to the Mobile Waterway Monitor. Sadly, it is probably the most difficult aspect to have total control over when using commonly available, off the shelf components. Lackluster battery life is offset by the accurate measurements of the MWM. This section will cover both performance and battery life of the river node. The server and Python script performance is far less crucial than the actual measurements as they do not degrade data received by transmission. There are three non-power related components which define system performance. They are the enclosure, the pH probe, and the GPS module. These components will be covered first followed by power related components which define system power usage.

The Pelican 1060 case is a cheap and impressive fix to a floating waterproof enclosure. The enclosure is rated to be waterproof at 1 meter for 30 minutes. The 1060 case is stable between -10° and 199° Fahrenheit. It also is buoyant with a load of 2.49 pounds and under. There is more than enough space to assemble and configure the MWM hardware within this case. It is also extremely tough. The polycarbonate construction takes rocks and small waterfalls with ease.

The pH probe is another highly reliable component. The DFRobot Gravity pH pro meter can record data in fluids for up to a year. Measuring the full pH range (0-14), the probe is both accurate and durable handling temperatures between 32°F and 140°F. The response time of this sensor is well under 1 minute. As it comes, the pH meter is accurate to within 0.1pH at 77°F. A tenth of a pH value is more than enough accuracy for detecting waterway trends and displaying precise readings. The only additional source of inaccuracy comes from shortening the connecting cable. This modification was required to fit everything in the case. The factory cable is around 7 feet long. The shortened cable changes the resistance of the reading wire. The pH meter accounts by this by having a variable resistor which can be configured to match calibration results. No

additional error is added by the Arduino or transmission. I have found that calibration takes care of the error from shortening the cable. Calibration is critical when the connection is shortened.

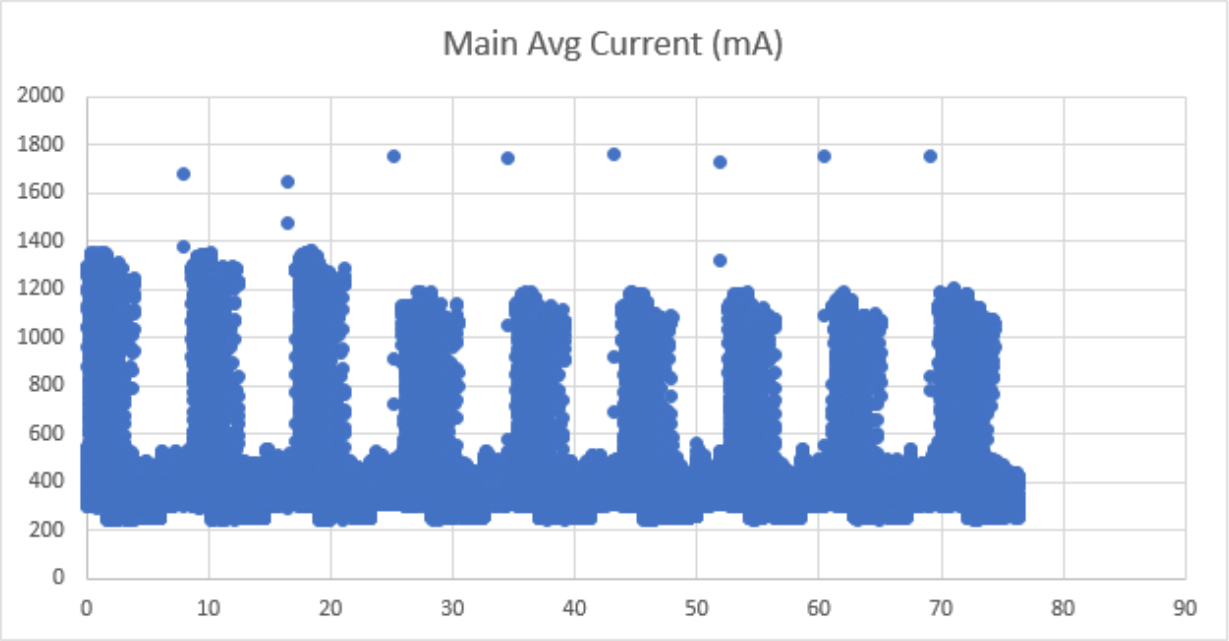
The Adafruit FONA GPS module ties the whole Mobile Waterway Monitor together and represents one of the most configurable performance defining components. The FONA breakout board has a GPS module which is accurate to around 2.5 meters, or 8.2 feet. Precision is as good or better than most cellphones. The GPS module supports 22 tracking channels and 66 acquisition channels. The large number of tracking channels allows the FONA to keep a more consistent GPS connection. More tracking channels means better positional accuracy, better sensitivity, and even slightly reduced power consumption. The large number of acquisition channels make sure that the time-to-first-fix is very low. The time-to-first-fix for the FONA is around 32 seconds when starting from cold and as fast as 1 second when given a hot start. This board has been more than adequate for the project and reliable with channel connection and location fix. The FONA 808 supports quad band cellular as well as data. Other FONA breakout boards can be purchased without GPS or cellular data support. Having both functionalities is critical to the Mobile Waterway Monitor implementation.

Besides system functionality and measurements, the power consumption and power management of the Mobile Waterway Monitor represent the last performance defining region of the project. Power management is handled by several by several components, however, only the more important will be mentioned. Small components such as wire gauge, connection types, and resistance values are both variable between systems and user configurable to fit specific needs. The backbone hardware which is constant across any implementation is the solar panel and voltage regulator. Once these are covered the entire systems power consumption will be discussed.

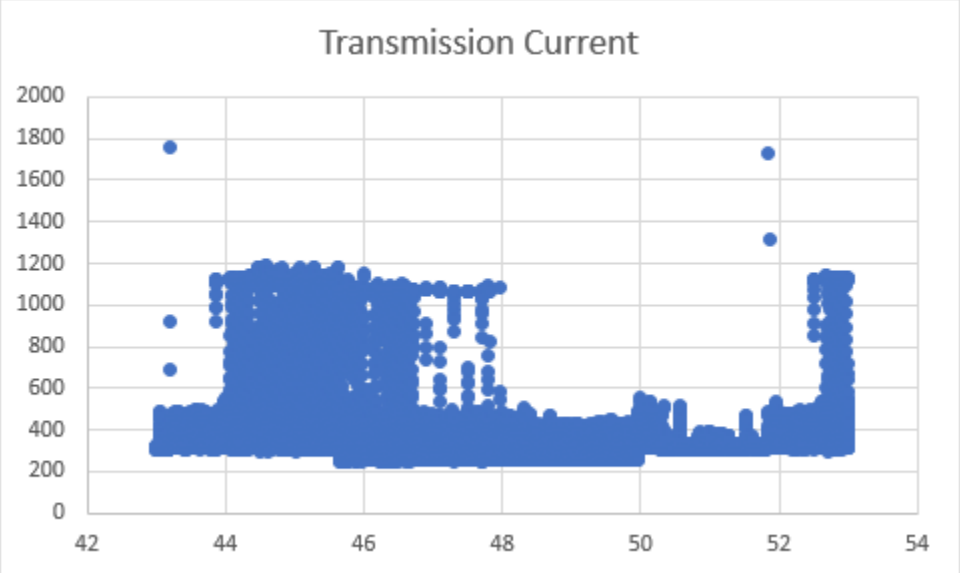
The Adafruit VERTER controls all the voltage regulation from the battery to the rest of the system. The VERTER uses a Texas Instruments TPS63060 Buck-Boost Converter. TI's chip will pull up low voltages from as little as 2.5 volts and drop high voltages from as high as 12 volts to a steady 5 volt output. Direct current is assumed. The datasheet [19] shows that the chip produces around a 1 Amp current when fed 3.7 volts from the battery. While producing this 1 Amp current the VERTER is 90% efficient. The TI chip is current limited to 2.25 Amps which is more than enough for the MWM. It is important to note that the FONA cellular board has an input power line which is fed directly from the battery and avoids the VERTER completely.

Now that all power consuming parts are covered only the solar panel remains. The small solar panel in the MWM supplies 2 Watts total at full luminosity. Full luminosity is defined by the manufacturer to be $100\frac{mW}{cm^2}$. Arizona averages around three quarters of that in the summer and a half in the winter according to the National Renewable Energy Laboratory. The solar panel at its peak will generate 6 volts and 378mA of current. The solar panel is 19% efficient. This will not completely power the setup, especially when considering for parasitic losses. What it does currently is extend the run time of the Mobile Waterway Monitor significantly.

The final area of exploration for this section is the overall power draw of the MWM. This power draw will be reported as current because the voltage demands are not easily determined thank to the voltage regulator. Even so, the current results are very interesting and provide a lot of information on how to make the MWM last longer or possibly even indefinitely. Below is the current draw for several transmissions near the beginning of the successful connection.



The cellular communication channel, as seen above, stabilizes around the 25 second mark. After the first 25 seconds the transmission cost is almost identical between communications. Included below is a plot which highlights a single transmission and explains the communication cycle.



Each individual transmission looks to be comprised of four distinct sections. The first region occurs between 43 and 44 seconds. This is the initial Arduino-FONA transmission burst. The intercomponent burst readies the cellular modules in the FONA board and verifies the connection between the breakout board and the Arduino. This burst is extremely brief. Maximum current draw is found during this communication burst according to the SIM800 documentation. The pH reading is also taken during this stage. The second region, between 44 and 46 seconds on the plot above, is where the cellular transmission takes place. This would explain the near constant 1200mA consumed during this face. The third phase, between 46 and 47 seconds, wraps up the transmission and begins to put the Arduino into a sleep setting. The FONA board at this point is still consuming burst of heightened current because it is wrapping up the communication and setting the cellular chip into a non-transmitting state. The final stage is between 47 and 51.5 seconds. In this phase, the Arduino is in sleep mode and no readings or transmissions are taking place. This cycle happens around every 8 seconds while a connection is maintained.

Below is a chart which summarizes the average current and voltage for these two plots.

Plot	**Current (mA)**	**Voltage (V)**
Main Avg Current	349.8844	3.880107
Transmission Current	351.8879	3.880105

Finding the average power consumption to be around 350mA for strings off successful transmissions seems right from testing. The node has been found to average slightly high at around 550mA current in areas with poor cell coverage. Reconnection is costly and brings the average up. What I have found was around a 12-hour run time for the MWM in mixed lighting conditions, during the months of February and March (about half solar panel output) and with movement and

reconnections. Testing in a summer month where solar energy is more intense would likely yield 14-16 hour run times. And this is with data transmitting about every 9 seconds.

Widening the transmission interval could save a substantial amount of power consumption. If we consider the costliest 5 seconds from the 10 second *Transmission Current* plot alone we find that it consumes about 40mA more than the average (around 390mA). The remaining 5 seconds then average a 310mA current draw. This isn't really a massive power savings for the precision lost. Building off this realization, an even wider transmission interval could be used to save energy. Instead of spacing out transmissions by seconds, sleeping for minutes at a time would save the system even more power, especially when disconnecting the cellular connection. The Arduino and other connected components could be woken up by a small timer which triggers an interrupt. The timer duration could be changed to better reflect accuracy needs. With proper, and wide enough, measurement spacing night monitoring could also be possible. Another solution is to use a more limited range transmission medium such as short range radio waves. Small boards like the XBee or ZigBee boards use radio waves to transmit data at a much lower power cost [20]. Short range radio would work especially well for lakes or contained bodies of water where several nodes could communicate to a single transmitting node. Introducing dozes of interconnected nodes into a body of water does get away from the design goal of minimal infrastructure needs and costs. Traversing a river and transmitting data while needing no supporting infrastructure is powerful, and it is a defining feature of the Mobile Waterway Monitor.

In summary, this prototype lasts a decent amount of time considering it is comprised of connected off the shelf components and contains many unused points for design expansion. Changes in components like using a smaller Arduino, a more configurable solar power system, or a different communication technology leaves much room for improvement. The current prototype

averages a current draw of around 350mA from its 3.7 volt battery. The prototype will last around 12 hours will average connectivity and solar panel output.

Chapter 4 Testing

Chapter 4.1 Experimental Results

By now most readers might be wondering what exactly happens when the Mobile Waterway Monitor is in use. And so far, a concise answer really hasn't been given. The Experimental Results section will cover what happens when using the Mobile Waterway Monitor and its connected website and Python dashboard. The following subsection, titled "Demonstration Materials" provides demonstration video which shows how the data presented here was collected. There are effectively three ways to interact with data collected by the MWM. All data can be interacted in near-real time. The most rudimentary way to view data is to directly export the data, either in its full form, or in a truncated form from Adafruit's website. The hosting website provides its own, more powerful, means of data access [21]. The projects Python dashboard provides the final interface, which has some built in data analysis.

Exporting data is a simple set of steps for a glut of unmodified, raw data. Directly exporting the data from Adafruit's website proves to be the easiest way to interact with values. Adafruit provides two methods of grabbing data. The first, and probably most useful, is downloading a single feed of data. This is much easier to view without confusion than downloading all data. For instance, grabbing the GPS tracked pH readings from the feed 'RIVER_NODE_LOCATION_PH' is a simple process. To download, navigate to the feed, and look under the actions menu, shown in an image below.

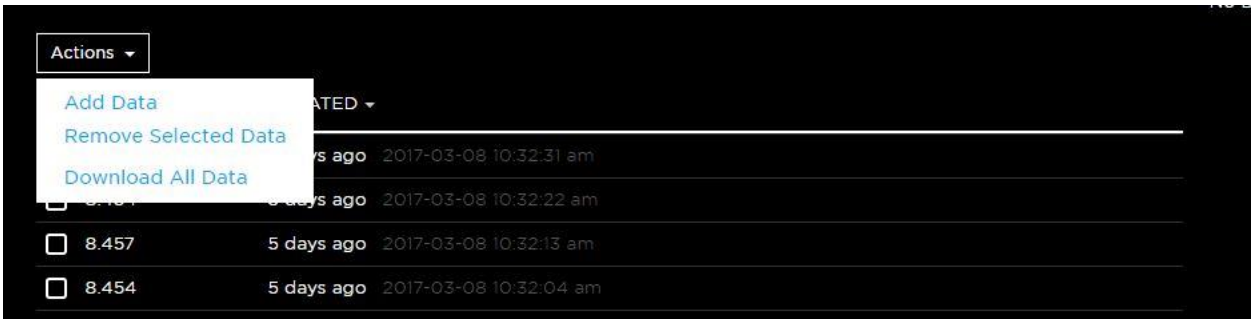


Figure 7: Adafruit Feed Export

From here, selecting 'Download All Data' will pull up a prompt which will allow either formatting the data in JavaScript Object Notation (JSON) or Comma-Separated Value (CSV) format. Both are common formats for programs and macros. The prompt will now be shown below.

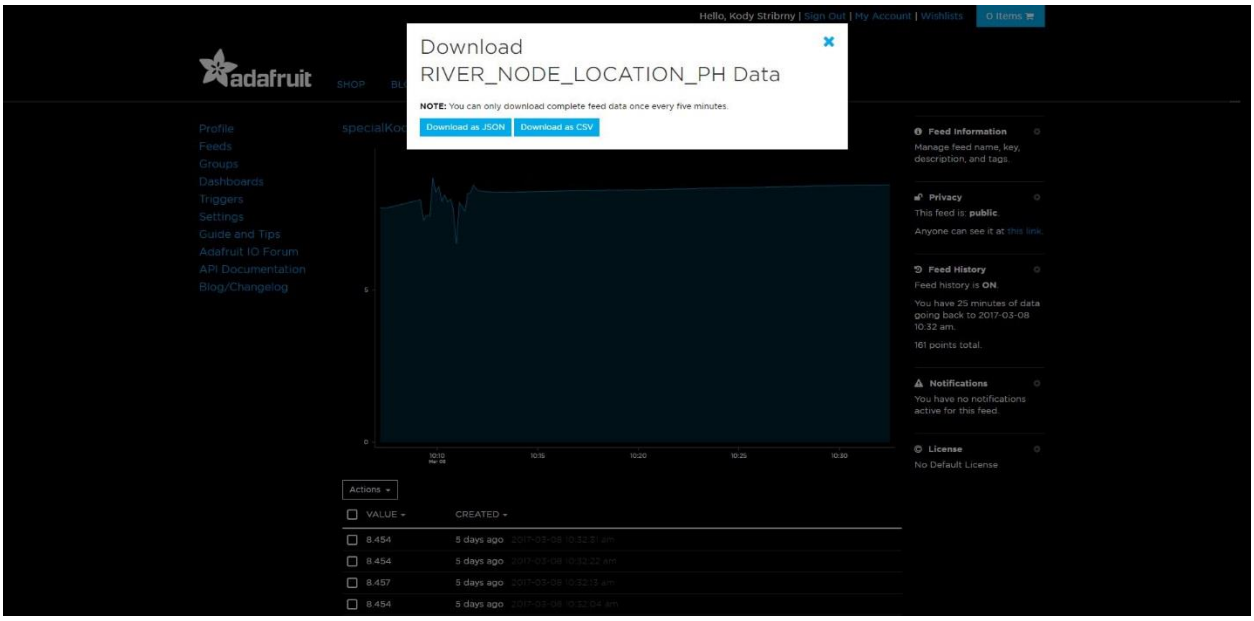
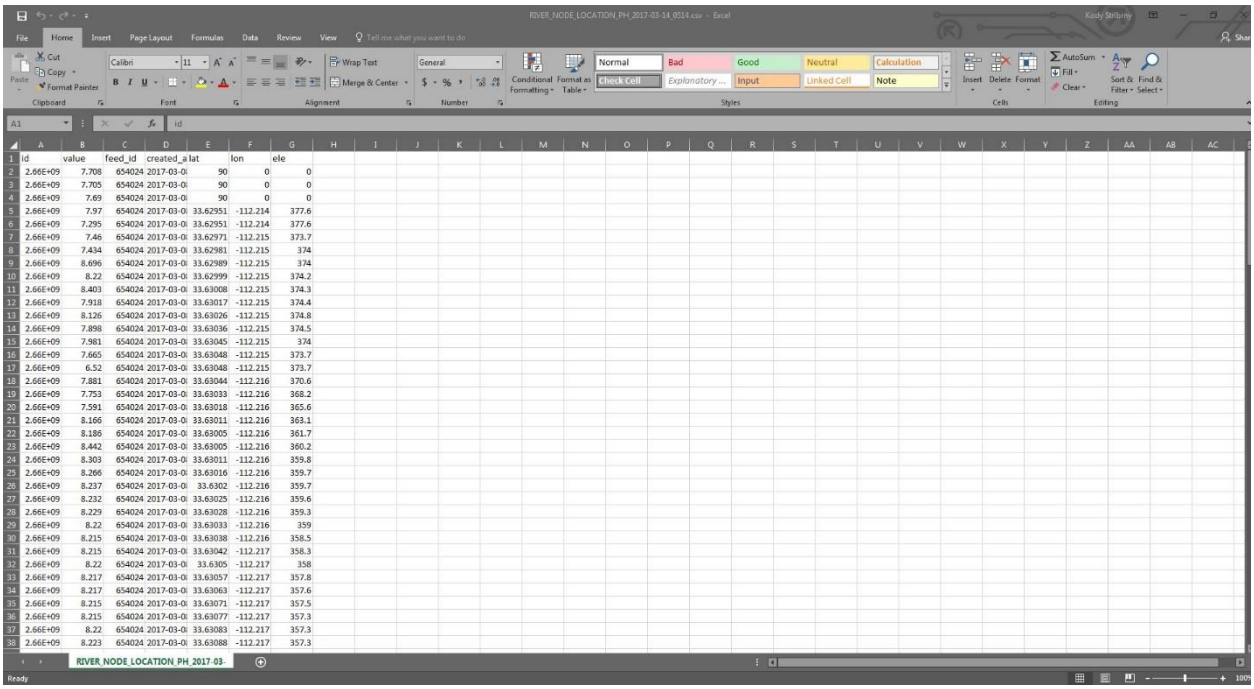


Figure 8: Adafruit Feed Export Options

For the example, I decided to download the data as a CSV file. The filename format seems to be:

FEED_NAME_year-month-day_count.csv

Where *FEED_NAME* is the name of the feed downloaded from. The date comprises the second part in the form *year-month-day* where the year is four digits, the month is two digits, and the day is two digits. The date does seem to be localized to New York. Adafruit is based out of New York. So, it is likely that the server exists there. The final component, *count*, seem to be a global download counter, or at least, I did not find it do have any significance to the feed. The data file is shown below when opened in Microsoft Excel to give an idea about formatting and settings.



id	value	feed_id	created_a	lat	lon	ele
2.66E+09	7.708	654024	2017-03-0	90	0	0
2.66E+09	7.705	654024	2017-03-0	90	0	0
2.66E+09	7.69	654024	2017-03-0	90	0	0
2.66E+09	7.97	654024	2017-03-0	33.62951	-112.214	377.6
2.66E+09	7.295	654024	2017-03-0	33.62951	-112.214	377.6
2.66E+09	7.46	654024	2017-03-0	33.62971	-112.215	373.7
2.66E+09	7.434	654024	2017-03-0	33.62981	-112.215	374
2.66E+09	8.696	654024	2017-03-0	33.62989	-112.215	374
2.66E+09	8.22	654024	2017-03-0	33.62999	-112.215	374.2
2.66E+09	8.403	654024	2017-03-0	33.63008	-112.215	374.3
2.66E+09	7.918	654024	2017-03-0	33.63017	-112.215	374.4
2.66E+09	8.126	654024	2017-03-0	33.63026	-112.215	374.8
2.66E+09	7.898	654024	2017-03-0	33.63036	-112.215	374.5
2.66E+09	7.981	654024	2017-03-0	33.63045	-112.215	374
2.66E+09	7.665	654024	2017-03-0	33.63048	-112.215	373.7
2.66E+09	6.52	654024	2017-03-0	33.63048	-112.215	373.7
2.66E+09	7.881	654024	2017-03-0	33.63044	-112.216	370.6
2.66E+09	7.753	654024	2017-03-0	33.63033	-112.216	368.2
2.66E+09	7.591	654024	2017-03-0	33.63018	-112.216	365.6
2.66E+09	8.166	654024	2017-03-0	33.63011	-112.216	363.1
2.66E+09	8.186	654024	2017-03-0	33.63005	-112.216	361.7
2.66E+09	8.442	654024	2017-03-0	33.63005	-112.216	360.2
2.66E+09	8.303	654024	2017-03-0	33.63011	-112.216	359.8
2.66E+09	8.266	654024	2017-03-0	33.63016	-112.216	359.7
2.66E+09	8.237	654024	2017-03-0	33.6302	-112.216	359.7
2.66E+09	8.232	654024	2017-03-0	33.63025	-112.216	359.6
2.66E+09	8.229	654024	2017-03-0	33.63028	-112.216	359.3
2.66E+09	8.22	654024	2017-03-0	33.63033	-112.216	359
2.66E+09	8.215	654024	2017-03-0	33.63038	-112.216	358.5
2.66E+09	8.215	654024	2017-03-0	33.63042	-112.217	358.3
2.66E+09	8.22	654024	2017-03-0	33.6305	-112.217	358
2.66E+09	8.217	654024	2017-03-0	33.63057	-112.217	357.8
2.66E+09	8.217	654024	2017-03-0	33.63063	-112.217	357.6
2.66E+09	8.215	654024	2017-03-0	33.63071	-112.217	357.5
2.66E+09	8.215	654024	2017-03-0	33.63077	-112.217	357.3
2.66E+09	8.22	654024	2017-03-0	33.63083	-112.217	357.3
2.66E+09	8.223	654024	2017-03-0	33.63088	-112.217	357.3

Figure 9: Adafruit Feed Export CSV

The method shown above only downloads data from a single feed. Downloads from the Adafruit service are limited to five minutes between downloads which at first might seem a bit impractical for backing up several feeds. To archive all data, Adafruit provides a different method. Downloading an entire data set does require account ownership. For instance, I can only archive my own profiles data. To do this, go to the 'Setting' Menu option on the left-hand side of the dashboard. From here a button will appear around midway down the settings webpage labeled

‘Download All Data’. This will download all feed data, groups, and account information for the user. Below is what my account settings page looks like. Another users page would hardly look different.

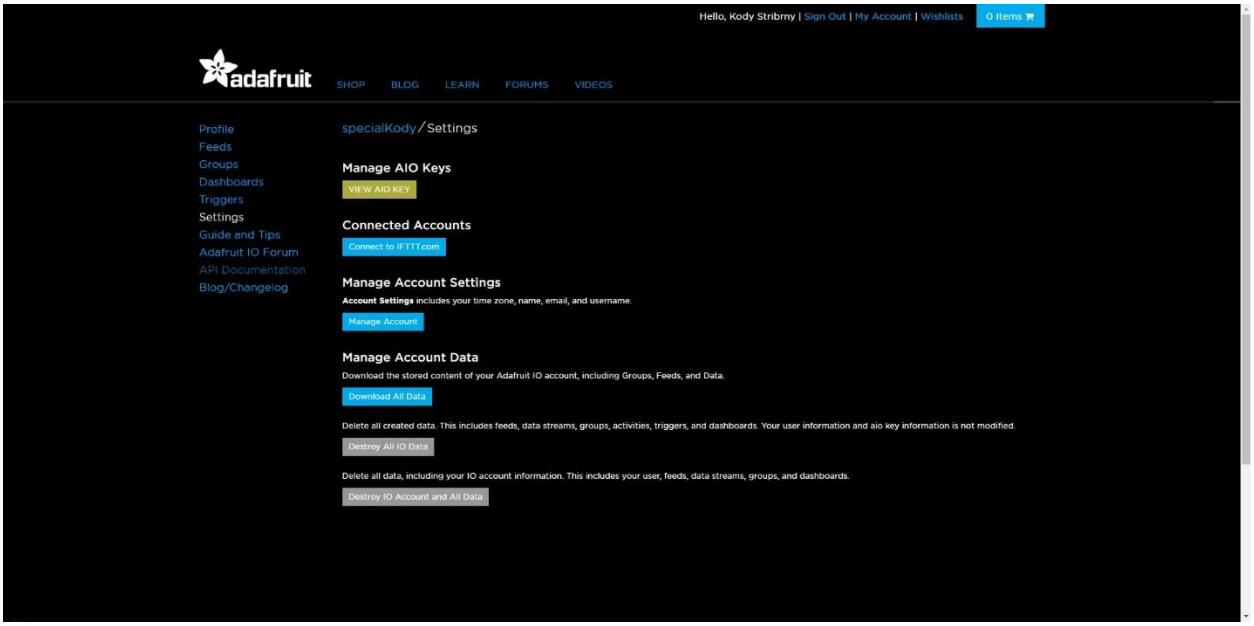


Figure 10: Adafruit Account Export

Downloading all user data does not present options like the previous feed specific download. All of the data will be downloaded as a JSON file. The file, as of writing this, defaults to the name ‘adafruit_io_data.json’. Below shows a small portion of the exported file.

```
"feeds":[
  {
    "id":654023,
    "key":"battery",
    "name":"battery",
    "description":"",
    "unit_type":null,
    "unit_symbol":null,
    "last_value":"90",
    "status":"offline",
    "visibility":"public",
    "enabled":true,
    "license":null,
    "created_at":"2017-03-08T16:51:10Z",
    "updated_at":"2017-03-08T18:36:08Z",
    "streams":[
      {
        "id":2661341227,
        "value":"85",
        "created_at":"2017-03-08T17:07:11Z",
        "updated_at":"2017-03-08T17:07:11Z",
        "completed_at":null
      },
      {
        "id":2661341934,
        "value":"86",
        "created_at":"2017-03-08T17:07:20Z",
        "updated_at":"2017-03-08T17:07:20Z",
        "completed_at":null
```


Figure 11: Account Export JSON File

The Python dashboard provides the last data exportation method. The script directly pulls the JSON file from Adafruit which contains the full JSON data set. Included in the export columns are empty strings and 'NaN', Python's shorthand for 'Not a Number'. As it is currently setup the Python dashboard removes these unfilled columns to reduce clutter. The dashboard 'Export Data To File' option then saves the truncated data. Stored into a text file, the shortened data is much easier to view and very portable. The export file is named 'dashboard_export.txt' and is written to the same directory as the Python script files. The output and output content can be seen below.

Figure 12: Python Dashboard Export

		created_at	ele	id	lat	lon	value
0	0	2017-03-08 17:32:31	349.80002	2661476778	33.630840	-112.22784	8.454
1	1	2017-03-08 17:32:22	350.10001	2661476010	33.630909	-112.22775	8.454
2	2	2017-03-08 17:32:13	350.20001	2661475200	33.630974	-112.22766	8.457
3	3	2017-03-08 17:32:04	350.20001	2661474363	33.631031	-112.22762	8.454
4	4	2017-03-08 17:31:55	350.30002	2661473583	33.631092	-112.22755	8.454
5	5	2017-03-08 17:31:46	350.50000	2661472845	33.631161	-112.22749	8.451
6	6	2017-03-08 17:31:37	350.60001	2661472136	33.631218	-112.22744	8.457
7	7	2017-03-08 17:31:28	350.60001	2661471401	33.631271	-112.22738	8.448
8	8	2017-03-08 17:31:19	350.60001	2661470675	33.631310	-112.22733	8.454
9	9	2017-03-08 17:31:10	350.80002	2661469933	33.631355	-112.22726	8.445
10	10	2017-03-08 17:31:01	350.80002	2661469177	33.631393	-112.22719	8.454
11	11	2017-03-08 17:30:52	350.70001	2661468492	33.631447	-112.22710	8.445

Figure 13: Python Dashboard Export File

The Python dashboard export is handy as it simplifies the export method from a CSV or JSON file to a simple collimated text file. It also shows exactly what data the Python script has been working with. While exporting collected data is nice, it doesn't help with important things like understanding what the data means. Data visualization is one of the best ways to highlight waterway trends. While the Mobile Waterway Monitor reads data at a specific time, location, and depth the overall goal is to highlight waterway characteristics. Trends in pH, speed, and possibly even changing routes can have a significant impact upon the surrounding landscape. Data visualization is accomplished, like downloading data, on two fronts. The first is the Adafruit IOT website. The second is the Python dashboard I have created.

This entire project has used the Adafruit IoT website as a staple ingredient. Adafruit's website, unlike Sparkfun's, has tons of data visualization tools and allows for two-way communication between the receiving server and the sending client. Sending data from the server to the Mobile Waterway Monitor has not been implemented. Data visualization comes in the form of configurable landing page widgets. The widgets can be arranged and sized upon a landing dashboard. The Adafruit dashboard is user specific and configurable. Creating a dashboard to

visualize data requires an account. On this demonstrative dashboard below, I have place a simple map which shows the location of the pH readings.

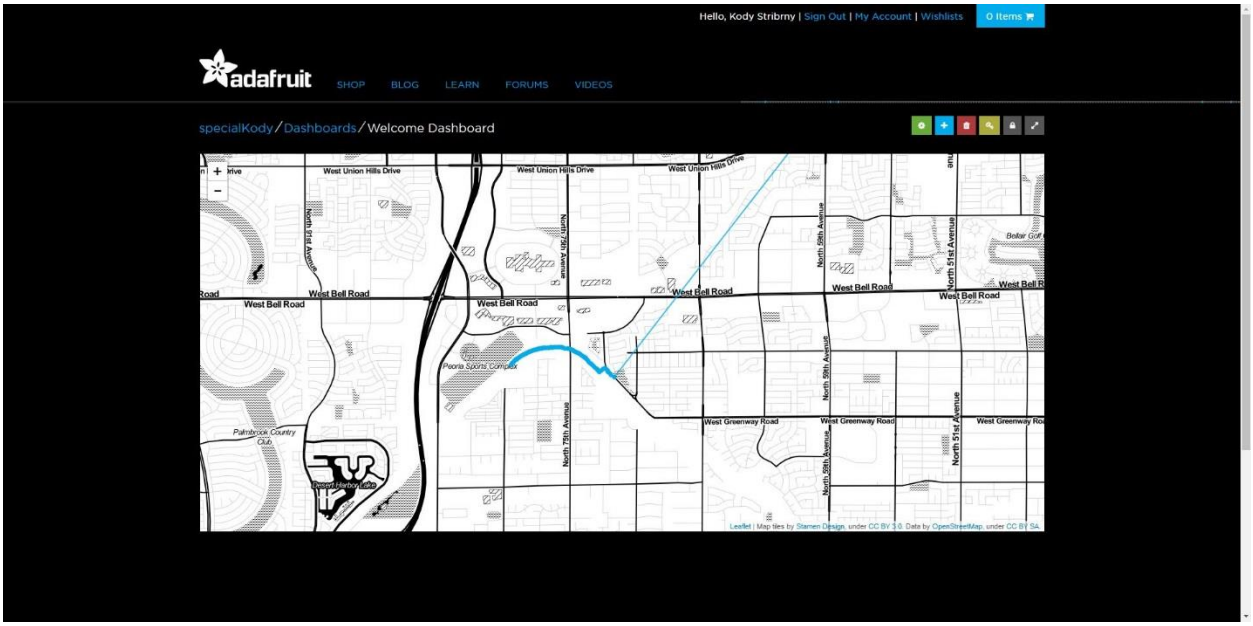


Figure 14: Adafruit Dashboard

The Adafruit pH map, shows each point as a dot. Each point is connected by a line to show the traversal of the readings over the map over time. When a point is clicked, additional data is displayed. This expanded information is shown below. There appear to be two major downfalls of the Adafruit mapping widget. The first is that the widget's greyscale color scheme sometimes makes it hard to read details and text from the image. Second, the map will typically show the first point somewhere in the arctic. This pulls the map to this location instead of starting at the real location. This arctic point is created at startup of the node and can't be ignored by the Adafruit widgets.

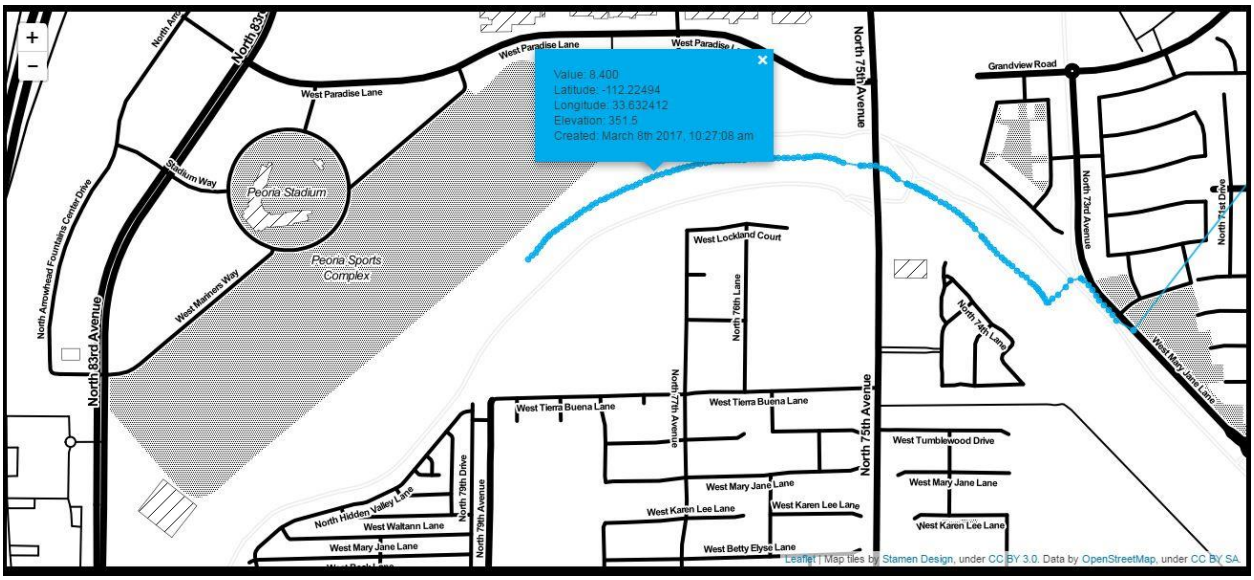


Figure 15: Adafruit pH Map

The Adafruit dashboard can contain much more than just a map. Below is a photo which shows all the current widgets. These widgets include graphs, data streams, pie charts, sliders, and buttons. And these are only the widgets so far under beta. It is a safe assumption that more widgets will become available as the service grows. Additionally, when the API is finalized for the website it is likely users will also implement their own widgets.

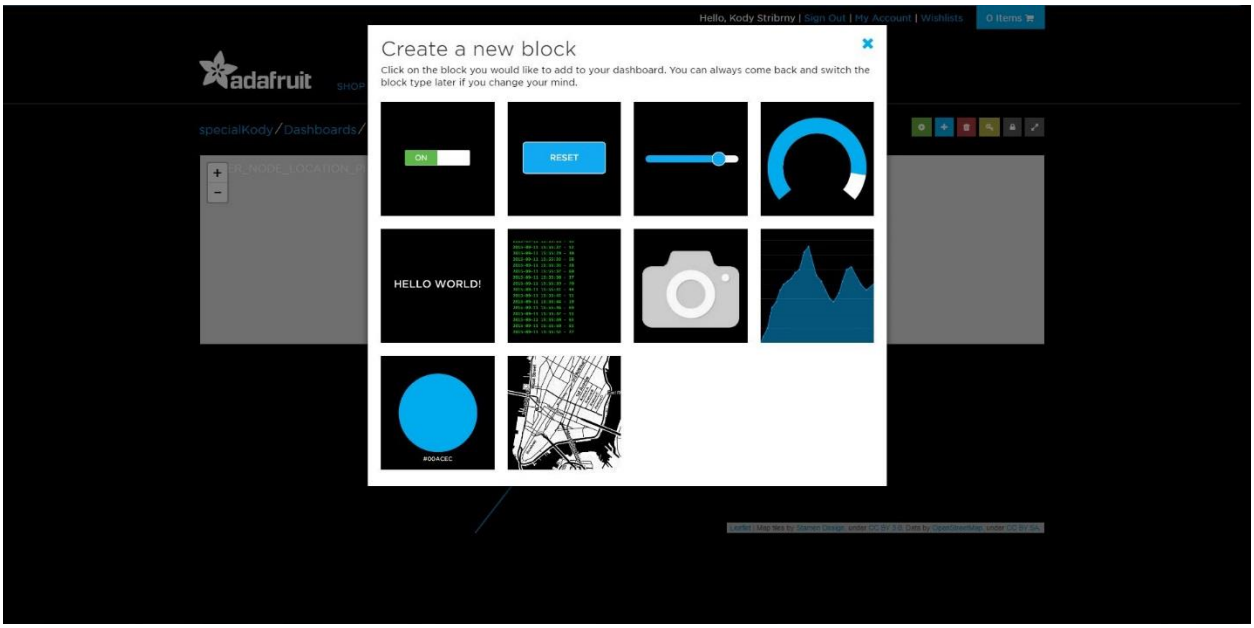


Figure 16: Adafruit Dashboard Widgets

Data visualization isn't just exclusive to the dashboard. The feeds themselves also display a live data stream as well as a live graph. Below is a picture which shows the pH feed. The value described in the data feed column 'value' represents the pH value. The y-axis of the plot also showcases this value. This is a pretty awesome graph for no development work.

Figure 17: Adafruit Feed

The improvements I wished to see on the website have been implemented into the Python dashboard. The dashboard has a lot of room for future growth which include more capable graphs or custom visualization suites. I describe some things I would have liked to add to the project later in this document, and the Python script is one of the larger areas of future grown and improvement.

Three types of data visualization have been implemented in the Python script. The first type is like what Adafruit has on their website, a simple line plot. The Python dashboard does implement a few changes to the display to make the graph more readable. The second visualization tool is the combined plot which shows pH, speed, and elevation on three different plots on the same page. This and the single line plot update in near real-time. The final display is a color-coded map which represents a selected value (pH for the prototype) as colored points along a user configurable map.

When starting the dashboard, you are greeted with a simple user interface which is built off the TkInter GUI library. The welcome window contains a greeting message, 6 buttons, and 2 checkboxes (at the time of writing this). Shown below is the welcome page.

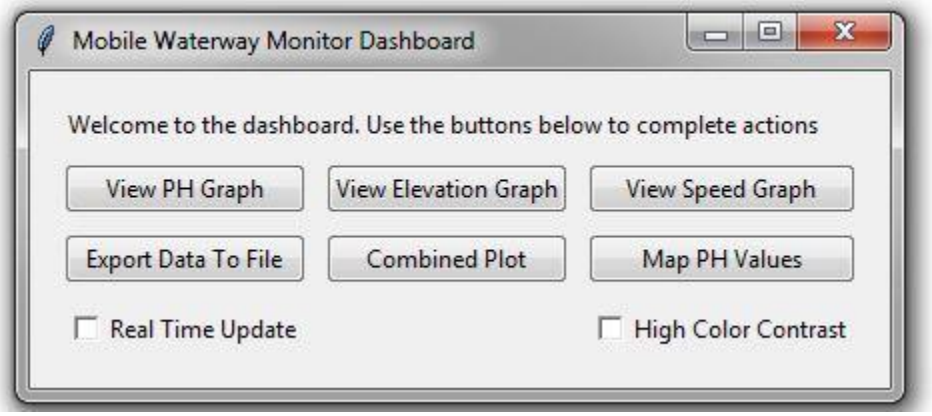


Figure 18: Python Dashboard Main Page

The dashboard checkboxes fix some of the small issues noticed in Adafruit's graphics widgets. High color contrast helps make visualization a little more eye catching, especially when presenting results or viewing from any distance. Adafruit's dark blue plot on a black background made reading feed line graphs somewhat difficult, even for young eyes. With poor lighting, a bad angle, or less than stellar visions the graphs became almost transparent. To fix this the dashboard implements a high color contrast setting. For the line plots, ticking this feature thickens the plotted line to make it more visible. For the color-coded map (discussed later), this changes the background color of the map. This makes both graph types easier to see. The real-time update option also fixes a simple deficiency that the Adafruit interface had, the ability to select whether to view a static plot, which describes a set of data at a moment in time, or a dynamic plot, which allows for viewing of data as time elapses. Having the ability to select which dataset to view allows for closer inspection when strange trends develop and saving the graph when interesting plots occur.

Viewing the pH graph, elevation graph, or MWM speed graph is as simple as pressing their dedicated buttons on the welcome dashboard. Below will include examples of all three graphs to give an idea of the scale, units, and what a waterway reading might look like when displayed through the Python GUI. These three graphs also benefit from the high color contrast, which widens the plot line, and real-time updates which refresh the graph with new data around every 10 seconds.

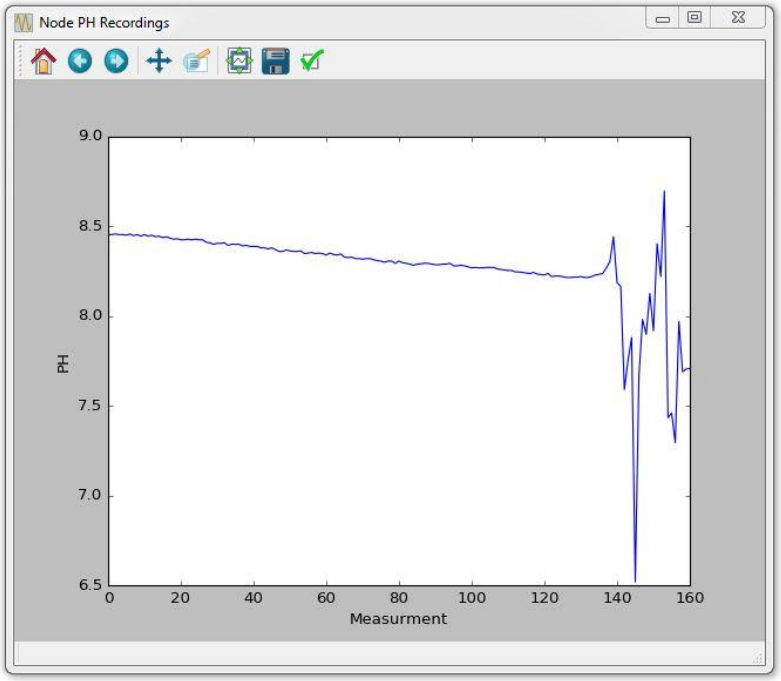


Figure 19: Python Dashboard pH Line Plot

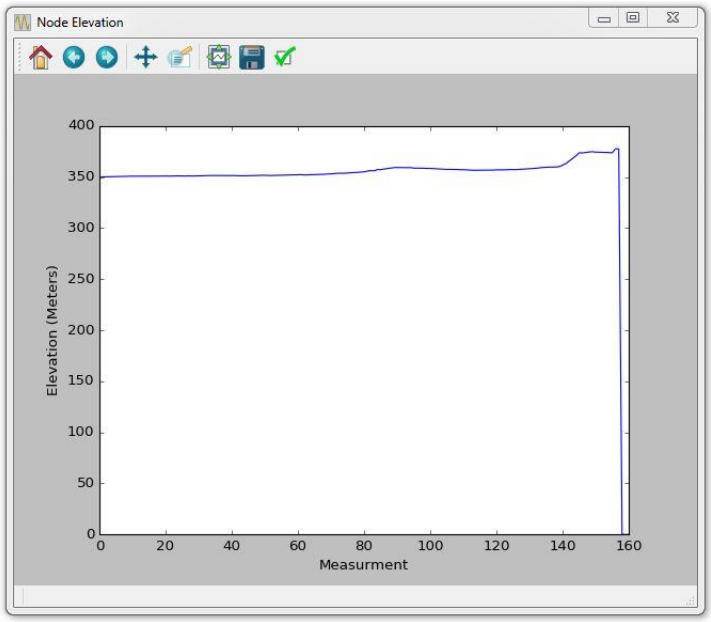


Figure 20: Python Dashboard Elevation Plot

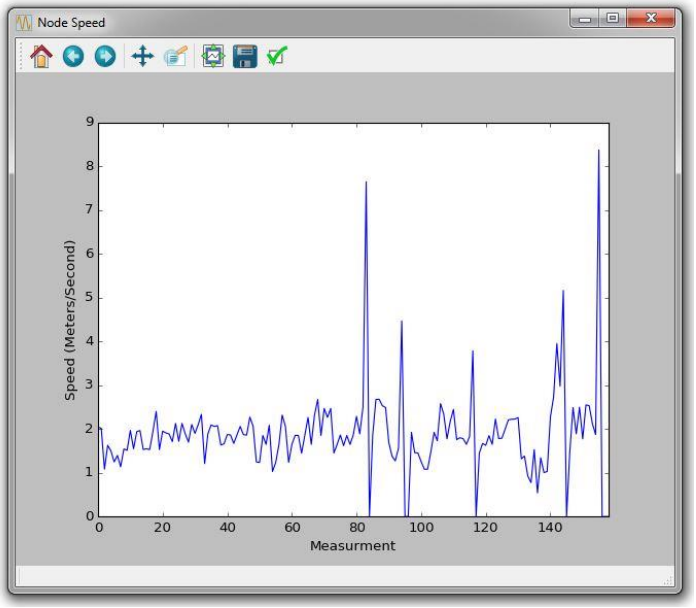


Figure 21: Python Dashboard Speed Plot

The x-axis of these plots all use the same indexing value. Instead of using time, which is specific but often smooths trends, the plot uses the measurement count. The measurement index correlates time to the measurement instance in a much more manageable way. The most recent measurement is indexed at zero. The least recent measurement has the greatest index value, for instance around 160 as seen in the above example. The indexes of all points will continually increase over time. Correlating index to time might seem slightly difficult, but considering that the node sends data about every 10 seconds, translation isn't all that difficult, and as described before, exporting the exact data is possible if more precise representations are desired.

Even better for noticing trends over time is the combined plot feature. Selecting this option from the dashboard displays all three line graphs (pH, elevation, and speed) on a single page for easier comparison. The idea behind including this plot is to allow trends between data sets to be easily seen. With future sensors added, viewing trends between water oxygen content, clarity, pH, and other measurements like salinity would be possible. Having multiple data sets on a screen at a single time could assist dramatically in discovering interdependencies between measurements in a waterway. Due to how data is packed, this feature isn't possible on the Adafruit website. Adafruit's website does not allow unpacking of data and calculation on different transmission parameters. Currently, the line plot widget on the website only uses the main value, which in this case is pH, as a variable on the line plot. The website also doesn't feature the ability to calculate values from changes between two measurements. Having a combined plot allows transmission specific, time varying calculations to be displayed on the same screen. The combined plot can update in real-time like the other single plots and can have a thicker line color through the high contrast checkbox. The combined plot is shown below for example. It does have the same window as the other plots. The window has been omitted in this example to allow for a larger display of the graphing area.

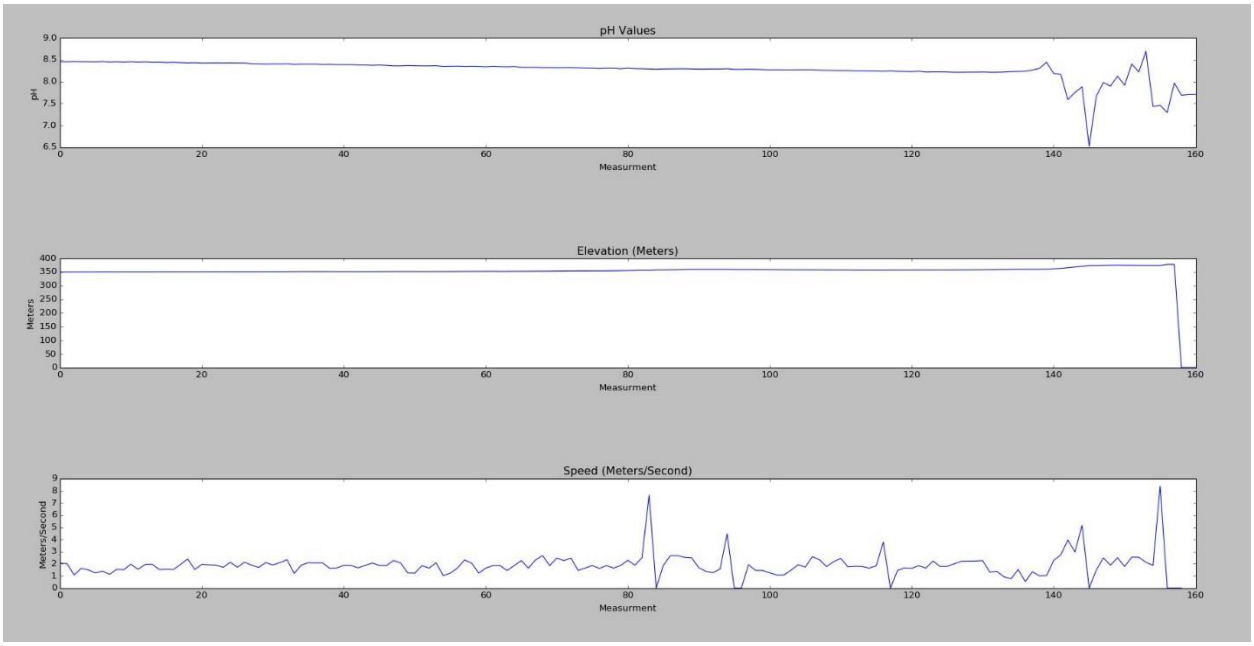


Figure 22: Python Dashboard Combined Plot

The final data visualization tool is the color-coded, pH map. Clicking on the button labeled "Map pH Values" pulls up the colored display. The map uses a range of 10 colors to represent the various pH values recorded. The pH values are displayed as color-coded dots at the location of where they were captured. This makes understanding the mapped data instantaneous when compared to the Adafruit interface. Instead of clicking on a point to view specific data as in the case of the Adafruit map, the Basemap graphic allows quick comprehension of measured pH values over distances. The Python map parameters are configurable to a specific location in code so if used in a different region only two coordinate pairs would need to be changed. Additional characteristics of this map are configurable such as the level of roadway detail (omitted due to time constraints) and the colors of regions. The map which appear when the interface button shown below is pressed. The standard coloration for the map is of earth tones. The map can be changed through the high contrast checkbox. This produces the second example below, with a simple khaki background.

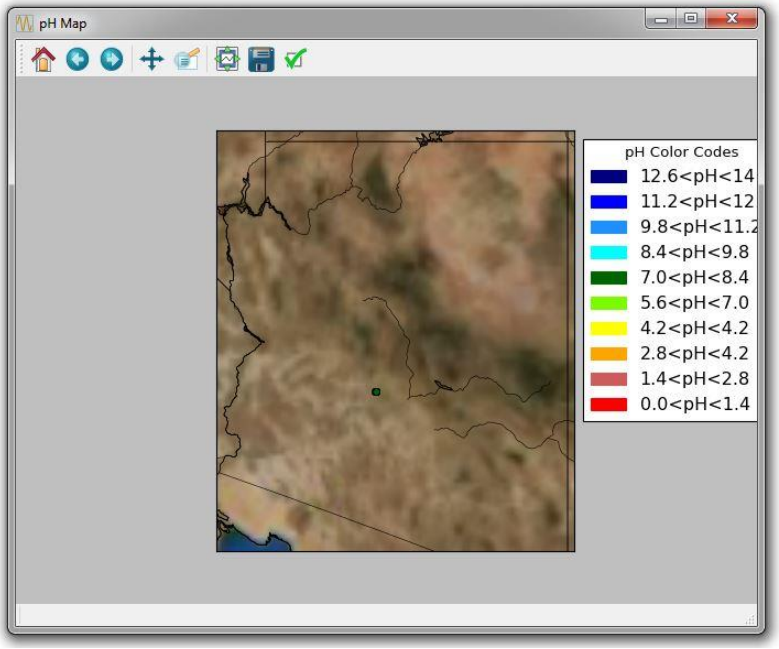


Figure 23: Python Dashboard pH Map

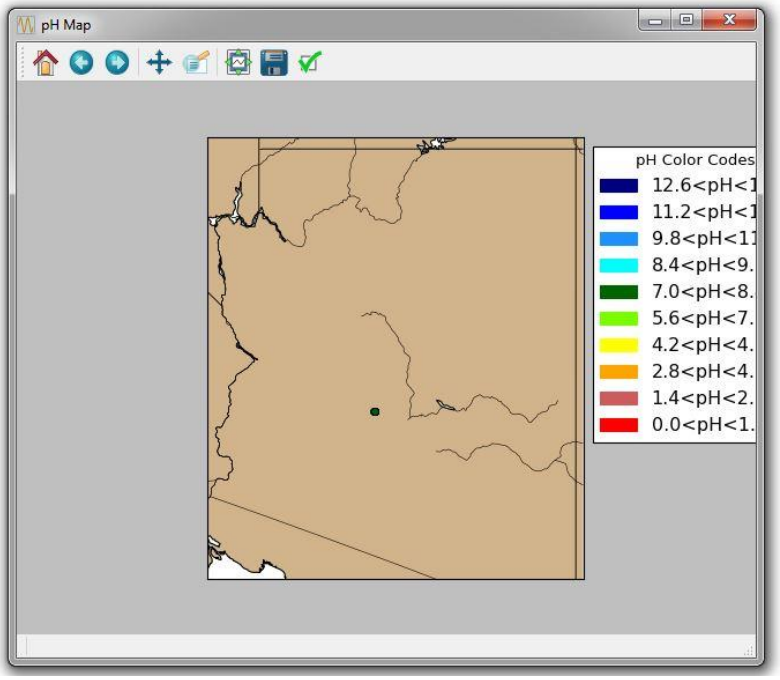


Figure 24: Python Dashboard pH Map (High Contrast)

The map benefits from the fact that Basemap is built on top of Matplotlib. Wrapping the basic plotting library allows the maps to have significant zoom. Without zoom, the maps would be almost unusable. Below shows the zoomed in map of pH values transmitted from the real-world demonstration described in the following section. Upon comparing this to the Adafruit map, it should be evident that it is identical in its shape. However, the one shortcoming of this Python map, unlike the line plots before, is that it hasn't been animated. Several attempts were made to animate this plot but the Basemap function did not work with any changes tested.

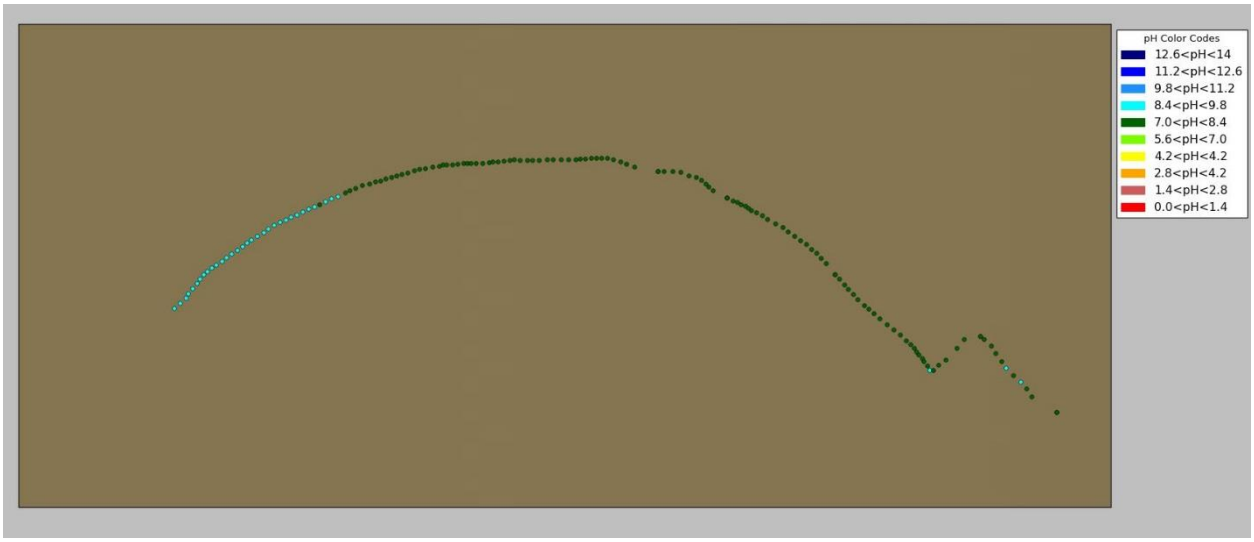


Figure 25: Python Dashboard pH Map Magnified

Overall, the current prototype collects an enormous amount of data. Both Adafruit and the Python dashboard have fantastic tools which allow painless and immediate viewing of the Mobile Waterway Monitor's measurements. The process is simple. Power on and set adrift the Mobile Waterway Monitor and either open the Adafruit website or execute the Python dashboard. Both tools allow for a wide range of data visualization, and when used in conjunction, present a holistic view of collected measurements.

The appendix will hold the Python dashboard code and repository weblink, the Adafruit viewing link for my current accounts data (which is public), and raw data. This raw data will be in three forms. The first being a complete export of my Adafruit data in a JSON file. Feed specific CSV files will be provided and so will the Python dashboard export.

Chapter 4.2 Demonstration Materials

Demonstrating the node was one of the highlights of the Mobile Waterway Monitor project. Seeing several months of work come together as imagined, and function as planned, was truly exciting. The capabilities of this node are evident from the testing.

Testing of the mobile waterway monitor took place on March 8th, 2017. With a high of around 81°F and a nighttime low of 48°F, the day was about perfect for recording the demonstration. The humidity was around 40% on average, and there was a slight breeze the whole day. The demonstration started along a spillway following the Arizona Canal Trail located just south of 63rd and Bell avenue. The node floated approximately a mile to near 80th and Bell avenue.

I chose this waterway as it both has a swift current, is highly directed, flowing through a man-made channel, and has diverse plant life along the route. The spillway is a relief channel for the nearby canal. This is also noteworthy because water does not always flow through this pathway. Seasonal water flow could have a large impact on measurements, along with the plant life, and nearby parks. The heightened human interaction near this spillway could create an interesting shift in waterway speed, pH, and physical obstacles, all of which test the node.

The graphs and website images in this document were generated from data collected during this demonstration.

To start, and this hopefully is seen on the pH graph, the node was powered on at my vehicle. I then walked down an embankment to place the node into the spillway. This location is about at the second right hand angle, a third bright blue point appears there faintly, in the map above.

The physical layout of the spillway is also nearly ideal. The first few hundred yards are a simple straight section. Following this is a slightly curved section. After curving around the node

passes under a small pedestrian footbridge and then flow into a slightly steep banked section. From here the small canal leads into a rocky section. After this rocky section is a 5-foot waterfall. The varying terrain allows the node to experience several degrees of challenging terrain. The prototype navigated these challenges with ease. Instead of describing these things and leaving a vague image, video links will be included below which show the node in action. Several pictures were taken but the majority turned out blurred. A single, crisp image is shown below. The video is a much better representation of the monitors abilities.

Figure 26: Mobile Waterway Monitor Demo

Video	Description	Link
1	River Straight	https://www.youtube.com/watch?v=Ln7J9N9Wy74
2	River Curve	https://www.youtube.com/watch?v=Vog97NSk7RQ
3	Under Footpath Bridge	https://www.youtube.com/watch?v=5fVBmiIp5rk
4	River Fast	https://www.youtube.com/watch?v=GmcxGIoD_QI
5	River Rock	https://www.youtube.com/watch?v=ZqsPZ7LWxhU
6	Waterfall	https://www.youtube.com/watch?v=Vp-R6WNM4nY

I am beyond happy with the results. The Mobile Waterway Monitor survived more challenging terrain that I even through possible. Future revision could build upon this design to reduce weight, increase buoyancy, and increase reliability.

Chapter 5 Future Extensions

Chapter 5.1 Stretch Goals

Chapter 5.1.1 River Node

Two successive senior year semesters were simply not enough time to complete everything I would have like to implement on the Mobile Waterway Monitor. And sadly, the web server, the MWM node, and the Python script all have some shortcomings due to the tight schedule.

Expansions for the Mobile Waterway Monitor node include implementing a disabling and reenabling interrupt, modulation of transmission of battery life, and onboard data storage. Modulating the cellular transmissions based off battery life would have been the simplest of the three to add. I think the idea is brilliant and would extend the operational time of the system significantly. Modulating transmissions by both the battery life and solar panel output is an even better solution. Having two variable modulation would consider the current state of the battery and the additional charge incoming to the battery. Simply regulating the transmission rate based off the battery is a much simpler solution as the battery life is already reported to the system; however, this single variable regulation does not consider the possible charging state of the MWM. Implementing the battery life transmission regulation scheme isn't as simple as a single conditional statement though. The reported battery percentage fluctuates quite a bit between reads and would require software buffering. The second scheme would require both software buffering of the battery life but also a function which maps the battery life and solar input to a specific transmission rate for a certain number of transmits. The main problem with slowing down the transmission rate is that it requires directly slowing down the rate of capturing data. Mitigating this issue can be accomplished in two ways. First, implementing a self-hosted Waterway Monitor server which

wouldn't limit the transmission rates. Creating a MWM specific IoT server would eliminate many additional problems like data storage limits and inflexible message formats, but it would take quite a bit of time to implement. The second solution relies upon additional on board storage capacity. If a transmission buffer could be created off the board so it is both non-volatile and large, then data could be recorded at a continuous rate but only sent out as able. This buffer would work as a simple queue. The first into the queue would be sent first while the newest message is sent last. This buffer could need to be very large though. For example, if battery life was low enough to reduce the transmission rate to one tenth of the collection rate, the buffer would grow nine messages for every one sent. A small buffer could quickly overflow. Onboard data storage using an SD card is critical to the implementation of a sizeable message buffer. But if the message buffer is buffering most the data, it makes more sense to dump all data to the SD card and transmit at larger intervals. A larger transmission gap does lead to the tradeoff in which fewer points are reflected on the server and Python dashboard. This would reduce the near real-time capabilities of both. Storing all data on board would reduce the power consumed by transmission while again removing the data read limit imposed by free and graciously hosted web services. Powering an SD card breakout, would add power draw, and it is not exactly clear which one is more efficient, however it is much more likely that powering a simple memory card is more effective. A disabling and reenabling interrupt is the final upgrade to the current Mobile Waterway Monitor which still can be added. This interrupt can be achieved by using the FONA 808 breakout board and its 16-pin interface. The ring indicator pin, abbreviated 'RI' on the board, can be configured to pulse when a call or text is received. The pulsing pin can be connected to the Arduino which can trigger a sleep interrupt based off the pulse. Only certain pins can support a hardware based interrupt on the Arduino Uno, but these pins could be set aside for this purpose specifically and only be populated by interrupt wires.

The interrupt would allow for directly controllable power consumption. The manually triggered interrupt could be used to ignore recently recorded regions where the monitor is passing through or disable the Mobile Waterway Monitor before entering a known cellular dead zone. For instance, if a waterway is particularly long, then the node could traverse the river once, recording the first half. Once retrieved and released at the head of the river for a second time, using a speed estimate from the previous traversal the Waterway Monitor could be called to awaken from sleep around the halfway mark to record the remaining portion of the river.

Chapter 5.1.2 Receiving Server

Given the current state of Adafruit's IoT site I do not believe there was much I could add to the server side. Currently, the IoT website is in beta. Currently in a semi-public release features are not fully hammered out. Several downsides to using a service in prerelease state include the fact that APIs can change and the websites direction could shift. By utilizing the websites current functions the Adafruit service will grow at a quicker pace while developing to meet future needs because of hosting real-world datasets. When the fully developed website is released, I have no doubt that the API will be extremely powerful and allow for a lot of flexibility. The current MWM prototype has not fully taken advantage of the Adafruit IoT API. This is mostly because of a large API and service changes which have been happening about every 5 weeks. The API has been stable for a few of these past releases but some significant changes to the API did happen previously.

Chapter 5.1.3 Python Script

The current Python dashboard which is somewhat dependent on the Adafruit website API has also had some setbacks because of upstream changes. As currently developed, the dashboards numerous line plots will update in near real-time. Mapped points using the Basemap library, do not. It should be possible to get the pH map animating points on a time interval. I haven't been able to successfully animate the Basemap display. Given more time or someone with a more complete Python skillset, redesigning the mapping function calls to animate the points over the map should be possible. Shortcut keys could also be implemented in the Python script. I haven't implemented them outside of having the escape key exit the dashboard. Having shortcut key mappings like the spacebar pausing the plot for further inspection when timed updates are enabled would be a much better interface than checkboxes which require exiting the graph. Additional keys could be defined for a user triggered update, high contrast color patterns, and recoloring the line plot. These small tweaks to the Python dashboard and Mobile Waterway Monitor node would lead to a much more capable system which is smoother to operate and more independent.

Chapter 5.2 Future Expansions

Chapter 5.2.1 River Node

There has been a surprising amount of interest in this project by professors and company representatives abroad and all have had neat applications for the Mobile Waterway Monitor. I will dive into these after expanding upon the future expansions that Dr. Vrudhula, Dr. Wu, and myself identified. There are three areas of future improvement the three of us have identified beside the typical improvement areas such as smaller form factor, improved power efficiency, and more accurate recordings. The first area for large improvement gains is a more controllable power management system. The addition of random movement abilities would also enhance the capabilities of the MWM node by distributing data collection over a river more evenly. Finally, data compression with large data sets could shorten transmission periods and associated transmission cost.

Introducing a more controllable solar system would allow a few large areas of improvement. The Mobile Waterway Monitor takes measurements at a specified interval until no power remains. As seen in previous sections, this interval is about 9 seconds for a full transmission. Working together the solar panel and battery aim to provide the necessary power for as long as possible. Therefore, if the battery and solar panel can supply 5 hours' worth of power to the MWM's various boards, the device will run for 5 hours. All 5 hours must be sequential. The runtime cannot be spaced out to cover a 9-hour period with hour long gaps between each hour of collection. A solar charge controller which could be signaled to cut and restore power based of battery life would allow this device to have near 24-hour monitoring capabilities and an around the clock water presence. Smart power management which uses battery charge and solar input to determine the Waterway Monitor's uptime should be able to charge the device throughout the

daytime and reduce transmission frequency at night for complete monitoring without need for manual charging. Controllable power delivery would also allow non-sequential time spans to be recorded instead of the current scheme which is constant monitoring over a period. Even better would be a power management system which can be configured by users to perform differently at different times of the day to increase or decrease sampling rate. For example, the pH might change significantly during dawn and dusk hours when light is rapidly changing. This effect could be studied for several days or even weeks with a rationing, user configurable power system which mostly charges the MWM during the daylight hours, puts to sleep the device during the nighttime hours, and performs frequent sampling during dawn and dusk hours. To accomplish this another small microcontroller would be needed to power on and off the system at intervals. The microcontroller would likely need to be custom made to reduce power draw to a bare minimum. The idea of customizable power management was avoided in the Mobile Waterway Monitor prototype as it consumes too much of the overall time budget for a single, very focused task. With a longer duration, smaller project scope, or a team of engineers, this reach goal would be feasible. The inspiration for this idea follows from Ben Guadette's water monitoring system which regulated power in a user defined way.

A random movement system would add increased flexibility and waterway coverage to the Mobile Waterway Monitor. Implementing a passive waterway movement system to the MWM would widen water coverage of the device by repeatedly kicking the device out of the current. The MWM would then drift back into the main current stream after some time to once again be ejected. Creating a passive and random movement system would be incredibly complex. A more realizable compromise would be having the MWM Arduino controller triggering a random movement motor, or solenoid, or other system, would allow the monitor to produce a more dispersed monitoring

pattern around a moving body of water. Random bursts of movement would be even more helpful in instances where the node would be used to monitor a lake. Instead of the node floating around aimlessly or being anchored to a central spot, which reduces mobility, having random movement would likely produce a good mapping of the lake. Standing water movement could not rely upon flowing water for energy and would more likely need a propulsion system attached or integrated into the MWM node. Surging movements could help the node in a couple of ways. If the node was to get caught when floating down river on a small branch or rock, forced, random movement could help it get unstuck without human interaction. While the node isn't likely to get stuck often due to its compact size, it is still a possibility. A possibility that would cost additional time and resources to fix. Random movement with enough of a sharp jerk could also be enough to kick the node from a slow current. This could help push the node into pooling water, which might traditionally be ignored by a fixed position monitor and bypassed by the current implementation of the Mobile Waterway Monitor.

Data compression, which was not implemented, could equate to massive power savings. Reducing the number of transmitted bits would cut communication time and energy. Communicating over a wireless network is a power intensive action and reducing the duration of transmission could shave off a lot of the power cost. Compression does increase power needs in other areas. The act of compressing data requires significantly more processor cycles than sending uncompressed values. Compressing data on a rather powerful chip is still much more power efficient than transmission given modern processors high levels of efficiency. Data compression libraries which run on an Arduino Uno seem rudimentary compared to some compression algorithms on larger chips. Mostly for this reason, implementing compression was dropped. If data compression was implemented the new transmission format would require a self-hosted server as

both Adafruit's and Sparkfun's IOT websites expect raw, uncompressed data. Instantiating a self-hosted server would be time consuming and have a significant learning curve.

Many ideas for uses and modifications to the Mobile Waterway Monitor have dripped in through Dr. Vrudhula's and Dr. Wu's travels and industry connections. The first are based off suggestions from a SRP representative. The second, from a professor researching green fuels.

Arizona's Salt River Project (SRP) provides both power and electricity to much of the state. SRP uses large canals to move water from the Colorado river, amongst others, to cities like Phoenix, Tempe, and Mesa. By design this provides some serious challenges. Monitoring hundreds of miles of canals using stationary probes is largely unfeasible. It creates the need for significant infrastructure and maintenance requirements. A small, portable water monitor is of great interest to them as it would reduce monitoring and testing costs while alerting to canal issues. The current Waterway Monitor doesn't satisfy all the measurement types which SRP is interested in. Adding things like a luminosity sensor and oxygen sensor could be an easy addition which provide a large amount of data. More of a stretch, but still possible would be adding a water clarity sensor. This would require the node to take up more depth and would only be measuring water clarity amongst the first few inches of water, which might not be enough to justify the added cost and complexity. Also, shrinking the footprint of the monitor would be greatly helpful. This would allow the device to pass through grating and over spillways much easier than the already small prototype. The final design change would be how offloading data is handled. Assuming that there are stationary monitoring stations along canal routes, changing the MWM to instead take advantage of this infrastructure for communications could save a significant amount of power. The modified Waterway Monitor could store data measurements and then, when in range of a base station, could quickly offload all its data to the permanent station. Having the Mobile Waterway Monitor firing

burst of data to base stations would require significant modification to both the floating node and stationary monitors which receive and further transmit the data.

A more radical design shift from this comes from a Venezuelan professor who suggested a modified version of the mobile waterway monitor for alternative fuel production, most notably with algae. Producing biofuels from algae is a newly rediscovered area of research. For most small applications using ethanol or biodiesel yield better results [23]. Ethanol can be produced from corn, which in most parts of the United States, is in no shortage. Fermenting small batches of ethanol can be accomplished in a straightforward manner. Biodiesels requires a similar alcohol base, typically methanol, to be mixed with an oil like cooking oils or greases. Adding used cooking oil reduces the cost for small scale biodiesel use drastically. However, both alternative fuels can become problematic at scale. Enter algae. Algae based fuels are easily made at scale. The only limiting factor is the area dedicated to controlled algae growth. The Waterway Monitor aids controlling algae growth conditions by introducing mobile and persistent monitoring at a low cost to growers. The MWM's small footprint doesn't require much area and block sunlight from the algae which requires sunlight to grow. Modifying the design of the Waterway Monitor slightly could lead to significant performance improvements. First, a taller case which sits deeper in the water would allow components to be stacked. Stacking components would reduce the surface area required by the node significantly. Also, a deeper case would allow the pH sensor to be oriented vertically downward. This would in turn reduce the unidirectional tendencies of the node (where the pH probe acts as a rudder) and would measure pH at increased depth which tends to be more accurate and stable. A small and controllable movement system could also be affixed to the node. Most ideas revolve around a small propeller which can be directed to push the enclosure around. This might not work if algae have deep and wiry roots which easily tangle in propellers. Controlled

algae tanks do remove some strict design limitations. The Mobile Waterway Monitor was designed to require very little water depth to achieve floatation so that varying river depth is not an issue. Man-made algae tanks would not have this issue and so a much deeper case, as mentioned above can be used. Because the current MWM enclosure is shallow, the sensors also occupy space based on how protected from damage they would be. Both design considerations could be ignored in controlled tanks. The algae specific node could take advantage of the more relaxed requirements and the characteristics of algae to produce a highly effective system which is not far removed from this system.

Chapter 5.2.2 Receiving Server

As mentioned previously, data compression would be powerful for this energy constrained device. Well, this data would need to be decompressed once received. Decompression would be the job of the server. Decompressing, manipulating, and directing data in a server is currently not with either Adafruit's website [21] or Sparkfun's [22]. Both sites required the data in a certain format for correct processing and storage, especially Adafruit. However, the power of having a set storage standard does allow for Adafruit to have awesome widgets which are quick and simple to use. Self-hosting would remove data caps for the monitor. As it stands, both Adafruit and Sparkfun have a limit on how many transmissions can happen per hour. While the limit is high, it might not allow for the level of detail needed. The amount of measurements which can be stored is also limited. Both web services discard the oldest data as the storage limit is reached to free space for new data. A self-hosted server could empty its storage at the beginning of monitoring, or just simply have more dedicated storage. Self-hosting isn't as ambitious of a goal as it once was,

especially for IOT projects. Sparkfun has publicly released their platform, called Phant. Adafruit will likely do the same once they move out of beta testing.

Chapter 5.2.3 Python Script

The Python dashboard could be improved in several ways. Almost all revolve around presenting data accurately and intuitively. As it stands, the mapping library presents a map which is hard to orientate on. Also, exporting data to a more familiar format like CSV would prove handy for large scale processing with macros. A user interface which could be threaded, to allow multiple plots and maps at once, would be a lofty goal, but one that would completely change the dashboard for the better.

Adding zip code boundaries, highways, and surface streets to make map reading easier can be achieved using shapefile data. Shapefile/GIS data is available online for free. One such website, which has shapefile data for every US state, is landstat.com. Utilizing the GIS data in Basemap (the Python mapping utility) is a great future feature that would help make the color-coded pH map more intuitive.

While exporting data in the Python dashboard is intuitive, sometimes the resulting export isn't. Exporting data in the current dashboard state results in a text file and while the text file is visually organized, with columns containing data being well spaced, it isn't optimal for machine processing and use. Adding formats like CSV, JSON, or even XLSX for direct importation into Excel would allow users greater control over data analysis while reducing the need to massage the data format between applications. Dumping data into a common program like Excel or Libre Office (two spreadsheet tools) allows users who are not familiar with Python programming to

manipulate data effectively. Broadening the user base almost always results in the creation of better tools and plugins.

TkInter is used extensively in the MWM's Python dashboard. Based off Tk, the tool provides a plethora of convenient interface items like buttons, checkboxes, textboxes, and layouts. The most significant downside to TkInter that I have been able to find is that it is singly threaded. The GUI thread is traditionally the only thread running. This creates some problems when trying to run background tasks. Some work arounds do exist, but they all seem to have their own problems. One example of these work arounds was to have a background task which self-schedules itself after executing for a few seconds later. This works fine for a background update when the program is running, but when the dashboard is closed the Python script doesn't complete because of the self-schedules task will constantly need to execute. As it stands now, the dashboard script only pulls new data when a graph is opened, or when a graph is opened and the real-time checkbox is ticked. The real-time checkbox will update the opened graph every 10 seconds, which isn't exactly quick, but matches up with the Adafruit websites maximum data push rate. A multithreaded GUI would allow data updates every 10 seconds without any plots or checkboxes needing to be selected. I prefer the idea of this to the current implementation as it would reduce the draw time for plots and graphs, which would produce a better user experience.

Appendix

Source Code

Notes

Included below is the code which comprises the first release of both the Mobile Waterway Monitor node and Python dashboard. Future changes may occur. These changes can be found at https://github.com/specialKody/Mobile-Waterway-Monitor.

Mobile Waterway Monitor Node

river_node.ino

Description: This file controls the Mobile Waterway Monitor node. Uploaded to the Arduino Uno, the file coordinate both the pH meter and FONA to allow for successful communication.

```
// Libraries
#include <SoftwareSerial.h>
#include <Adafruit_SleepyDog.h>
#include <Adafruit_FONA.h>
#include <Adafruit_MQTT.h>
#include <Adafruit_MQTT_FONA.h>
#include "ph.hpp"

const int ph_sensor_pin = 0;     //pH meter analog pin
uint8_t txFailures = 0;          //Counts number of sequential publish
failures
int timeSlept;                   //Record the time slept at each iteration
#define DEBUG_SKETCH    0        //If set to 1 print statements are activated

//Latitude & longitude for distance measurement
float latitude, longitude, speed_kph, heading, altitude;
//Battery value
uint16_t vbat;

//FONA pins configuration
//Note - Addtional pins can be defined for increased functionality
#define FONA_RX              2   //FONA serial RX pin
#define FONA_TX              3   //FONA serial TX pin
#define FONA_RST             4   //FONA reset pin

//FONA GSM infromation
#define FONA_APN             "epc.tmobile.com"  //APN used by cell data
service (leave blank if unused)
#define FONA_USERNAME        ""                 //Username used by cell data
service (leave blank if unused).
```

```
#define FONA_PASSWORD        ""                  //Password used by cell data
service (leave blank if unused).

//Adafruit IO configuration
#define AIO_SERVER           "io.adafruit.com"       //Adafruit IO server
(io.adafruit.com)
#define AIO_SERVERPORT       1883                    //Adafruit IO port
(always 1883)
#define AIO_USERNAME         ""                      //Adafruit IO username
(user specific)
#define AIO_KEY              ""                      //Adafruit IO key (user
specific)

//Configuring the FONA instance
SoftwareSerial fonaSS = SoftwareSerial(FONA_TX, FONA_RX);     //FONA software
serial connection.
Adafruit_FONA fona = Adafruit_FONA(FONA_RST);                 //FONA library
connection.

//Create the MQTT FONA fobject by supplying FONA class and Adafruit IO info
Adafruit_MQTT_FONA mqtt(&fona, AIO_SERVER, AIO_SERVERPORT, AIO_USERNAME,
AIO_KEY);

//Adafruit IO data feeds parameters
#define LOCATION_PH_FEED_NAME      "RIVER_NODE_LOCATION_PH" //Feed which logs
river node location and pH value
#define MAX_TX_FAILURES      3                             //Maximum number
of sequential publish failures before resetting

//Adafruit IO data feeds
Adafruit_MQTT_Publish location_ph_feed = Adafruit_MQTT_Publish(&mqtt,
AIO_USERNAME "/feeds/" LOCATION_PH_FEED_NAME "/csv");
Adafruit_MQTT_Publish battery_feed = Adafruit_MQTT_Publish(&mqtt,
AIO_USERNAME "/feeds/battery");

//Initializes the GSM and FONA connections. Also readies the watchdog timer
along with the pH meter.
void setup() {
  Serial.begin(115200);
//Initialize serial output at the specified baud rate
  #if DEBUG_SKETCH
    Serial.println(F("River Node Setting Up"));
  #endif
  ph.phSetup(ph_sensor_pin);                                    //Sets up
the pH meter connection

  Watchdog.enable(10000);                                       //Set
watchdog timeout for 10 seconds (FONA connection <10s)
  Watchdog.reset();                                             //Reset
watchdog countdown


  // Initialize the FONA module
  #if DEBUG_SKETCH
    Serial.println(F("Initializing FONA....(may take 10 seconds)"));
  #endif
```

```
  fonaSS.begin(4800);                          //FONA serial connection baud
rate
  if (!fona.begin(fonaSS)) {                   //If no connection could be made,
halt until watchdog reset
    halt(F("Couldn't find FONA"));             //Print error message when
halting
  }
  fonaSS.println("AT+CMEE=2");                 //Send command to FONA
  #if DEBUG_SKETCH
    Serial.println(F("FONA is OK"));
  #endif
  Watchdog.reset();                            //Reset watchdog countdown

  #if DEBUG_SKETCH
    Serial.println(F("Checking for network..."));                    //Print
debug status
  #endif
  while(!(fona.getNetworkStatus())) {                                //While
fona isn't connected
   delay(500);                                                       //Delay
half a second and try again
  }

  Watchdog.reset();                                 //Reset the watchdog
countdown
  fona.enableGPS(true);                             //Enable GPS
  fona.setGPRSNetworkSettings(F(FONA_APN));         //Start the GPRS data
connection.
  delay(2000);                                      //Delay 2 seconds once the
GPRS connection parameters are set
  Watchdog.reset();                                 //Reset the watchdog
countdown
  #if DEBUG_SKETCH
    Serial.println(F("Disabling GPRS"));
  #endif
  fona.enableGPRS(false);                           //Don't know if GPRS is
enabled or enabled, default to disabled
  delay(2000);                                      //Delay another 2 seconds
once GPRS module disabled
  Watchdog.reset();                                 //Reset the watchdog
countdown
  #if DEBUG_SKETCH
    Serial.println(F("Enabling GPRS"));
  #endif
  if (!fona.enableGPRS(true)) {                             //Reenable GPRS. Halt
if enable fails
    halt(F("Failed to turn GPRS on, resetting..."));
  }
  #if DEBUG_SKETCH
    Serial.println(F("Connected to Cellular!"));
  #endif

  Watchdog.reset();                                 //Reset the watchdog
countdown
  delay(3000);                                      //Delay to stabilize
connection
```

```
  int8_t ret = mqtt.connect();                          //Now attempt to make
connection with stable network
  if (ret != 0) {                                       //If connection didn't work,
reset
   #if DEBUG_SKETCH
    Serial.println(mqtt.connectErrorString(ret));
   #endif
    halt(F("MQTT connection failed, resetting..."));
  }
  #if DEBUG_SKETCH
    Serial.println(F("MQTT Connected!"));
  #endif
  Watchdog.reset();                                     //Reset the watchdog
countdown

}

//Loop will now continually read and transmit values to the MQTT server
void loop() {
  Watchdog.reset();     //Reset the watchdog countdown (just in case setup-
>loop transition takes longer than expected)

  //If not connected or too many transmit failures, wait for watchdog reset
  if (!fona.TCPconnected() || (txFailures >= MAX_TX_FAILURES)) {
    halt(F("Connection lost, resetting..."));
  }

  ph.phRead();
//Read in the PH value
  bool gpsFix = fona.getGPS(&latitude, &longitude, &speed_kph, &heading,
&altitude);    //Get GPS reading
  fona.getBattPercent(&vbat);
//Get battery percent

  log_ph_loc(ph.getPH(),latitude, longitude, altitude, location_ph_feed);
//Log PH & location to pH_Location feed
  logBatteryPercent(vbat, battery_feed);
//Log battery life to battery feed

  timeSlept = Watchdog.sleep(5000);
//Sleep system and watchdog for 5 seconds
}

//Publishes battery life percentage to battery feed
void logBatteryPercent(uint32_t indicator, Adafruit_MQTT_Publish&
publishFeed) {
  #if DEBUG_SKETCH
    Serial.print(F("Publishing battery percentage: "));
    Serial.println(indicator);
  #endif
  if (!publishFeed.publish(indicator)) {                    //Publish failed,
increment failed count
    #if DEBUG_SKETCH
        Serial.println(F("Publish failed!"));
    #endif
    txFailures++;
  }
```

```
  else {                                                    //Publish
succeeded, reset to zero
    #if DEBUG_SKETCH
        Serial.println(F("Publish succeeded!"));
    #endif
    txFailures = 0;
  }

}

//Publishes pH and location information to pH/Location feed.
//Message sends data in CSV format.
void log_ph_loc(float phVal,float latitude, float longitude, float altitude,
Adafruit_MQTT_Publish& publishFeed){
  char sendBuffer[120];                                 //Stores the message to
be send
  memset(sendBuffer, 0, sizeof(sendBuffer));            //Clear the entire buffer
  int index = 0;                                        //Reset index to buffer
head

  //pH value is sent as first value in CSV list
  dtostrf(phVal, 4, 3, &sendBuffer[index]);             //Appends float
characters to buffer (4 character excluding period with three after decimal)
  index += strlen(&sendBuffer[index]);                  //Move the index forward
by the appended length
  sendBuffer[index++] = ',';                            //Append a comma and move
index forward one space

  //Similar pattern as above for latitude, longtitude, and altitude
  dtostrf(latitude, 2, 6, &sendBuffer[index]);
  index += strlen(&sendBuffer[index]);
  sendBuffer[index++] = ',';
  dtostrf(longitude, 3, 6, &sendBuffer[index]);
  index += strlen(&sendBuffer[index]);
  sendBuffer[index++] = ',';
  dtostrf(altitude, 2, 6, &sendBuffer[index]);

  //Send the character buffer to the website over the connection
  #if DEBUG_SKETCH
      Serial.print(F("Publishing location: "));
      Serial.println(sendBuffer);
  #endif
  if (!publishFeed.publish(sendBuffer)) {                   //If publish
failed, increment failed counter
    #if DEBUG_SKETCH
        Serial.println(F("Publish failed!"));
    #endif
    txFailures++;
  }
  else {                                                    //Else, reset
counter to zero
    #if DEBUG_SKETCH
    Serial.println(F("Publish succeeded!"));
    #endif
    txFailures = 0;
  }
}
```

```
// Halt function called when an error occurs. Relies upon watchdog reset.
void halt(const __FlashStringHelper *error) {
  #if DEBUG_SKETCH
    Serial.println(error);                    //Prints the passed in message
  #endif
  while(1);                                   //Busy loop until watchdog reset
}
```

ph.hpp

Description: Declares needed class and methods to encapsulate interaction with pH meter.

```
#include "Arduino.h"

#ifndef ph_hpp
#define ph_hpp

//Defines the phClass type which contains several public functions along
which manipulate
//private data.
class phClass{
  public:
    phClass();
    void phSetup(uint8_t pin);
    void phRead();
    float getPH();
    void setPH(float phValue);
  private:
    float value;
    uint8_t pin;
};

//Declares an external ph object which can be used where this file is
included
extern phClass ph;

#endif
```

ph.cpp

Description: Defines functions and members declared in ph header file.

```
#include "ph.hpp"

//Default constructor with pin 0 assumed as well as a perfectly balanced ph
phClass::phClass(){
  this->pin = 0;
  this->value = 7.00;
}
```

```
//Setup method which overrides the input/output pin
void phClass::phSetup(uint8_t pin){
  this->pin = pin;
}

//Reads several values from the pH meter, averages, and then sets the private
//ph value data member.
void phClass::phRead(){
  int buf[10];                  //Integer buffer used to average store 10
reads
  int temp;                     //Integer for later sorting
  unsigned long int avgValue;   //Unprocessed average pH value from pH meter

  for(int i=0; i<10; i++){      //Reads in 10 pH values with a 10ms delay
between each read
    buf[i] = analogRead(pin);
    delay(10);
  }
  //Selection sort which orders the 10 values from smallest to largest
  for(int i=0;i<9;i++){
    for(int j=i+1;j<10;j++){
      if(buf[i]>buf[j]){
        temp=buf[i];
        buf[i]=buf[j];
        buf[j]=temp;
      }
    }
  }
  avgValue=0;                   //Clears the average value
  for(int i=2; i<8; i++)        //Sums the middle 6 values
    avgValue += buf[i];

  //Processes the large integer into a floating point representation which is
stored in value
  value = (float)avgValue * 17.5/1024/6;
}

float phClass::getPH(){
  return this->value;
}

void phClass::setPH(float phValue){
  this->value = phValue;
}

//Instantiates the extern ph object delared in the header file
phClass ph = phClass();
```

Python Dashboard

dashboard.py

Description: Defines the Python dashboard layout as well as links the backing functions to each GUI element.

```
# STANDARD LIBRARIES
from tkinter import *
from tkinter import ttk
import threading

# MY FILES
from dashboard_backer import *

#**************************************************************************
**********#
#                                  Functions
#
#**************************************************************************
**********#

def close_window(*args):
    """
        :param *args: allows an undetermined amount of parameters to be
passed in. No functional effect.

    Description:
        A simmple function which exits the application when the ESC key is
pressed.
    """
    root.destroy()

def plot_elev_tk():
    """
    Description:
        A Tkinter call which passes in the high_contrast checkbox value to
the plot elev function.
    """
    plot_elev(CheckVar1.get(), CheckVar2.get())

def plot_ph_tk():
    """
    Description:
        A Tkinter call which passes in the high_contrast checkbox value to
the plot_ph function.
    """
    plot_ph(CheckVar1.get(), CheckVar2.get())

def plot_speed_tk():
    """
    Description:
        A Tkinter call which passes in the high_contrast checkbox value to
the plot_speed function.
    """
    plot_speed(CheckVar1.get(), CheckVar2.get())

def plot_combined_tk():
```

```
    """
    Description:
        A Tkinter call which passes in the high contrast checkbox value to
the plot_combined function.
    """

    plot_combined(CheckVar1.get(), CheckVar2.get())

def map_ph_tk():
    """
    Description:
        A Tkinter call which passes in the high_contrast checkbox value to
the map_ph function.
    """
    map_ph(CheckVar2.get())


def update_data_tk():
    """
    Description:
        Initializes the feed data series after the GUI loads.
    """
    update_data()

#**********************************************************************
**********#
#                                TkInter Script
#
#**********************************************************************
**********#

root = Tk()
root.title("Mobile Waterway Monitor Dashboard")

mainframe = ttk.Frame(root, padding="12 12 12 12")
mainframe.grid(column=0, row=0, sticky=(N, W, E, S))
mainframe.columnconfigure(0, weight=0)
mainframe.rowconfigure(0, weight=0)

CheckVar1 = IntVar()
CheckVar2 = IntVar()

ttk.Label(mainframe, text="Welcome to the dashboard. Use the buttons below to
complete actions").grid(row=0, column=0,columnspan=3, sticky=(E,W))
C1 = Checkbutton(mainframe, text = "Real Time Update", variable = CheckVar1,
onvalue = 1, offvalue = 0).grid(column=0, row=3, sticky=(N, W, S))
C2 = Checkbutton(mainframe, text = "High Color Contrast", variable =
CheckVar2, onvalue = 1, offvalue = 0).grid(column=2, row=3, sticky=(N, E, S))

ttk.Button(mainframe, text="View PH Graph",
command=plot_ph_tk).grid(column=0, row=1, sticky=(N, W, E, S))
ttk.Button(mainframe, text="View Elevation Graph",
command=plot_elev_tk).grid(column=1, row=1, sticky=(N, W, E, S))
ttk.Button(mainframe, text="View Speed Graph",
command=plot_speed_tk).grid(column=2, row=1, sticky=(N, W, E, S))
```

```
ttk.Button(mainframe, text="Export Data To File",
command=export_data).grid(column=0, row=2, sticky=(N, W, E, S))
ttk.Button(mainframe, text="Combined Plot",
command=plot_combined_tk).grid(column=1, row=2, sticky=(N, W, E, S))
ttk.Button(mainframe, text="Map pH Values", command=map_ph_tk).grid(column=2,
row=2, sticky=(N, W, E, S))

for child in mainframe.winfo_children(): child.grid_configure(padx=5, pady=5)

root.bind('<Escape>', close_window) #Defines the escape key as a shortcut to
close the window

root.after(10,update_data_tk)
root.mainloop()
```

dashboard_backer.py

Description: Contains almost all the functions called by the dashboard.

```
# STANDARD LIBRARIES
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import pandas as pd

# MY FILES
from data_calc import *
from list_conversions import *
from map_data import map_points

dataList=[]
#**************************************************************************
**********#
#                                 Functions
#
#**************************************************************************
**********#

def update_data():
    """
    Description:
        Updates the data list as well as the feed data.
    """
    global feed_data
    feed_data =
pd.read_json('https://io.adafruit.com/api/v2/specialKody/feeds/river-node-
location-ph/data')
    feed_data['created_at'] =  pd.to_datetime(feed_data['created_at'],
infer_datetime_format=True)
    #This removes the unused data columns
    feed_data.drop(feed_data.columns[[0,2,4,5,6,9,11]], axis=1, inplace=True)

    lat = feed_data['lat']
    lon = feed_data['lon']
    dist = calculated_distance(lat,lon)
```

```
    speedSeries = list_to_series(calculate_speeds(feed_data['created_at'],
dist))

    global dataList
    dataList= [lat, lon, feed_data['ele'], feed_data['value'],
feed_data['created_at'], speedSeries]

def map_ph(high_contrast):
    """
        :param high_contrast: This specifies the color contrast of the map.
0=regular contrast, 1=heightened contrast

    Description:
        Maps the pH values on the Basemap map through the map_points function
call.
    """
    map_points((dataList[1]).tolist(), (dataList[0]).tolist(),
(pd.Series(dataList[3])).tolist(), high_contrast)

def elev_update(dump, line, ax, high_contrast):
    """
        :param dump: Believe this is needed as garbage data goes into first
parameter
        :param line: The line to be updated
        :param ax: The plot the line is currently on
        :param high_contrast: This specifies the color contrast of the map.
0=regular contrast, 1=heightened contrast

    Description:
        Updates the elevation line plot after pulling new data.
    """
    plt.cla()
    update_data()
    elevation = pd.Series(dataList[2])
    if(high_contrast):
        line = ax.plot(elevation, linewidth=3.0)
    else:
        line = ax.plot(elevation)
    return line

def plot_elev(real_time, high_contrast):
    """
        :param real_time: This specifies if real time updates are to occur.
0=static data, 1=updating data
        :param high_contrast: This specifies the color contrast of the map.
0=regular contrast, 1=heightened contrast

    Description:
        This functions plots the elevation data.
    """
    if(real_time):
        elevation = pd.Series(dataList[2])
        fig, ax = plt.subplots()
        fig.canvas.set_window_title("Node Elevation")
        ax.set_ylabel("Elevation (Meters)")
        ax.set_xlabel("Measurment")
        line = ax.plot(elevation)
```

```
        ani = animation.FuncAnimation(fig, elev_update, interval=1000,
fargs=(line, ax, high_contrast), blit=True)
    else:
        plt.figure("Node Elevation")
        elevation = pd.Series(dataList[2])
        if(high_contrast == 1):
            elevation.plot(linewidth=3.0)
        else:
            elevation.plot()
        plt.ylabel("Elevation (Meters)")
        plt.xlabel("Measurment")
    plt.show()

def ph_update(dump, line, ax, high_contrast):
    """
        :param dump: Believe this is needed as garbage data goes into first
parameter
        :param line: The line to be updated
        :param ax: The plot the line is currently on
        :param high_contrast: This specifies the color contrast of the map.
0=regular contrast, 1=heightened contrast

    Description:
        Updates the ph line plot after pulling new data.
    """
    plt.cla()
    update_data()
    values = pd.Series(dataList[3])
    if(high_contrast):
        line = ax.plot(values, linewidth=3.0)
    else:
        line = ax.plot(values)
    return line

def plot_ph(real_time, high_contrast):
    """
        :param real_time: This specifies if real time updates are to occur.
0=static data, 1=updating data
        :param high_contrast: This specifies the color contrast of the map.
0=regular contrast, 1=heightened contrast

    Description:
        This function plots the PH data. The PH data is stored as 'value' by
the Adafruit IOT website.
    """
    if(real_time):
        values = pd.Series(dataList[3])
        fig, ax = plt.subplots()
        fig.canvas.set_window_title("Node pH Recordings")
        ax.set_ylabel("PH")
        ax.set_xlabel("Measurment")
        line = ax.plot(values)

        ani = animation.FuncAnimation(fig, ph_update, interval=20000,
fargs=(line, ax, high_contrast), blit=True)
    else:
```

```
        plt.figure("Node PH Recordings")
        values = pd.Series(dataList[3])
        if(high_contrast == 1):
            values.plot(linewidth=3.0)
        else:
            values.plot()
        plt.ylabel("PH")
        plt.xlabel("Measurment")

    plt.show()

def speed_update(dump, line, ax, high_contrast):
    """
        :param dump: Believe this is needed as garbage data goes into first
parameter
        :param line: The line to be updated
        :param ax: The plot the line is currently on
        :param high_contrast: This specifies the color contrast of the map.
0=regular contrast, 1=heightened contrast

    Description:
        Updates the speed line plot after pulling new data.
    """
    plt.cla()
    update_data()
    speed = dataList[5]
    if(high_contrast):
        line = ax.plot(speed, linewidth=3.0)
    else:
        line = ax.plot(speed)
    return line

def plot_speed(real_time, high_contrast):
    """
        :param real_time: This specifies if real time updates are to occur.
0=static data, 1=updating data
        :param high_contrast: This specifies the color contrast of the map.
0=regular contrast, 1=heightened contrast

    Description:
        This function plots the calculated speed values. This requires a call
to the list_to_series fucntion and the
        calculate_speeds function.
    """
    if(real_time):
        speed = dataList[5]
        fig, ax = plt.subplots()
        fig.canvas.set_window_title("Node Speed")
        ax.set_ylabel("Speed (Meters/Second)")
        ax.set_xlabel("Measurment")
        line = ax.plot(speed)

        ani = animation.FuncAnimation(fig, speed_update, interval=20000,
fargs=(line, ax, high_contrast), blit=True)
    else:
        plt.figure("Node Speed")
        speedSeries = dataList[5]
```

```
        if(high_contrast == 1):
            speedSeries.plot(linewidth=3.0)
        else:
            speedSeries.plot()
        plt.ylabel("Speed (Meters/Second)")
        plt.xlabel("Measurment")
    plt.show()

def export_data():
    """
    Description:
        Exports the feed data to a text file. This feed data has unused
columns trimmed.
    """
    global feed_data
    a = feed_data.to_string(buf=None, columns=None, col_space=None,
header=True, index=True, na_rep='NaN', formatters=None, float_format=None,
sparsify=None, index_names=True, justify=None, line_width=None,
max_rows=None, max_cols=None, show_dimensions=False)
    f = open('dashboard_export.txt', 'w')
    f.write(a)

def elev_update_nr(dump, line, ax, high_contrast):
    """
        :param dump: Believe this is needed as garbage data goes into first
parameter
        :param line: The line to be updated
        :param ax: The plot the line is currently on
        :param high_contrast: This specifies the color contrast of the map.
0=regular contrast, 1=heightened contrast

    Description:
        Updates the elevation plot without updating data.
    """
    plt.cla()
    elevation = pd.Series(dataList[2])
    if(high_contrast):
        line = ax.plot(elevation, linewidth=3.0)
    else:
        line = ax.plot(elevation)
    return line

def speed_update_nr(dump, line, ax, high_contrast):
    """
        :param dump: Believe this is needed as garbage data goes into first
parameter
        :param line: The line to be updated
        :param ax: The plot the line is currently on
        :param high_contrast: This specifies the color contrast of the map.
0=regular contrast, 1=heightened contrast

    Description:
        Updates the speed plot without updating data.
    """
    plt.cla()
    speed = dataList[5]
    if(high_contrast):
```

```
        line = ax.plot(speed, linewidth=3.0)
    else:
        line = ax.plot(speed)
    return line

def plot_combined(real_time, high_contrast):
    """
        :param real_time: This specifies if real time updates are to occur.
0=static data, 1=updating data
        :param high_contrast: This specifies the color contrast of the map.
0=regular contrast, 1=heightened contrast

    Description:
        This function places all three line plots (pH, elevation, speed) on a
single window one above another.
        This is a seperate button as the stacking creates smaller graphing
windows.
    """
    if(real_time):
        elevation = pd.Series(dataList[2])
        values = pd.Series(dataList[3])
        speed = dataList[5]
        fig,(ax1, ax2, ax3) = plt.subplots(3, sharex=False, sharey=False)
        fig.canvas.set_window_title("Combined Plots")
        ax1.set_title("Node pH Recordings")
        ax2.set_title("Node Elevation")
        ax3.set_title("Node Speed")
        ax1.set_ylabel("pH")
        ax1.set_xlabel("Measurment")
        ax2.set_ylabel("Meters")
        ax2.set_xlabel("Measurment")
        ax3.set_ylabel("Meters/Second")
        ax3.set_xlabel("Measurment")
        line1 = ax1.plot(values)
        line2 = ax2.plot(elevation)
        line3 = ax3.plot(speed)

        ani1 = animation.FuncAnimation(fig, ph_update, interval=20000,
fargs=(line1, ax1, high_contrast), blit=True)
        ani2 = animation.FuncAnimation(fig, elev_update_nr, interval=20000,
fargs=(line2, ax2, high_contrast), blit=True)
        ani3 = animation.FuncAnimation(fig, speed_update_nr, interval=20000,
fargs=(line3, ax3, high_contrast), blit=True)
    else:
        fig,(ax1, ax2, ax3) = plt.subplots(3, sharex=False, sharey=False)
        values = pd.Series(dataList[3])
        elevation = pd.Series(dataList[2])
        speedSeries = dataList[5]

        if(high_contrast):
            ax1.plot(values, linewidth=3.0)
            ax2.plot(elevation, linewidth=3.0)
            ax3.plot(speedSeries, linewidth=3.0)
        else:
            ax1.plot(values)
            ax2.plot(elevation)
            ax3.plot(speedSeries)
```

```
        ax1.set_title("pH Values")
        ax2.set_title("Elevation (Meters)")
        ax3.set_title("Speed (Meters/Second)")
        ax1.set_ylabel("pH")
        ax1.set_xlabel("Measurment")
        ax2.set_ylabel("Meters")
        ax2.set_xlabel("Measurment")
        ax3.set_ylabel("Meters/Second")
        ax3.set_xlabel("Measurment")
        fig.canvas.set_window_title("Combined Plot")
    plt.tight_layout(h_pad=2.0)
    plt.show()
```

data_calc.py

Description: Contains functions which calculate inferred measurements from data (i.e. speed, distance).

```
# STANDARD LIBRARIES
import math

#**************************************************************************
**********#
#                                Functions
#
#**************************************************************************
**********#

def haversine_distance(lat1, lat2, long1, long2):
    """
        :param lat1: Point 1's latitude
        :param lat2: Point 2's latitude
        :param long1: Point 1's longitude
        :param long2: Point 2's longitude

    Description:
        The function returns the haversine (big circle) distances from the
two points on earth. Does not take into account elevation change.
    """
    radius = 6371000                #Radius of the earth (meters)
    lat1r = math.radians(lat1)
    lat2r = math.radians(lat2)
    long1r = math.radians(long1)
    long2r = math.radians(long2)
    deltaLat = lat2r-lat1r
    deltaLong = long2r - long1r

    a = (math.sin(deltaLat/2)*math.sin(deltaLat/2)) +
math.cos(lat1r)*math.cos(lat2r)*(math.sin(deltaLong)*math.sin(deltaLong))
    c = 2*math.atan2(math.sqrt(a), math.sqrt(1-a))
    return radius*c

def calculated_distance(latitudes, longitudes):
    """
```

```
        :param latitudes: A list of latitudes (floating point format)
        :param longitudes: A list of longitudes (floating point format)

    Description:
        This function calls the haversine_distance function for each pair of
points. Each pair is i, i+1 in the list.
        Returns distances which is a listing of haversine distances. This has
length of len(latitudes)-1
        Input parameters longitudes and latitudes must have the same number
of elements.
    """
    i = 0
    distances =[]
    for i in range(0, len(latitudes)-1):
        distances.append(haversine_distance(latitudes[i], latitudes[i+1],
longitudes[i], longitudes[i+1]))
    return distances

def elevation_change(elevation):
    """
        :param elevation: A list of elevation points (floating point format)

    Description:
        Calculates and returns the differences in elevation between two
points. This is completed for all sequential pairs in a list.
        Returns deltaElev which is a listing of elevation changes. This has a
length of len(elevation) -1
    """
    deltaElev =[]
    for i in range (0, len(elevation)-1):
        deltaElev.append((elevation[i+1]-elevation[i]))
    return deltaElev

def calculate_speeds(times, distances):
    """
        :param times: A list of datetime objects which corresponds to the
timestamps on transmissions
        :param distances: A list of calculated distances. Each I distance
corresponds to location difference between I and I+1

    Description:
        Takes a list of times and distances and returns the speed in KPH for
the river node.
    """
    speeds=[]
    for i in range (0, len(distances)-1):
        deltaTime = times[i] - times[i+1]                        #calculate
time difference here
        calc_speed = distances[i]/(deltaTime.total_seconds())    #Right now
returns the speed in meters per second
        if(calc_speed >10.0):                                    #Added if
check to remove speeds >10m/s.
            speeds.append(0)
        else:
            speeds.append(calc_speed)
    return speeds
```

list_conversions.py

Description: Simple function which converts a Python list into a Pandas series.

```
#STANDARD LIBRARIES
import pandas

#**************************************************************************
**********#
#                                Functions
#
#**************************************************************************
**********#

def list_to_series(list):
    """
        :param list: A list of values

    Description:
        This function takes a generic list of values and converts to
        a Pandas series with the iterator as the index.
    """
    dict={}
    for i in range(0, len(list)):
        dict[i] = list[i]
    return pandas.Series(dict)
```

map_data.py

Description: Contains the functionality which draws the color-coded pH map.

```
# STANDARD LIBRARIES
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
import matplotlib.patches as mpatches
import numpy as np

#**************************************************************************
**********#
#                                Functions
#
#**************************************************************************
**********#

def map_points(longitudes, latitudes, values, high_contrast):
    """
        :param high_contrast: This specifies the color contrast of the map.
0=regular contrast, 1=heightened contrast

    Description:
```

```
        This maps all points (which can be imagined as a 3-tuple) from the
longitudes, latitudes, and values list.
        The function builds the tuple by selecting a long, lat, and value
from the same list row.
        It is VERY IMPORTANT that longitudes, latitudes, and value have the
same length.
        High_Contrast determines it the map is drawn with realistic color or
a uniform tan.
    """
    map = Basemap(projection='merc', resolution = 'h', area_thresh = 0.01,
llcrnrlon=-114.85, llcrnrlat=31.4, urcrnrlon=-108.93, urcrnrlat=37.15)
    cmap1 = mcolors.LinearSegmentedColormap.from_list("my_colormap", ((0, 0,
0), (1, 1, 1)), N=14)

    plt.figure("pH Map")
    map.drawcoastlines()
    map.drawcountries()
    map.drawstates()
    map.drawrivers()
    # Use fill continents with tan for high contrast
    if(high_contrast):
        map.fillcontinents(color = 'tan')
    else:
        map.bluemarble()
    map.drawmapboundary()

    for lon, lat, val in zip(longitudes, latitudes, values):
        x,y = map(lon, lat)
        clr=ph_color_code(val)
        test=map.plot(x, y, marker='o', color=mcolors.cnames[clr],
markersize=5)

    navy_patch = mpatches.Patch(color=mcolors.cnames['navy'],
label='12.6<pH<14')
    blue_patch = mpatches.Patch(color=mcolors.cnames['blue'],
label='11.2<pH<12.6')
    db_patch = mpatches.Patch(color=mcolors.cnames['dodgerblue'],
label='9.8<pH<11.2')
    aqua_patch = mpatches.Patch(color=mcolors.cnames['aqua'],
label='8.4<pH<9.8')
    dg_patch= mpatches.Patch(color=mcolors.cnames['darkgreen'],
label='7.0<pH<8.4')
    lg_patch = mpatches.Patch(color=mcolors.cnames['lawngreen'],
label='5.6<pH<7.0')
    yellow_patch = mpatches.Patch(color=mcolors.cnames['yellow'],
label='4.2<pH<4.2')
    orange_patch = mpatches.Patch(color=mcolors.cnames['orange'],
label='2.8<pH<4.2')
    ir_patch = mpatches.Patch(color=mcolors.cnames['indianred'],
label='1.4<pH<2.8')
    red_patch = mpatches.Patch(color=mcolors.cnames['red'],
label='0.0<pH<1.4')
    plt.legend(handles=[navy_patch, blue_patch, db_patch, aqua_patch,
dg_patch, lg_patch, yellow_patch, orange_patch, ir_patch,
red_patch],loc=2,bbox_to_anchor=(1,1), title='pH Color Codes')
    plt.show()
```

```
def ph_color_code(value):
    """
        :param value: This is a pH value which is having its color
multiplexed.

    Description:
        This takes a pH value as input and returns a color to be used in the
form of a string.
    """
    if value > 12.6:
        return 'navy'
    elif value >11.2:
        return 'blue'
    elif value >9.8:
        return 'dodgerblue'
    elif value >8.4:
        return 'aqua'
    elif value >7.0:
        return 'darkgreen'
    elif value >5.6:
        return 'lawngreen'
    elif value >4.2:
        return 'yellow'
    elif value >2.8:
        return 'orange'
    elif value >1.4:
        return 'indianred'
    else:
        return 'red'
```

Schematics

Mobile Waterway Monitor Case Layout

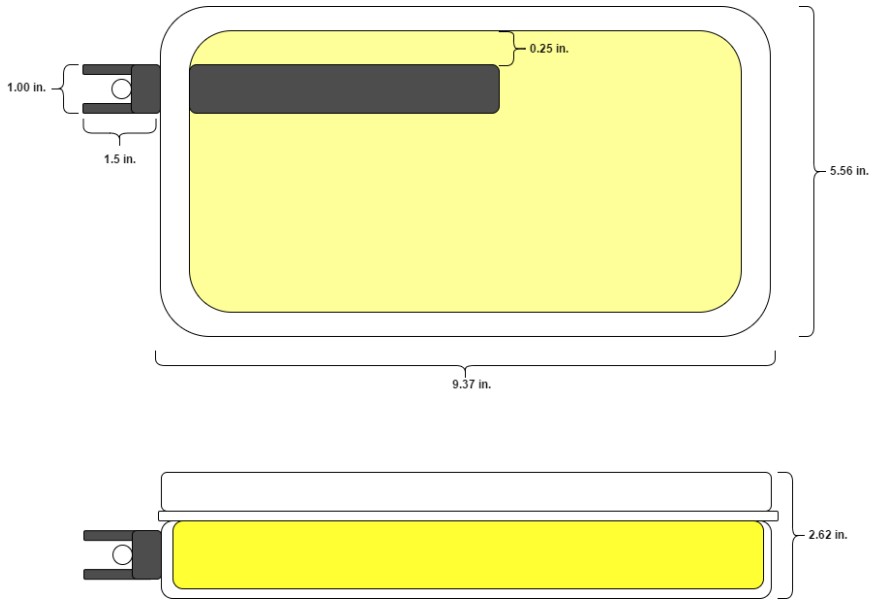


Figure 27: Case Layout

Mobile Waterway Monitor Wiring

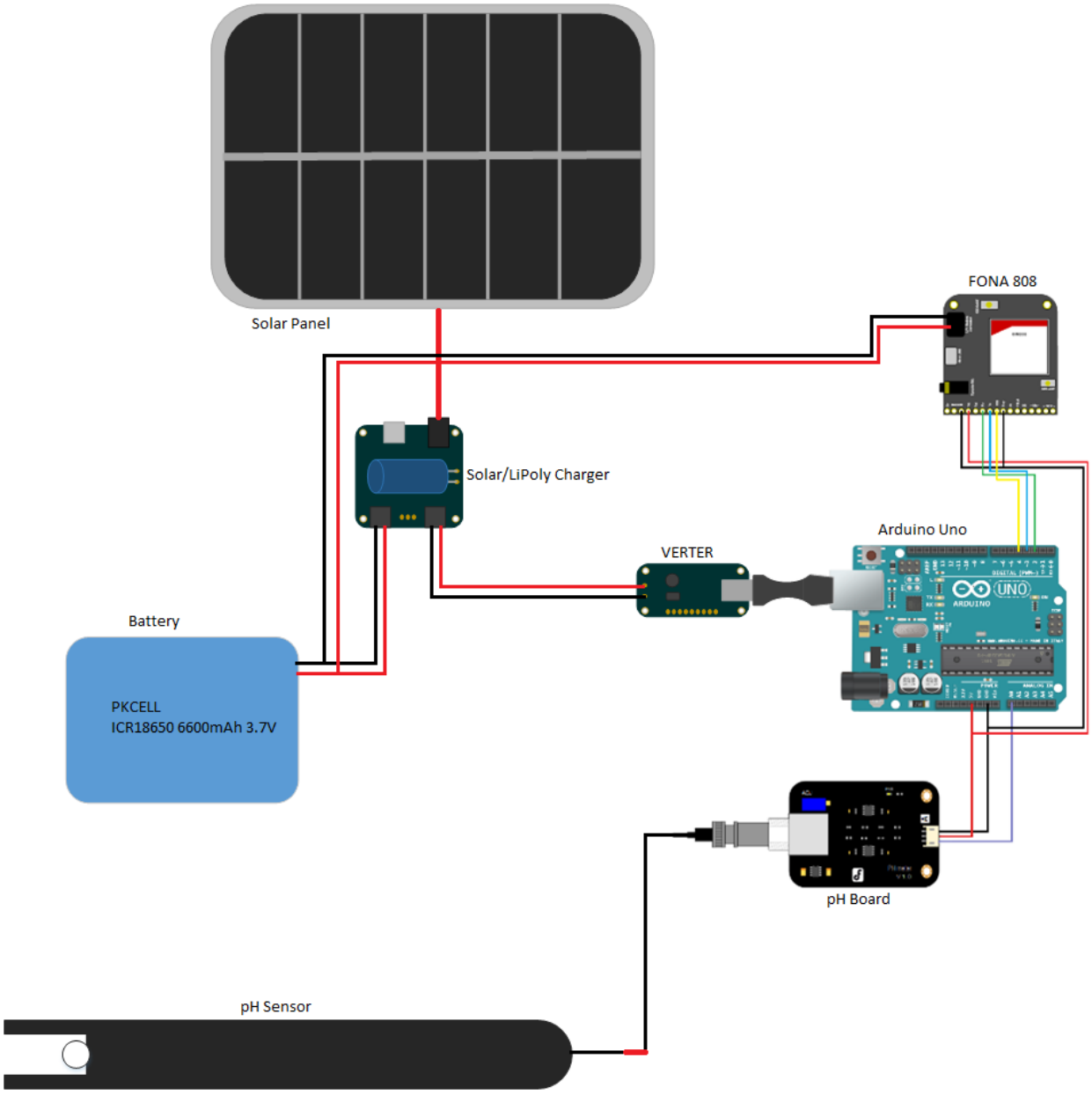


Figure 28: Full MWM Wiring Diagram

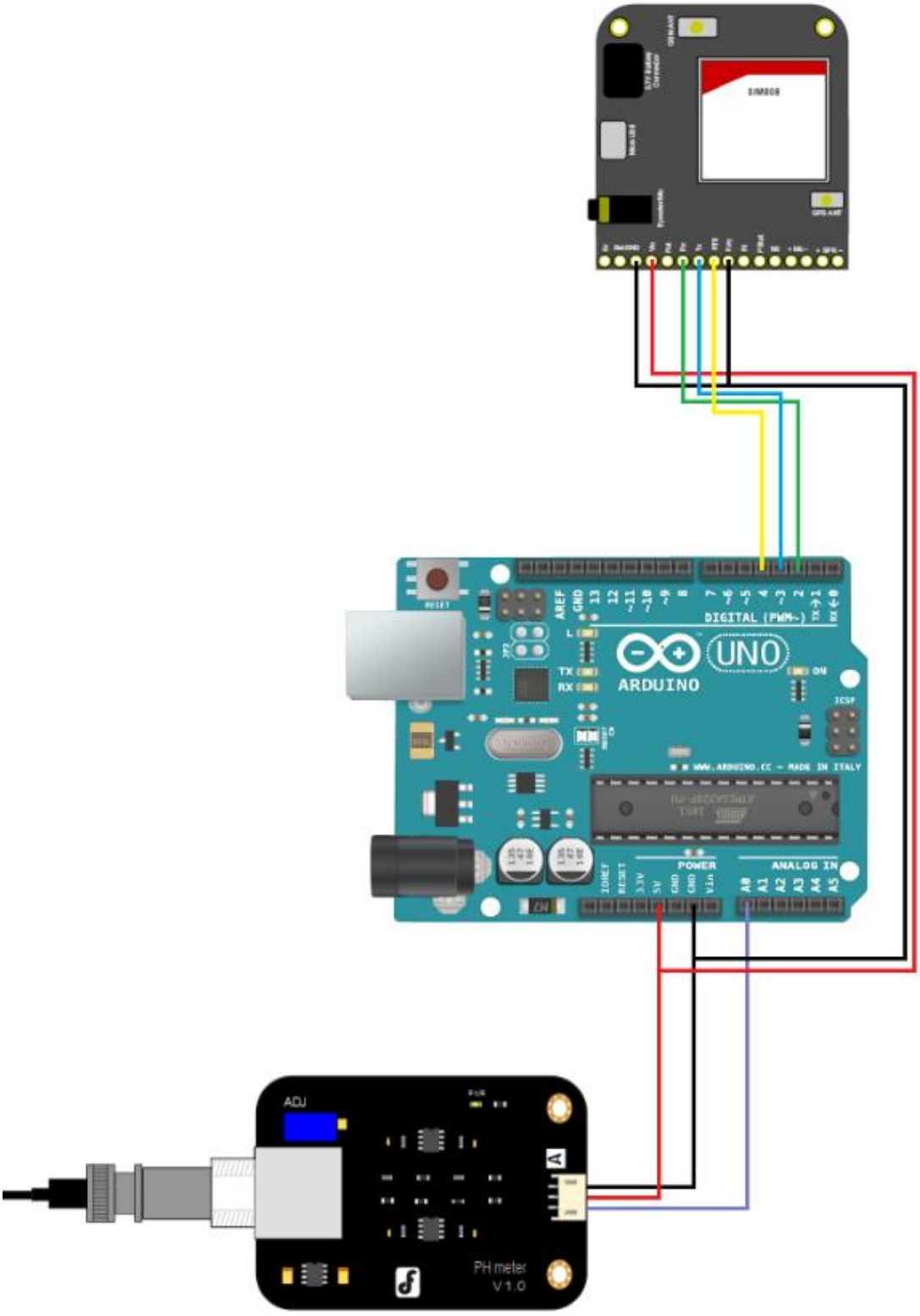


Figure 29: Detailed Wiring Diagram

Wiring Table

Number	From	To	Wiring Diagram Color
1	Arduino GND	pH board black lead	black
2	Arduino GND	FONA 808 GND	black
3	Arduino GND	FONA 808 KEY	black
4	Arduino 5V	pH board red lead	red
5	Arduino 5V	FONA 808 VIO	red
6	Arduino A0	pH board blue lead	slate blue
7	Arduino digital 2	FONA 808 RX	green
8	Arduino digital 3	FONA 808 TX	blue
9	Arduino digital 4	FONA 808 RTS	yellow
10	Solar/LiPoly 2.1mm input	Solar panel	red (thick)
11	Solar/LiPoly load JST-2 red	VERTER positive VIN	red
12	Solar/LiPoly load JST-2 black	VERTER negative VIN	black
13	Battery JST-2	Solar/LiPoly batt JST-2	black & red
14	Battery JST-2	FONA 808 JST-2	black & red
15	pH sensor (coaxial)	pH board connector (coaxial)	

Manual

Installing the Mobile Waterway Monitor dashboard and programming the node shouldn't take someone with moderate experience very long. Several steps need to be completed to run either the dashboard or program the MWM node. The process consists of installing the proper environment, staging the files, programming the Arduino, and finally running the completed system.

Installing the proper Arduino and Python environments will likely be the longest part of the process, especially for new programmers. Using standardized libraries makes installing both programming suites much simpler. To work with the Arduino, and MWM node, the Arduino IDE must be installed. The Integrated Development Environment is available for free on the Arduino website. From here the IDE will need to be opened to install necessary libraries. The Arduino library also has a wonderful guide on installing libraries which provides much more detail than I possibly can here [24]. For the MWM node the Adafruit_FONA, Adafruit_MQTT_Library, and Adafruit_SleepyDog libraries will need to be installed. The newest version of each of these libraries will work. This completes the setup for the MWM node environment. Next, download and install the folder titled *river_node* which contains the files shown below.

Figure 30: Folder Contents (river_node)

Once this folder is downloaded, connect the Arduino to the computer using the USB connection and program the Arduino by uploading the sketch, *river_node.ino*. To use the MWM, unplug the USB computer connection and reconnect the battery and voltage regulator as seen above. The device will automatically power on and start when enough power is supplied. This completes the necessary steps to program and compile the Mobile Waterway Monitor Arduino code.

Unfortunately, working with the Python dashboard and the supporting scripts is not a simple as the Arduino code. This doesn't mean that is must be painful. The Python dashboard is

created in Python 3.5 and requires a number of mathematics and science libraries to work properly. For this reason, Anaconda, a packaged version of Python with preinstalled libraries centered around math, is the preferred Python distribution. The specific version of Anaconda used when creating the dashboard was Anaconda 4.0 built off Python 3.5.1. For most package installs the command *pip install pckg_name* where pckg_name is the package you want to install will work. Additional install methods can be found in the Anaconda documentation [25]. Packages you will need include Matplotlib, Basemap, Pandas, and TkInter. In my distribution, Matplotlib, Pandas, and TkInter all came preinstalled. Each respective website also contains installation instructions if the libraries are not preinstalled. Basemap can also be installed through the command line or manually installed by a wheel file [26]. The specific version used to create the dashboard is version *basemap-1.0.8-cp35-none-win_amd64*. With these libraries installed, the python dashboard folder named *python_dashboard* can be downloaded. The folder structure is shown below. The Python dashboard can now be executed using two ways. Navigating to the folder and double clicking on *dashboard.py* is the simplest manner to run the program. The program can also be run in the Anaconda terminal by navigating to the directory and typing in the command *python dashboard.py*.

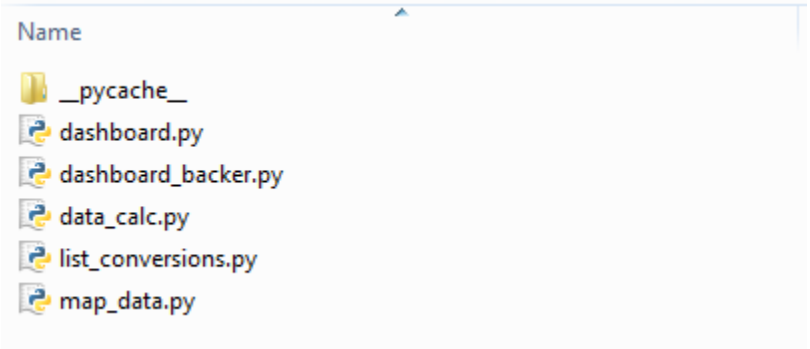


Figure 31: Folder Contents (python_dashboard)

This concludes the instructions for installing, running, and using the Mobile Waterway Monitor node and Python dashboard.

Parts List

The first list will detail the parts which are critical to this project. Without these components, the waterway monitor cannot function.

1. **Arduino Uno R3**

   Cost: ~$25 name brand, ~$5 knockoff

   Distributor:

   Arduino, Amazon, Adafruit, Sparkfun, and many others

   Major Specifications:

   ATmega328P microprocessor, 14 I/O pins, 32kB flash memory

   Documentation/Website:

   https://www.arduino.cc/en/Main/ArduinoBoardUno

   Purchasing Link:

   https://storeusa.arduino.cc/products/a000066?utm_source=redirects&utm_medium=store.arduino.cc&utm_campaign=303_Redirects

   Notes:

   This board is central to the river node. It facilitates the data exchange between GSM breakout board and the pH meter. The board has many pins which are available for future expansion both for additional sensors, LED lights, and measurements types such as monitoring solar intensity through voltage.

   If no sensors are added a smaller Arduino like the Micro could be used. A more advanced, same form factor board like the Arduino 101 could also be used. This has a couple of built in sensors like an accelerometer which could expand data collection even more.

2. **Gravity pH Sensor**

   Cost: $56.95

   Distributor:

   DFRobot

   Major Specifications:

Linear voltage through pH range, +/- 0.1pH, response time < 1 minute

Documentation/Website:

https://www.dfrobot.com/wiki/index.php/PH_meter(SKU:_SEN0161)

https://www.dfrobot.com/wiki/index.php/Industrial_pH_electrode(SKU:FIT0348)

Purchasing Link:

https://www.dfrobot.com/index.php?route=product/product&product_id=1110&search=sen0169&description=true%23.Vl07WPmqqAx

Notes:

This probe is more expensive than a similar DFRobot probe, however, it is for a good reason. The probe purchased for this project is the industrial version. It will last longer (up to a year) in many solutions. It can read the entire pH range. It can measure pH in temperatures ranging from 0° to 60°. It also came with a very long cable which could be handy for larger expansions or desktop calibration. This cable did get in the way when encasing the device. Also, a convenient voltage to pH value conversion table is available in the documentation along with the wiring diagram.

3. Adafruit FONA 808

Cost: $49.95

Distributor:

Adafruit

Major Specifications:

Quad-band GSM, receives phone calls/texts/GPRS, standard SIM card

Documentation/Website:

https://learn.adafruit.com/adafruit-fona-mini-gsm-gprs-cellular-phone-module/downloads

https://cdn-shop.adafruit.com/product-files/2637/SIM800+Series_AT+Command+Manual_V1.09.pdf

http://simcom.ee/documents/SIM800/SIM800_Hardware%20Design_V1.08.pdf

Purchasing Link:

https://www.adafruit.com/products/2542

Notes:

FONA uFL version would also work. It was out of stock when I purchased my parts. Here is its purchasing link: https://www.adafruit.com/product/1946.

This little board has a lot of functionality. I doubt I really used it to its full potential. The AT command syntax is powerful and easy to learn with the correct documentation. The board can take a while to connect but this can be overcome in the Arduino sketch easily. Adafruit has also provided a library for the board which is powerful and stable. The board itself can receive message which can help implement sleep interrupts. Also, there are many antennas available, I opted for a low dBi antenna (decibels isotropic) which directs the signal in a more distributed pattern than a high dBi antenna.

4. Pelican 1060 (yellow/clear)

Cost: $20.71

Distributor:

CaseTech

Major Specifications:

Interior (L*W*D) 8.25*4.25*2.25, 2.49lbs buoyancy

Documentation/Website:

http://www.pelican.com/us/en/product/watertight-protector-hard-cases/micro-case/standard/1060/

Purchasing Link:

https://4casetech.com/shop/pelican-micro-1060/

Notes:

Really like the color scheme of the case. The yellow should make the MWM easier to spot in the water while the clear lid allows the solar panel to be encased in the polycarbonate case. The yellow rubber liner helps protect circuitry from harsh vibrations. The purge vent seems to work very well without letting in water. It has been more than buoyant for my needs. And would likely be very good for future expansions.

5. GSM Antenna

Cost: $2.95

Distributor:

Adafruit

Major Specifications:

75mm long, 2mm thick, 3dBi GSM antenna

Documentation/Website:

None. Specifications are provided on Adafruit's purchasing page.

Purchasing Link:

https://www.adafruit.com/products/1991

Notes:

A simple, slim, and robust antenna. Seems to grab the GSM signal easily. Has a 3dBi gain. I try face the clear, green tinted size upwards through the clear polycarbonate. More friendly form factor than a SMA antenna.

6. Passive GPS antenna

Cost: $3.95

Distributor:

Adafruit

Major Specifications:

15mm*15mm*6.8mm, 50mm long uFL cable, 1dBi gain

Documentation/Website:

None. Specifications are provided on Adafruit's purchasing page.

Purchasing Link:

https://www.adafruit.com/products/2461

Notes:

Same as the GSM antenna. I haven't had any problems with this.

7. USB/DC/Solar Lithium Ion/Polymer Charger

Cost: $17.50

Distributor:

Adafruit

Major Specifications:

Charges 3.7V/4.2V LiPoly batteries with 6V solar panel while providing power to LOAD port

Documentation/Website:

https://learn.adafruit.com/usb-dc-and-solar-lipoly-charger/using-the-charger?view=all#downloads

Purchasing Link:

https://www.adafruit.com/products/390

Notes:

This is the heart of the solar system of the monitor. This device will split power between the load and battery ports. It also allows power from the battery to be used to power the load connection when the solar panel does not provide enough power. Uses JST 2 pin connections for both the battery and the load connection which is very handy. Has three color indicator LEDS as well as a low battery indicator. Sadly, it does not have any additional interfaces for intensive monitoring.

8. VERTER 5V Buck-Boost Regulator

Cost: $9.95

Distributor:

Adafruit

Major Specifications:

Regulates 3-12VDC to 5VDC output (covers both battery and solar panel input case)

Documentation/Website:

https://github.com/adafruit/Fritzing-Library

https://github.com/adafruit/Adafruit-Verter-PCB

Purchasing Link:

https://www.adafruit.com/products/2190

Notes:

Boosts low voltage input (3V<input<5V) and drops high voltage input (5V<input<12V) to a stable 5V output. Both must be direct current. Has around 90% operating efficiency. Allows for varying batteries and solar panel input. This will allow for future battery upgrades. So far it has been a good part, the JST input crimping connectors aren't that good. I would probably remove this in the future and just solder the JST 2 pin connector wires in.

9. Lithium Ion Battery (3.7V 6600mAh)

Cost: $29.50

Distributor:

Adafruit

Major Specifications:

JST 2-pin connection, 2A output, protection circuity

Documentation/Website:

https://cdn-shop.adafruit.com/product-files/353/C450_-_ICR18650_6600mAh_3.7V_20140729.pdf

https://cdn-shop.adafruit.com/product-files/353/18650_battery_pack_UN38.3__1_.pdf

Purchasing Link:

https://www.adafruit.com/products/353

Notes:

Large capacity battery for the unit. The added protection circuity is handy for voltage spikes. No overheating protection, however the unit being partially submerged in water should help keep operating conditions cool.

Also, a second set of JST connections needed to be spliced into the single pair. This allows the battery to power both the Arduino board and the GSM board.

10. Medium Solar Panel (6V 2W)

Cost: $29.00

Distributor:

Adafruit

Major Specifications:

6V/2W, 4.4” *5.4”, cell efficiency 19%+

Documentation/Website:

None. Specifications are provided on Adafruit’s purchasing page.

Purchasing Link:

https://www.adafruit.com/products/200

Notes:

Fits in the lid of the case almost perfectly. Does have to be angled in just a little bit. Can power the Arduino directly under modest sunlight. It is also waterproof (besides the output connection) and is damage resistant. Very high build quality. Takes up about 75% of the clear lid area which is perfect. This allows the other antennas and LEDs to be visible.

This secondary list contains both suggested parts. These parts are can be bought in many forms. The function matters more than the actual part itself. Some of these items are not used in the final

projects but were helpful when prototyping and others can be used in the future when a more permanent set of functions are determined.

1. **Lithium Ion Battery (3.7V 1200mAh)**

   Cost: $9.95

   Distributor:

   Adafruit

   Major Specifications:

   JST 2-pin connection, 2A output, protection circuity

   Documentation/Website:

   https://cdn-shop.adafruit.com/datasheets/503562+1200mah.pdf

   https://cdn-shop.adafruit.com/datasheets/503562+1200mAh+report.pdf

   https://system.netsuite.com/core/media/media.nl?id=1760869&c=1327152&h=dd21074dd6e7416e6499&_xt=.pdf

   Purchasing Link:

   https://www.adafruit.com/products/258

   Notes:

   Used this for testing the GSM connection before deciding which solar configuration to use. Did a great job and charged to full capacity much quicker than the 6600mAh battery which was handy when testing battery life data transmission. Don't need it in future expansions and project iterations.

2. **Adafruit Micro LiPoly Charger**

   Cost: $5.95

   Distributor:

   Adafruit

   Major Specifications:

   USB to JST 2-pin charger

   Documentation/Website:

   https://learn.adafruit.com/adafruit-microlipo-and-minilipo-battery-chargers/downloads

   Purchasing Link:

https://www.adafruit.com/products/1304

Notes:

This was much easier to charge batteries with than using the USB/DC/Solar Lithium Ion/Polymer Charger. The USB/DC/Solar charger uses a USB mini connector which was probably common 5 or 6 years ago, but is hard to find now. This charger plugs directly into a computers USB port. I would suggest charging though a computers USB port as voltage and current are already restricted compared to newer cellphone chargers which can use higher currents to speed up charge time. I would recommend this especially considering the price. It made charging the batteries much easier.

3. Adafruit Perma-Proto PCB Breadboard (half size)

Cost: $4.50

Distributor:

Adafruit

Major Specifications:

Same connection scheme as breadboard, can be soldered to for permanent mounting of components.

Documentation/Website:

None. Specifications are provided on Adafruit's purchasing page.

Purchasing Link:

https://www.adafruit.com/products/1609

Notes:

Didn't end up using this. However, this will be critical if future iterations call for more sensors and/or smaller form factor. It has the same hole connection scheme as a breadboard which simplifies mounting components. Having soldered, and highly affixed, components will be needed if the monitor is to go down any bumpy waterways.

4. Male/Male Jumper Wires

Cost: $1.95

Distributor:

Adafruit

Major Specifications:

28 AWG, 6" long

Documentation/Website:

None. Specifications are provided on Adafruit's purchasing page.

Purchasing Link:

https://www.adafruit.com/products/1957

Notes:

Convenient prototyping wire. Other wire could be used. This does have the nice characteristic of coming as a wiring ribbon which can help clean up wiring a fair bit. Not necessary for future projects if different wiring is available.

5. Fine Tip Curved Tweezers

Cost: $3.95

Distributor:

Adafruit

Major Specifications:

120mm long, 9mm gap at tips while open

Documentation/Website:

None. Specifications are provided on Adafruit's purchasing page.

Purchasing Link:

https://www.adafruit.com/products/422

Notes:

Was necessary when positioning components to be soldered and de-soldered. Not critical for future projects but close to it. Soldering was less difficult when using these.

6. 3.5mm to 5.5mm DC Jack Adapter

Cost: $0.95

Distributor:

Adafruit

Major Specifications:

3.5mm or 1.3mm to 2.1mm DC adapter cable, 6.5" long

Documentation/Website:

None. Specifications are provided on Adafruit's purchasing page.

Purchasing Link:

https://www.adafruit.com/products/2788

Notes:

I used the adapter as intended. This can be spliced directly into the solar panel but that kills the versatility of having multiple solar panel output jack sizes. Splicing would also remove the ability to use different components with other jack sizes in the future. I avoided this. A cheap fix to a simple problem. Needed for the project.

7. USB A to B Cable

Cost: $3.95

Distributor:

Adafruit

Major Specifications:

3ft long

Documentation/Website:

None. Specifications are provided on Adafruit's purchasing page.

Purchasing Link:

https://www.adafruit.com/products/62

Notes:

Connects the Arduino to a computer over USB for programming. Needed as the Arduino didn't come with a cable.

References

[1] Perlman, USGS Howard. "Water Science Photos: Measuring Water." Water Pictures: Measuring Water - USGS. United State Geological Survey, 02 Dec. 2016. Web. 09 Apr. 2017.

[2] Matheny, Keith. "Study: Michigan Rivers among Worst for Organic Pollutants." *Detroit Free Press*. Detroit Free Press, 31 July 2016. Web. 09 Apr. 2017.

[3] "PH of Water." Environmental Measurement Systems. Fondriest Environmental, n.d. Web. 09 Apr. 2017. <http://www.fondriest.com/environmental-measurements/parameters/water-quality/ph/#p6>.

[4] Wagner, Richard J., Robert W. Boulger, Jr., Carolyn J. Oblinger, and Brett A. Smith. "Guidelines and Standard Procedures for Continuous Water-Quality Monitors: Station Operation, Record Computation, and Data Reporting." United State Geological Survey, 10 Apr. 2006. Web. 9 Apr. 2017.

[5] Perlman, USGS Howard. "How the U.S. Geological Survey Monitors Water." How the U.S. Geological Survey Monitors Water. United States Geological Survey, 15 Dec. 2016. Web. 10 Apr. 2017. <https://water.usgs.gov/edu/watermonitoring.html>.

[6] Barton, Blain. "IoT PH Flow Controls and Automation with Costa Farms." Microsoft Technical Case Studies. Microsoft, 03 Nov. 2016. Web. 9 Apr. 2017.

[7] Sensorex SAM-1 Smart Aqua Meter. https://www.sensorex.com/sam-1-smart-aqua-meter/

[8] Floating Sensor Network. University of California - Berkeley, n.d. Web. 10 Apr. 2017. <http://float.berkeley.edu/>.

[9] Wikipedia – GSM. https://en.wikipedia.org/wiki/GSM

[10] MQ Telemetry Transport. http://mqtt.org/faq

[11] Adafruit MQTT Library. https://github.com/adafruit/Adafruit_MQTT_Library

[12] Adafruit FONA Library. https://github.com/adafruit/Adafruit_FONA

[13] Adafruit Sleepy Dog Library. https://github.com/adafruit/Adafruit_SleepyDog

[14] Wikipedia – Watchdog Timer. https://en.wikipedia.org/wiki/Watchdog_timer

[15] Matplotlib website. http://matplotlib.org/

[16] Basemap website. http://matplotlib.org/basemap/

[17] Pandas website. http://pandas.pydata.org/

[18] TkInter wiki page. http://tkinter.unpythonic.net/wiki/

[19] TPS63060 Documentation. http://www.ti.com/product/tps63060

[20] XBee Hardware. https://www.digi.com/lp/xbee/hardware

[21] Adafruit IoT Website. io.adafruit.com

[22] Sparkfun IoT Website. https://data.sparkfun.com/

[23] "Biofuels Basics." Biofuels Basics | NREL. National Renewable Energy Laboratory, n.d. Web. 10 Apr. 2017.

[24] Installing Additional Arduino Libraries. https://www.arduino.cc/en/Guide/Libraries

[25] Anaconda Package List. https://docs.continuum.io/anaconda/pkg-docs

[26] Gohlke, Christoph. "Unofficial Windows Binaries for Python Extension Packages." Python Extension Packages for Windows. N.p., n.d. Web. 11 Apr. 2017. <http://www.lfd.uci.edu/~gohlke/pythonlibs/>.