UCSC Capstone Design Project

– Intuitive Auto-Irrigation –

In Collaboration with UC Santa Cruz's Kresge Co-Op Garden

Sam Aiken & Brian Naranjo & Grant Skidmore & Henry Tuckfield
(saaiken@ucsc.edu) (brnaranj@ucsc.edu) (gskidmor@ucsc.edu) (htuckfie@ucsc.edu)

October 2019 - June 2020

Abstract

Acknowledging residential irrigation as a significant contributor to unnecessary wastewater, the Intuitive Auto-Irrigation team devised a water-efficient automatic irrigation system that uses an array of sensors and weather forecasting data to determine when to efficiently water plants and minimize water consumption. Through wireless communication, the sensor arrays relay information to a central hub that collects the data and triggers water delivery when the soil is deemed dry and conditions are optimal for irrigation. The goal of the automatic irrigation system is to reduce unnecessary water consumption without sacrificing plant health, effectively leading to a more sustainable method of residential landscaping and gardening practices.

Contents

1 Introduction to Sensor-based Automatic Irrigation

1.1 Solving the Irrigation Wastewater Problem

Wastewater from irrigation is a significant problem threatening municipal water reserves and natural conservancy efforts alike. According to the US Geological Survey (USGS), the Western United States' has a high per capita water consumption, which is heavily impacted by residential landscape irrigation [21]. Also, according to a later study performed by the Environmental Protection Agency (EPA), "Outdoor water use accounts for 30 percent of household use, yet can be much higher in drier parts of the country and in more water-intensive landscapes" [2]. Later in this study, the EPA states, "The arid West has some of the highest per capita residential water use because of landscape irrigation" [2]. This problem worsens due to the excess wastewater produced from timer-based automatic irrigation systems. The days of irrigation systems filling gutters with unused fresh water needs to come to an end; it is about time residential irrigation systems get updated to the 21st century.

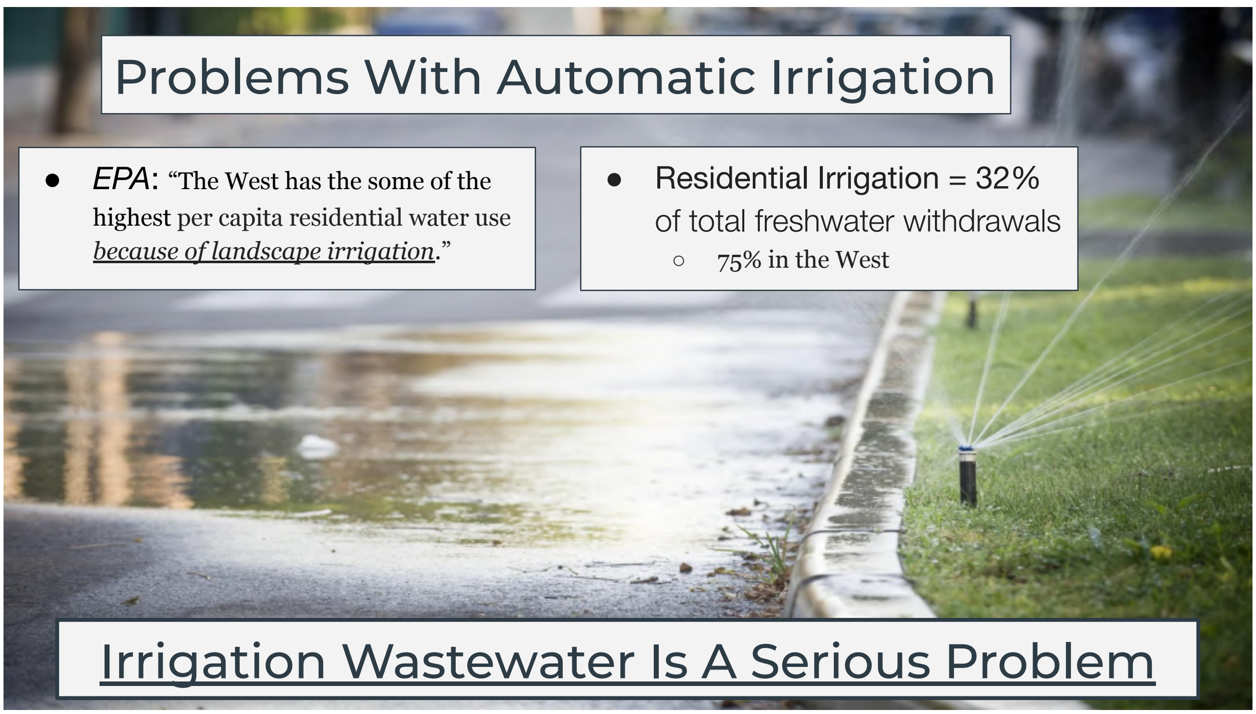


Figure 1: Outlining the problem caused by mismanaged residential irrigation, the EPA and USGS provide useful data that support the design of our improved automatic irrigation system. [1]

Presently, the only available options for auto-irrigation systems are timer-based, meaning they operate based on clock cycles and turn on whenever that cycle elapses, regardless of the conditions outside. A simple search on Amazon reveals that most of the systems use a central hub that costs anywhere from $80 to $150; these hubs simply connect to a hose and open the valves to each hose when specified by the timers. Not only are these options inefficient at water management, they are surprisingly expensive for just a simple central hub, without many (if any) high-tech features like wireless compatibility or a configurable user interface. And on top of that, unless you are paying at

[1] Figure courtesy of City of Santa Barbara. [17]

the very end of that $150 price range, these systems only work on a single hose. So if you wanted to have multiple system lines for different plants, you would need to buy even more of these system hubs.

Researchers previously noted the water-saving benefits of utilizing sensors to control crop irrigation. Engineers at the Centro de Investigaciones Biológicas del Noroeste designed a system comprised of a wireless sensor network and a general packet radio service (GPRS) module to control crop irrigation [7]. Their design used commercial moisture and temperature sensors located at the site of irrigation to detect soil parameters and determine water necessity. The system was set in a sage crop field that was watered for approximately 140 days; the data showed water reductions of up to 90% compared to traditional irrigation practices . Similar designs, including the one made by researchers at the U.S. department of agriculture, have also proven the efficacy of real-time soil-moisture monitoring and site-specific watering for large-scale farm applications [8]. It is clear that these wireless sensor networks, which precisely control crop irrigation, can reduce water waste, however buying one is not always an option considering that these systems routinely cost in excess of $2000. There have not been any attempts to provide affordable versions of these products to household consumers and smaller-scale gardens, so our team looks to solve this issue by creating a cost-efficient alternative to automatic irrigation systems that still provide the wireless and sensor-based functionalities that these more expensive models offer.

Introducing Intuitive Auto-Irrigation, an intelligent way to go about everyday irrigation benefitting both water reserves and plant health. The system takes advantage of real-time sensor data and forecast predictions to only trigger water delivery when necessary, actively reducing the quantity of wastewater produced through automatic irrigation. All while offering it at a cost that is kind to the consumer.

1.2 Stakeholder Goals - Kresge Co-Operative Garden

After looking at many communities that could potentially benefit from the Intuitive Auto-Irrigation project, we concluded that stakeholders benefitting the most from our system would be local gardeners, since they tend to find difficulty in managing the precise details required to optimally water their plants. With an automatic system in place to handle this operation, gardeners can spend the extra time attending to other tasks, increasing their everyday efficiency and available time. Varying sizing of lawns or gardens implicates the need for a system that could be easily modified for a wide array of irrigation usages. Therefore, it is our team's priority to design a modular system that focuses on small-scale versions of the complex commercial automatic irrigation systems that also has the capability of scaling-up to larger gardening or farming operations.

To demonstrate the effective use and benefits our automatic irrigation system brings to both casual and more experienced gardeners alike, we sought the opinions of local gardening groups to obtain useful feedback and to optimize our system around their needs. With regards to contacting gardening enthusiasts on campus, we were lucky enough to come across members of the Kresge Co-Operative Garden, who tend the most extensive student-run garden at the University of California, Santa Cruz (UCSC) campus. The members of the Kresge Garden Co-Op were more than happy to tell us about their gardening strategies and operations. They were willing to work with us to create a system that not only benefits them, but the other gardeners and farmers that come after them.

Introducing us into the Kresge garden, the Co-Op members discussed specifics about the garden, informing us about the types of plants and trees that they grow, the quality of their soil, the typical periods of growth, and other critical aspects of managing a garden. The Kresge garden grows a multitude of different plants and vegetables ranging from fruit trees to radishes and green onions. Their reputation of growing the largest student run garden on campus has a lot to due with how they sustain the quality of their soil. To accomplish this, they evenly distribute flax seeds, legumes and soybeans within their naturally soft soil to provide necessary nutrients and allow for a layer of pre-crop to grow. This pre-crop provides a healthy base that the plants can grow in while preventing the need for artificial fertilizers. In order to ensure proper growth occurs, it is imperative that water is sufficiently supplied to the plants especially during periods of extremely dry weather. The garden members have previously found that this dry period mostly encompasses the months of April to August but it does happen to vary substantially and subsequently has to be monitored to ensure proper garden care.

An important feature we specifically asked about was their watering strategy. Their current system was a central spigot system, having four separate hoses that each distribute water to a network of soaker hoses. Therefore, the process of watering their plants revolves around manually turning on the desired hose and waiting until the soil is sufficiently wet. Although this task is not considered challenging by traditional means, the challenge comes with determining when the water needs to be turned on and off. Automating this process would remove the human error of having to estimate the soil moisture content, drastically reducing the amount of water used for irrigation while simultaneously protecting plants from over-watering. Subsequently, the Intuitive Auto-Irrigation system would reduce irrigation costs and give the gardeners one less thing to worry about.

In terms of design considerations, we asked the members initial questions regarding practical features that would be helpful within an ideal auto-irrigation system. The Kresge Co-Op gardeners determined they would allow our system to monitor and control the irrigation of their plants if and only if there would be no risk in water leakage from the central spigot. This indicated the need for flow meters to monitor the water consumption at the central hub. They also found that if our system is sensor-based, it would be handy to be able to access or monitor the state of the plants from the central hub. This drove the design of a user interface used to control the system from a central location. As for suitable sensors, they seemed most interested in the soil moisture, light, and temperature sensors, confirming the first two as critical factors for optimized irrigation.

One downside to the location of the Kresge Garden is the lack of 120V AC wall power within a 500-foot radius from the central spigot. This adds a major constraint onto our design and final implementation in the Kresge Garden. When discussing this issue with the Co-Op members, they understood our dilemma and we reached a point of consensus. They agreed with us that due to the difficulty of controlling the water distribution with no wall outlet and to avoid water leakages on campus property, we would instead focus on the goal of monitoring the soil of their crops and relaying useful information back to them in a way that is visually clear and descriptive. Therefore, in order to tend to these goals, we will be working to log the data for each of these sensors and store the information both within a database as well as on a local SD card in an easily manipulable data format such as .csv or .txt.

1.3 Criteria of a Successful Project

The Intuitive Auto-Irrigation project was built around the fundamental design considerations put forth by the caretakers of the Kresge Co-op Garden and the garden's inherent limitations. Therefore, as a team we compiled a set of specifications upon which the project would be based. These specifications range widely from quantitative radio frequency (RF) considerations to qualitative plant health objectives.

The primary objective of the Intuitive Auto-Irrigation project, from an environmental viewpoint, is to reduce the amount of wastewater from irrigation. The goal is to use less water than timer-based automatic irrigation systems; if we can prove that a prototype sensor-based automatic irrigation system can outperform a timer-based system, then we know there is sufficient justification to warrant further product development. With that in mind, we plan to compare water consumption on a monthly basis using a flow meter to record flow rate. Success for this section of the project would have a lower water consumption, by any margin, than an automatic system. Table 1 outlines the criteria we wish to meet in order to validate the effective water-saving capabilities of the system.

Table 1: Water Delivery Success Criteria

Criteria	Indicator	Client Goal	Measurement Strategy
Lower water consumption vs standard automatic irrigation systems	Monthly Water Consumption is lower than manual or timer based alternatives	Consume equivalent or lower amounts of water for irrigation during testing.	Use flow meters to track water consumption of the water delivery and compare to projected water usage from daily irrigation.
Accurate monitoring of system's water consumption	Flow meters can record the water flow through the system with minimal error.	Achieve maximum of 5% error within water measurements after flow sensor calibration.	Conduct trials where set amounts of water are passed through the flow meted and compare the resulting water measurement to the actual amount of water.
Electrical control of multiple sources of water delivery for plant irrigation	Control of latching solenoid valves to distribute water delivery for irrigation.	Individual control of three latching solenoid valves for seperate water delivery actuation.	Test actuation of seperate latching valves and check if irrigation can be initiated and stopped.

The next set of considerations, shown in Table 2, outlines plant health guidelines from a qualitative perspective. Since the goal of the Intuitive Auto-Irrigation project is grounded on the notion of reducing excess irrigation wastewater, simply letting the plants die off is not an option either. This criteria for success is geared at verifying that plants subjected to Intuitive Auto-Irrigation control are at least as healthy as their timer-based counterparts. This verification is best performed through objective surveys of visual plant health. It is time-consuming and expensive to perform quantitative analyses on plant health when the option exists to survey qualified random samples, questioning them on their objective opinion of the in-questioned plant's health.

Table 2: Plant Health Success Criteria

Criteria	Indicator	Client Goal	Measurement Strategy
Retain or improve plant health	Qualitative features of plant. (Color, leaf texture, relative growth)	Prevent death of plants under testing through irrigation and obtain an average rating of 6 or better from surveys.	Survey users to subjectively grade plant health. Provide users images of two different plants: one manually watered and one equipped with the IAI system, and have them rate each plant from 1-10 .

In order to implement a successful sensor-based irrigation system, there needs to be a way for the sensors to talk with the system's water delivery control. Instead of stringing together long wires from sensor to a central location, it is more practical and efficient to install wireless communication. Therefore, regardless of where the sensors are located, assuming they are within range, they will be able to communicate with the central control hub. With this protocol in mind, there are a couple of factors that are important to a working system, namely the range of the wireless communication and its reliability. The range is pretty self-explanatory; a longer-range system is preferred in order to maximize the potential sensor coverage. The reliability, however, is measured by the number of messages lost. For this context, messages are the medium through which data is shared over wireless channels. These two measurements, range and reliability, are inversely proportional because as the range increases, the number of messages lost (and therefore reliability) decreases. Increasing this range while still providing reliable communication is desired for maximizing the efficiency of the wireless communication system. Table 3 describes the criteria which will be met as well as the method that will be used to validate the system's long-range and reliable wireless communication.

Table 3: Wireless Communication Success Criteria

Criteria	Indicator	Client Goal	Measurement Strategy
Long-range transmission of sensor data	Messages can be sent between sensor nodes and central hub over given distance	Successful wireless communication over 400 feet distance between sensor nodes and central hub	Measure distance between sensor nodes and central hub and test that 5 consecutive messages can be sent and received.
Reliable wireless communication	Sensor data packets are not lost when within specified range of central hub.	Zero data packets lost during testing if the sensor nodes are within the specified range.	Check time stamps during testing and ensure that there are no cases where sensor data packets are not received by the central hub.

After figuring out a baseline for our project in terms of its environmental impact (from water consumption) without sacrificing the performance of automatic irrigation systems (plant health) and how we would actually implement this design (RF), we looked toward the user for our final success criteria. User's need a way to customize the system's configuration through a simple user interface (UI). This provides the capability of mapping each sensor to a specific hose to set up automatic irrigation. The main purpose of the UI is therefore to provide users the ability to configure system settings as well as interact with information acquired from the sensor nodes.

Providing these capabilities to the user in a easy and presentable manner would deem this subsystem successful.

1.4 Introduction to the System Design

The Intuitive Auto-Irrigation system uses C++ programming to implement control of an ad-hoc one-hop network comprised of a single master node (ie. Central Hub) and an army of sensor nodes to monitor real-time conditions and determine when water is needed. The purpose of this system is to control *when* to water; instead of watering everyday at a specific time, our intuitive auto-irrigation system determines when to water based on sensor data from each of the sensor nodes. Individual sensor nodes numbered from 1 to n connect to the central hub microcontroller, where $n < 126$. They then record data from the sensors and transmit that data to the central hub. The central hub records the sensor data onto an SD card, uploads it to a database, and drives the latching solenoid valves, Organic-LED (OLED) UI display and flow meters. When it is determined (based on sensor data) that water should be delivered, the central hub triggers a latching solenoid valve, which opens to allow water to flow to all the plants connected on that hose. Encapsulating the full, high-level system design, the intuitive auto-irrigation system is shown in the block diagram, Figure 2. The system block diagram outlines how sub-components interact and communicate to form the full system. This is a high-level overview of the system from a technical standpoint; for a schematic of all electronic components, refer to Section 6.1.

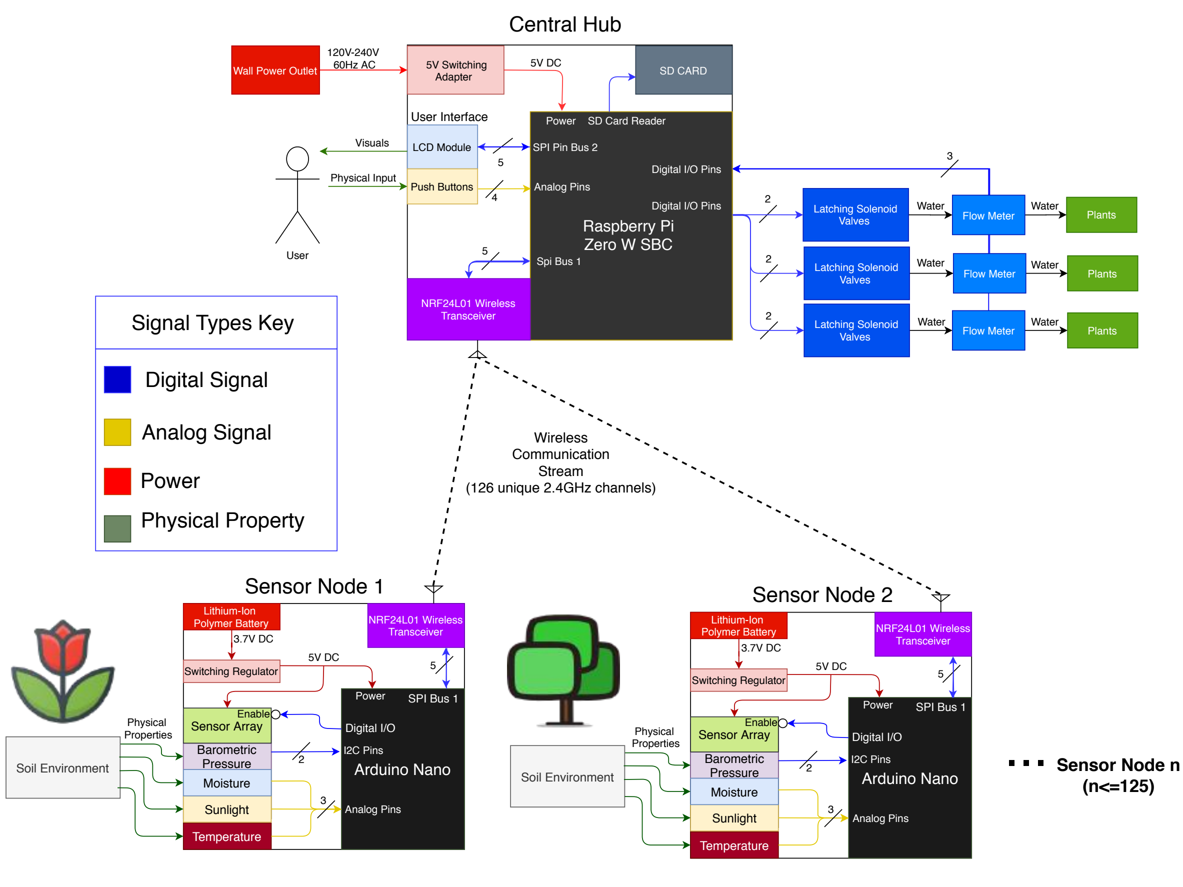


Figure 2: System Block Diagram. [2]

The wireless subsystem is designed around low-power radio-frequency (RF) wireless radio transmitters. These transmitters, the nRF24L01+ modules (covered in detail in Section 2.2), allow for long-range wireless communication between sensor nodes and a central control hub to determine if water delivery needs to be triggered or not. The RF radio transceivers on the sensor nodes are controlled via Arduino Nano microcontrollers powered by rechargeable lithium-polymer batteries. These microcontrollers allow the RF modules to interface with a low-power microprocessor, the ATMega328P in order to both read sensor data and report back to the central control hub regarding location-based water requirements. Since the sensor nodes should be able to transmit data about a plant that is geographically far from the central hub, the sensor nodes need to be able to communicate wirelessly. For that reason, we implemented a one-hop network to relay information between nodes, see Figure 3 for a visualization of the sensor node network implementation.

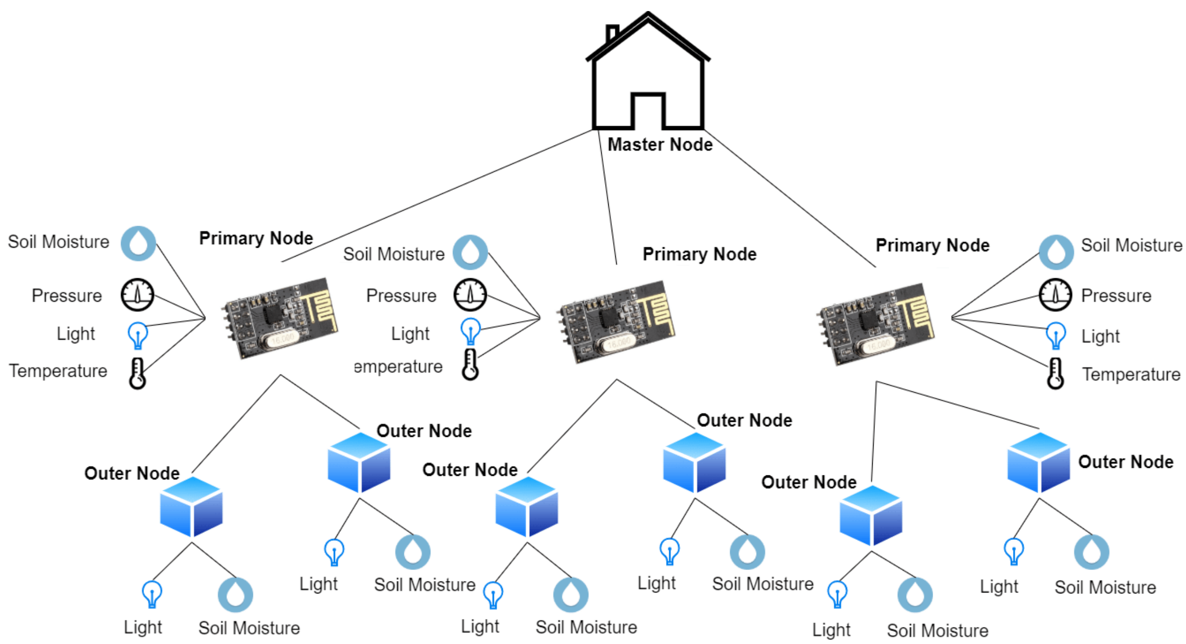


Figure 3: Network Diagram depicting the one-hop network, complete with equipped sensors. [3]

Each Arduino-RF module pair constitutes a sensor node; however, no sensor node is complete without the actual sensors themselves. Sensor nodes come armed with an array of sensors to actively monitor soil moisture content and light level while some also include temperature and and barometric pressure sensors. For specifics regarding the sensor array, refer to Section 2.6. The data from these sensors allow the Intuitive Auto-Irrigation system to control water delivery at a level that humans could never do. Of course, there needs to be some amount of customization in order to configure according to the user's design; user customization is the reason the system includes a user interface (see Section 2.8).

Each sensor node operates on a timer, where most of its time is spent sleeping to conserve battery life. Periodically the sensor nodes awaken on a watchdog interrupt to record sensor data and transmit information to the master node. The master node keeps track of all the sensor information and triggers up to four separate hoses if the configuration networks indicate water is required. While

[2]This figure was made with Draw.io. [4]

[3]This figure was made with Draw.io. [4]

this method appears somewhat like a black box, the algorithm used to determine when to trigger water delivery acts primarily off the moisture sensor while using the other sensors to further optimize efficiency (see Section 2.2.3 for a detailed analysis of the watering algorithms).

On the central hub, the RF transceivers instead interface with a commercial Raspberry Pi Zero W SBC (single board computer) powered via $120V_{AC}$ wall power that is converted to a usable 5V rail. This SBC has many useful features that allow for additional system functionality; more on the Raspberry Pi can be found in Section 2.1.

consider the project

2 Technical Discussion of the Automatic Irrigation System Design

2.1 Microcontrollers - Raspberry Pi and Arduino Nano

As a way to initiate wireless communication and interface with sensors, microcontrollers are used at the central location where the main system is found as well as remote areas where the sensors will be placed. Microcontrollers allow us to write software that can autonomously control the irrigation system. However, due to differing design constraints, the system needs two different types of microcontrollers - one that controls the central hub and another used on the sensor nodes. The parameters that were taken into consideration when choosing microcontrollers for the central hub and sensor nodes are discussed throughout this subsection.

In the process of selecting a microcontroller for the sensor node, we took into account the expected functions of this subsystem. The sensor node's role is to:

1. Periodically wake on an interrupt-driven timer
2. Read the sensors
3. Analyze the data to see if water is needed
4. Send the raw data and data analysis to the central hub
5. Then go back to sleep

Since the sensor nodes are operated using battery power to allow for remote application, it is critical that they are only operated for short periods of time so that battery life is conserved. For these microcontrollers, we are willing to sacrifice high core clock speeds (which allows for fast computation) and more extensive memory storage in exchange for lower power consumption. Typically, higher core clock speeds and more memory are preferred, but here we have other priorities in our design that contrast with those properties. Cost is also an important factor since we ideally, we want many sensors, so the cost increases quickly with expensive models.

The last important parameter when choosing a sensor node microcontroller is the size. The microcontrollers need to be relatively small to minimize the space they consume in gardens. We used a Pugh Chart to help with the decision-making process by laying out all of our specifications in columns and assigning weights to the specifications based on their importance in the design. We then assigned each of the possible component choices (in the rows) different values based on their specs and tallied up all the columns to determine the best choice for our microcontroller. Figure 4 shows this process of using weighted considerations to conclude which device was selected.

We followed a similar process for determining the central hub microcontroller. The master hub's priorities differ: the specifications stay the same, but the weights for the individual specifications are different. For the master hub, the most important considerations are the amount of available memory, additional features, and price. To reflect these priorities, the weightings in the Pugh Chart shown in Figure 5 change accordingly. We prioritize more features on the master node microcontroller, like built-in WiFi and SD card memory, to simplify the central hub design. Additional built-in features make the central hub design less complicated; with many of the standard features

Feature	Cost	Size	Addressibility	Memory	Features	Power Consumption	Computing Power	Total
Weighting	2x	2x	1x	1x	1x	3x	1x	
PIC/Uno32	- -	-	++	++	0	0	+	-1
Arduino Nano	++	+	- -	- -	0	++	-	7
Arduino Uno	+	0	-	- -	0	+	0	1
Arduino Mega	-	0	+	+	0	+	0	1
Raspberry Pi Zero W	0	+	+	+	++	-	+	4
Raspberry Pi 3	- -	0	+	++	++	-	++	3

Figure 4: Sensor Node Microcontroller Pugh Chart showing the criteria resulting in selection of the Arduino Nano.

already included, we can spend more time on the actual design of the auto-irrigation system. The master hub also requires more memory than the sensor nodes because of larger program file sizes and the need to write sensor data to system memory.

Furthermore, the price is significant as well. While we are only buying a single central hub, the more powerful microcontrollers fit for the central hub are significantly more expensive than the microcontrollers used for the sensor nodes. Therefore, while the justification for cost consideration is different from the sensor nodes to the central hub, the result for the weighting stays the same. Therefore, while the justification for cost consideration is different from the sensor nodes to the master hub, the result for the weighting stays the same. The result of our weighted decision-making process is shown below in Figure 5:

Feature	Cost	Size	Addressibility	Memory	Features	Power Consumption	Computing Power	Total
Weighting	2x	1x	1x	2x	2x	1x	1x	
PIC/Uno32	- -	-	++	++	0	0	+	2
Arduino Nano	++	+	- -	- -	0	++	-	0
Arduino Uno	+	0	-	- -	0	+	0	-2
Arduino Mega	-	0	+	+	0	+	0	2
Raspberry Pi Zero W	0	+	+	+	++	-	+	8
Raspberry Pi 3	- -	0	+	++	++	-	++	6

Figure 5: Central Hub Microcontroller Pugh Chart showing the criteria resulting in selection of the Raspberry Pi Zero W.

As shown in these two Pugh Charts, we settled on the Arduino Nano microcontroller for the sensor nodes and the Raspberry Pi Zero W single-board computer (SBC) for the master node. For the sensor nodes, this means we implemented a low-power, small microcontroller to read sensors, perform calculations, and operate the RF module. Moreover, regarding the central hub, we get access to many features, like data logging with the SD card, on-board WiFi to get API forecast data, and more memory to sustain larger program files. The rest of the project utilizes these microcontrollers involving the design discussed throughout the rest of this section.

2.2 Wireless Communication Using nRF24L01 Wireless Transceivers

One of the key facets of the Intuitive Auto-Irrigation system is its reliance on wireless communication to transmit information between sensor nodes and the master node. For our purposes, there are two different options for wireless communication, unipoint and multipoint protocols. Unipoint communication is the simplest form of wireless communication where nodes connect directly to talk to each other. It is very reliable because it limits each node to a single state, only the receiving (RX) or transmitting (TX) states, at any given time. Unfortunately, this protocol is not a powerful or efficient method of wireless communication, especially for larger systems, because of the redundancy of having to manually switch between the TX and RX states.

On the other hand, multipoint protocols create a network of interconnected nodes and allow any node to communicate with any other node in the network. These protocols do not limit nodes to a TX or RX mode, rather they default to RX and maintain the ability to transmit while in said RX mode. For small networks, this has little impact on transferring information; for networks of only two nodes, the change from unipoint to multipoint has almost no impact at all. But as the number of network nodes increases, the transfer of information becomes exponentially more complicated, which is especially true for networks with nodes that are spread far apart. Instead of having to hard code all the necessary network addresses beforehand with unipoint communication, multipoint communication automatically redirects messages through necessary intermediate nodes in order to deliver each message to its intended recipient.

This project employs nRF24L01+ wireless radio transceiver modules (see Figure 6), which use the unipoint RF24 Arduino library made by TMRh20 [24]. However, our project does not directly utilize functions from the RF24 library. Instead, the unipoint RF24 library simply serves as the underlying framework for multipoint communication libraries to build off of. In other words, these other libraries use the RF24 library to efficiently create a better network of sensor nodes. Even though our specific transceivers were originally designed for unipoint communication, we use a dynamically-assigned multipoint protocol to create a network of nodes that can communicate over long distances.

Encapsulating many different protocols, multipoint communication is an umbrella term which can be used broadly. The Intuitive Auto-Irrigation system implements an ad-hoc one-hop network, which is a direct network of nodes that dynamically connect to several other nodes within range. The software for much of this network control is implemented through the framework provided in the RF24 Network and RF24 Mesh libraries, originally authored by TMRh20 [25][26]. Our one-hop network consists of a central hub acting as a master node with several primary and subsidiary nodes in slave configurations, which are used to actively gather sensor information and report back to the master.

Essentially, multipoint protocols are a more sophisticated form of unipoint communication. Multipoint protocols revolve around many nodes inter-communicating, allowing for a more efficient

[4] Figure from Newegg.com [15].

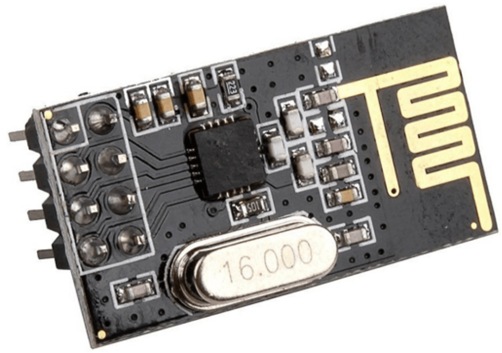


Figure 6: nRF24L01+ wireless radio transceiver module showing the small PCB, 16 MHz clock crystal and RF antenna. [4]

transfer of data. This means that instead of limiting communication to only a single other device (as with unipoint communication), we can easily talk to all the other nodes on the network. As stated above, our network follows a one-hop topology, which has a central node that controls the routing and relaying of information. See Figure 3 for a visual depiction of this ad-hoc network.

The difference between traditional one-hop networks and our system is the use of a hierarchical network, meaning nodes geographically closer to the master are assigned as parents while further nodes are assigned as children. The network achieves this type of protocol due to dynamic address assignment through the master node, which describes how nodes are automatically assigned network addresses as they connect to the network. Dynamic assignment allows the master to monitor all nodes on the network and assign new nodes addresses that correspond to their geographical location. The master keeps a record of all addresses, which also describe the parent-child relationships, to control how messages are routed through parent nodes to the child nodes that may be geographically out-of-range from the master.

For example, when the first node in the network connects to the master, it may be assigned an address of 4. Since there is only one node in the network besides the master, there is no complexity in the way that messages need to be routed. However, once a second node connects to the network, the routing protocol takes into effect. There are two scenarios here, either the node is in range of the master, or it is not in range of the master but is in range of the node assigned address 1. In the first scenario, the new node will be assigned a different address, a number 1-6 not including 4. The reason the address cannot be greater than six lies in the inherent limitation that any node can only listen to 6 other nodes at a time. Dynamic address assignment also helps here as it tells each node exactly which other nodes they should be listening to. Moving on to the second scenario, where the new node is far from the master but close to the other established node, this new node will be assigned the address 41. For the purpose of routing messages, the 4 indicates the parent node while the 1 indicates the first child of node with address 4. If another node were to be added close to the node with address 41, it would likely be assigned address 42. Therefore, the dynamic address assignment protocol can access any node on the network by simply routing through the parents to the children.

There are numerous benefits to our version of a dynamic ad-hoc one-hop network, each offering a unique aspect of multipoint communication:

- Automatic rerouting through parent-child nodes
- Automatic reconnection if disconnected
- Dynamically assigned addresses based on location

All of these available features are utilized in our system. First, automatic rerouting removes the process of figuring out where and how to send messages to other nodes trying to find the intended recipient, making the process many times faster than manual routing. Basically, instead of having to ping all available nodes for a response to check if they are in range, the transmitting node sends its message the master node. The master node stores the network topology and all parent-child relationships and routes the message accordingly. And in the reverse scenario where a specific node is not sending a message but receiving one, if that specific node is not the intended recipient of the message, it will re-route the message to its specified parent or child. This is another way how the one-hop network propagates messages through the network and allows nodes that are geographically far from each other to maintain communication.

The next benefit is automatic reconnection which is useful for battery-related issues. In the case where one of the sensor nodes dies and the battery must be recharged, as soon as the node is turned back on (assuming it is within range of the network), it will automatically reconnect. We can also force manually reconnection when a sensor node detects it has lost connection to the network. Assuming no outstanding circumstances, it only takes the node less than a second to reconnect to the network once it loses connection.

Dynamically assigned addresses, as discussed above, also make the process of routing messages easy. As long as each node has a unique nodeID identifier, they are assigned a unique network address as soon as they connect to the network based on their location from the master. After this occurs, all nodes in the network are quickly notified of the network update and are able to route messages to and from the new network node.

Wireless communication protocol aside, even though the processes for unipoint and multipoint communication are quite different, the hardware connections are identical. Each of the wireless transceivers uses serial peripheral interface (SPI) protocol to communicate with a microcontroller. This involves the use of Master Out-Slave In (MOSI), Master In-Slave Out (MISO), serial clock (SCK), chip enable (CE), and chip select (CSN) pins to synchronize and transmit signals between the transceiver and the microcontroller. The transceivers also run off 3.3V, which is important as the transceivers have no internal regulator, only a logic-level converter to change the 5V digital logic from the microcontroller pins to 3.3V logic for the transceiver chips. All together, there are 7 total connections to each transceiver (including the 5 mentioned plus Vcc and Gnd). Refer to Figure 7 and Figure 8 for the pinouts of the transceivers to both microcontrollers used in the system.

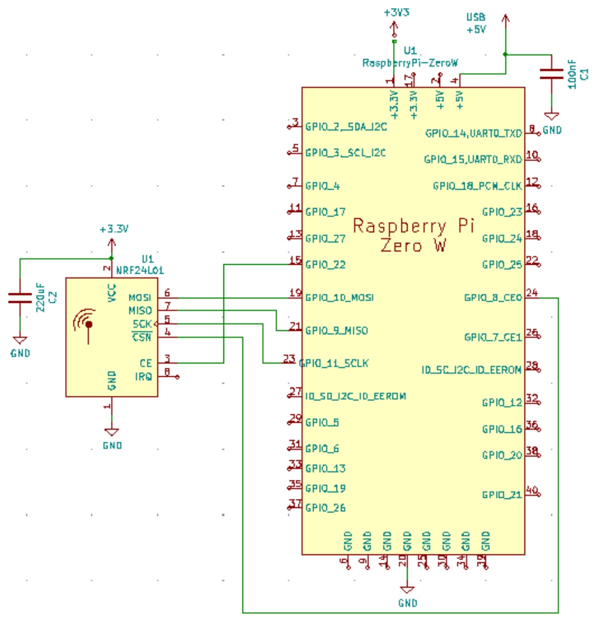


Figure 7: nRf24L01 to Raspberry Pi pinout showing how the wireless transceiver, U1, is interfaced to the central hub microcontroller.

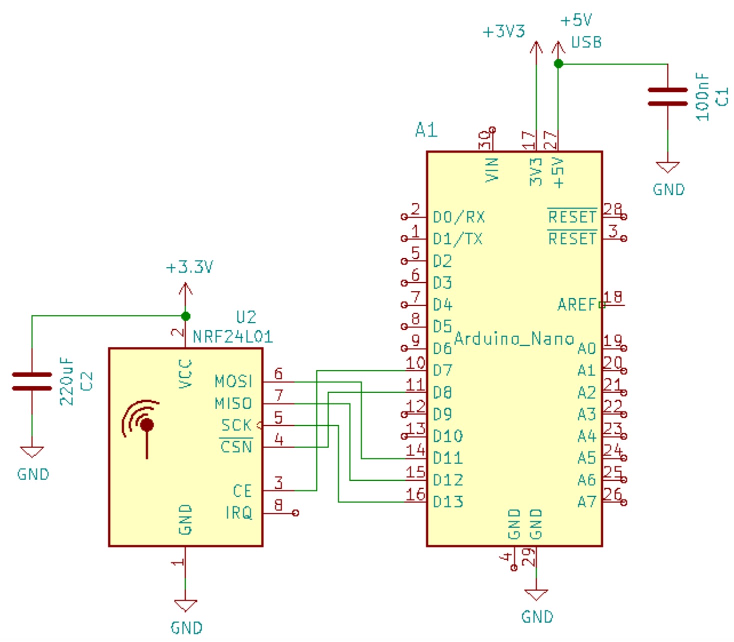


Figure 8: nRf24L01 to Arduino Nano pinout showing how the wireless transceiver, U2, is connected to the microcontroller within each sensor node.

---

[4]Figure was made with KiCAD. [14]

2.2.1 Multipoint Message Protocol

Through the framework provided in the three RF24 libraries (RF24, RF24 Network, RF24 Mesh), the system sends unique messages to and from other nodes to ultimately deliver sensor data to the master node. Each message gets an identifier to specify the type of message, allowing nodes to differentiate their responses to different types of data. The majority of the communication relies on a ping-out, pong-back protocol to conserve battery life, but the special message types allow for reconfiguration of the network through varied message responses. For reference, a ping-out, pong-back protocol involves two communicating nodes where one node sends an initial message and the recipient node sends a confirmation message back. Each of the different message types are listed below:

1. 'D' Message: 'D' messages are how the sensor nodes ping out to the master node. Each 'D' message contains a struct of all the sensor information as well as supplementary information relevant for data logging.

2. 'S' Message: 'S' messages are one way the master node pong's back to the sensors. An 'S' type message confirms reception of the sensor node's 'D' type message and tells the sensor node to go back to sleep.

3. 'C' Message: 'C' messages are alternatives to the 'S' type messages. A 'C' type message still confirms reception of the sensor node's 'D' type message and tells the sensor node to go back to sleep, but it also reconfigures the sensor node's threshold struct. In essence, it performs the same functionality as an 'S' type message but it also reconfigures the thresholds for the watering algorithm in Section 2.2.3.

Message types get handled differently as each transmission involves a separate data type requiring specific set-up. For example, the unique data types for the sensor data and data thresholds are shown in Figure 9. These structs are sent via wireless communication from node-to-node over a single message, reducing the total number of messages sent which minimizes network chatter. Each struct is designed to be smaller than the internal maximum frame size from the RF24 Network Library. This max frame size is defined as 256 bits, or 32 bytes, minus the size of the network header. The documentation tells us that the size of a network header is 10 bytes [26], which means if we want to send the structs as a single message, they need to be smaller than 22 bytes. Looking at the larger of our two structs, we have 2 uint8_t variables, 4 uint16_t variables, and 1 uint32_t variable, which gives us a total of 112 bits, or 14 bytes. Therefore, we easily have enough bandwidth to send each struct as a single message because the size of the largest message struct is 8 bytes smaller than the maximum.

The process of initializing a one-hop network follows the steps outlined in Figure 10. To first initialize a one-hop network, there needs to be exactly one master node, indicated by a node ID of 0. This node gets turned on to initialize the network. Then at least one sensor node, with a node ID between 1 and 125, must be turned on. Assuming the sensor node is in range of the master or another node on the network, it will automatically request the master node for a network address. All network addresses are assigned dynamically by node ID (DHCP Protocol; commonly used with IP networks), meaning they get uniquely assigned to each node as they connect to the network. Each node's lease time for the network addresses (the duration the assigned address is valid for) is indefinite as it gets stored in a .txt file; even if the central hub gets power cycled, the sensor node's network addresses remains static.

```c
// C_Struct stores relevant thresholds
typedef struct {
    float sM_thresh;
    float sM_thresh_00;
    float lL_thresh;
    uint16_t tC_thresh;
    uint16_t time_thresh;
} C_Struct;

// D_Struct stores the relevant sensor data
typedef struct {
    float soilMoisture;
    float lightLevel;
    uint16_t temp_C;
    uint8_t digitalOut;
    uint8_t nodeID;
    uint8_t battLevel;
} D_Struct;
```

Figure 9: Sensor data and threshold data types used for network communication.

Once the master processes the request for a network address and returns that address to the sensor node, all other nodes can send it a message by including the recipient's network address in the network header. Network headers simply contain information pertinent to the sending of the message, including the original sender's address and the intended recipient's address. If the intended recipient's address is unknown, any node can request an address from the master by providing a node ID. The master stores an array of all connected network nodes with their unique nodeIDs and network addresses. The lookup is straightforward from there, match the supplied nodeID with the network address and return that to the node that requested it.

Once the one-hop network is running and at least one sensor node is connected, the sensor nodes will send data to the master node based on interrupts from their own watchdog timers. These timers allow the sensor nodes to sleep, conserving battery, and wake up when the timer elapses. When the timer elapses and an interrupt service routine (ISR) is triggered, the node records all sensor information and maps them to usable integers, with sizes from 8 to 32 bits. The sensor node then runs an algorithm to determine if water is required or not. All of this data, including the algorithm output and the sensor node's node ID are then stored in the data struct (see Figure 9) and sent to the master. The master then stores all the data into an array and, using the most recent data from all the sensor nodes configured to that hose, decides whether or not to turn on the water. This process runs in parallel across all sensor nodes so that they all intermittently send sensor data to the master node over a user-specified time interval. This parallel protocol is how the wireless one-hop network records and utilizes sensor data to reduce irrigation water consumption. For the actual code used to run the full system, see Section 6.4 and Section 6.5 or refer to our repository on GitHub by using the following link:

https://github.com/GSkid/SkidLess

2.2.2 Multipoint Wireless Range Testing

While the multipoint wireless one-hop network runs consistently and smoothly, it is essential to test the range of the wireless transceivers. These transceivers have a limited range, and since they use

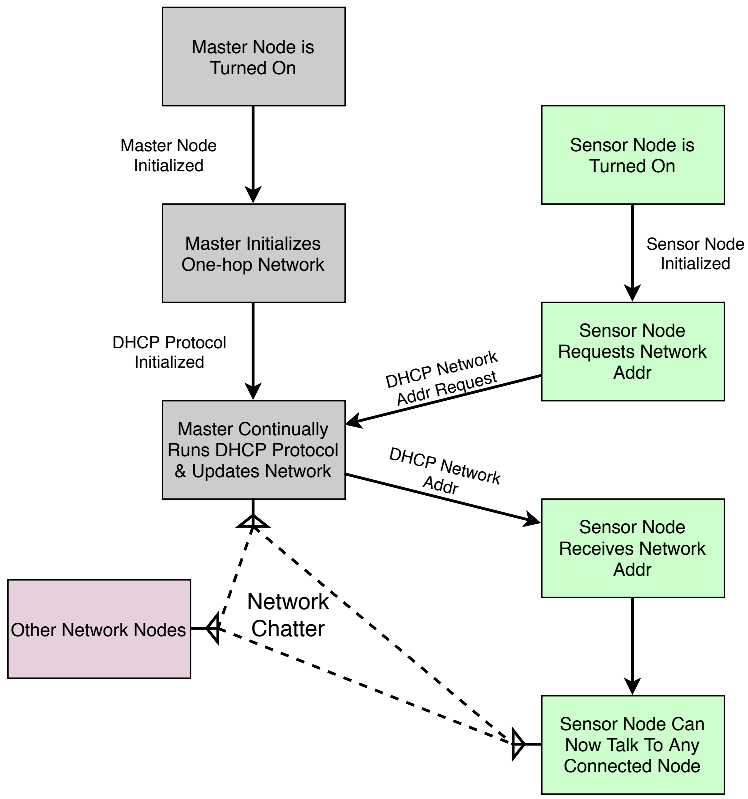


Figure 10: Network Initialization Flowchart shows the process for initializing the one-hop network and how a single node connects to the network. [5]

PCB-mounted antennas for power consumption, their range is certainly a limiting factor in their implementation. The Multipoint Wireless Range Testing Standard Operating Procedure (SOP) defines a set of steps for determining the transceivers' effectiveness at different ranges and heights from the ground. This section will look at a brief overview of the SOP itself as well as the results of our own testing as an example.

The first step in the SOP is to get a working master and sensor node on the same network. This is pretty simple as once the master and sensor nodes are turned on, they should automatically create and connect to the one-hop network respectively. Once reliable communication has been established between the two nodes, we then needed to find an open area to test the range in. As a side note, reliable communication is defined for our purposes as 5 consecutive, successful packet transmissions from sensor node to master or master to sensor node. Moving on, once we established a location with a large open area, we then moved the sensor node away from the master node. There are three distinctions we need to make here. First, it is important to maintain a direct line-of-sight throughout the testing procedure. Second is to try to keep the sensor node at the same height from the ground at all times; this is because the RF signals are affected by proximity to the ground. Lastly, both RF modules must be positioned so that their antennas are pointed at each other, see Figure 11 for reference.

As we moved away from the master node, the important thing was to verify reliable communication

[5]This figure was made with Draw.io. [4]

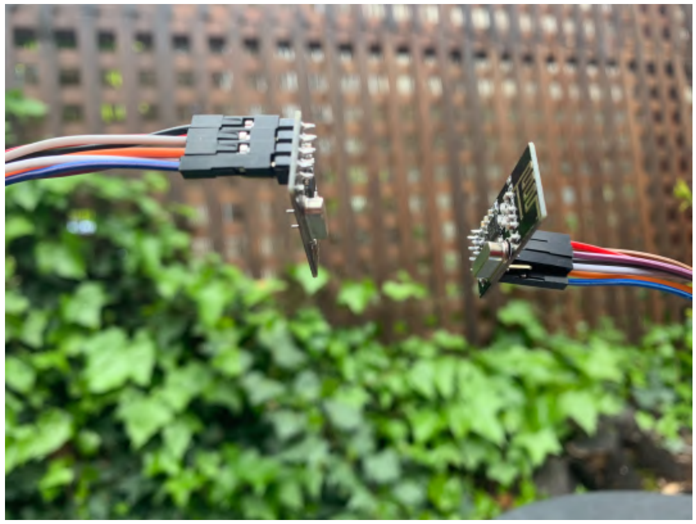


Figure 11: Antenna Orientation showing how to orient the antennas to ensure maximum signal strength.

at specified range intervals. For the range test, the range intervals are as follows:

$$100ft,\ 150ft,\ 200ft,\ 250ft,\ 350ft,\ 400ft,\ 425ft,\ 450ft,\ 475ft,\ 500ft.$$

At each of the range intervals, we stopped moving to verify reliable communication was established before moving on to the next distance interval. If we were unable to verify reliable communication as defined previously in this section, then the sensor node's current distance from the master node is considered the maximum range of that sensor node. For further in-depth testing protocol, see Section 6.2.

Our measured results when performing the test are shown in Table 4. In this table, the following are conventions for notation: 'HoF' stands for 'Height off Ground' and 'Distance' is the distance from master node to sensor node. All distances are measured in feet while all heights from the ground are measured in cm. The table is then broken down into sets of four based on the distance between master and sensor node and separated within those groups by the heights from the ground. The most important metrics in the table are the ones where the mater node and sensor node are 50 cm and 5 cm off the ground respectively because this is the scenario that the system will actually use. We can see from the results that the total maximum distance achieved was 450ft while the maximum distance for the ideal scenario is 425ft. These are solid ranges and meet our deliverable of 400ft.

One of the peculiar results we can get from here is that the 5-5cm testing, where both the master and sensor nodes are 5cm from the ground, is not very reliable at long distances. One of the main challenges with this method, especially in implementation, is direct line-of-sight. A direct line-of-sight is much more difficult to maintain when both sensor nodes are barely above the ground which is why it is imperative to test on a completely flat surface.

Distance B/t Nodes	Master HoF	Sensor HoF	Result
100	5	5	Works
100	50	5	Works
100	50	50	Works
100	125	125	Works
150	5	5	Works
150	50	5	Works
150	50	50	Works
150	125	125	Works
200	5	5	Works
200	50	5	Works
200	50	50	Works
200	125	125	Works
250	5	5	Works
250	50	5	Works
250	50	50	Works
250	125	125	Works
300	5	5	Fails
300	50	5	Works
300	50	50	Works
300	125	125	Works
350	5	5	Fails
350	50	5	Works
350	50	50	Works
350	125	125	Works
400	5	5	Fails
400	50	5	Works
400	50	50	Works
400	125	125	Works
425	5	5	Fails
425	50	5	Works
425	50	50	Works
425	125	125	Works
450	5	5	Fails
450	50	5	Fails
450	50	50	Works
450	125	125	Works
475	5	5	Fails
475	50	5	Fails
475	50	50	Fails
475	125	125	Fails
500	5	5	Fails
500	50	5	Fails
500	50	50	Fails
500	125	125	Fails

Table 4: Expected wireless range testing results at differing heights and distances.

2.2.3 Sensor Data-to-Watering Algorithm

As mentioned in previous sections, the sensor data-to-watering algorithm is how the sensor nodes convert raw sensor data into a usable signal for the master node. This sensor data is comprised of three central components: soil moisture, air temperature, and light level. The algorithm looks at each of the sensor values and based on static priorities combined with configurable thresholds, determines if water is needed or not. Refer to Figure 12 to look at the process for determining if water is required or not. This value then gets stored in the digitalOut value of the data struct (see Figure 9) and the whole struct is sent to the master. However, simply because a single node reports that it needs water does not mean that the corresponding hose will actually get turned on. For that, we use a different algorithm on the central hub.

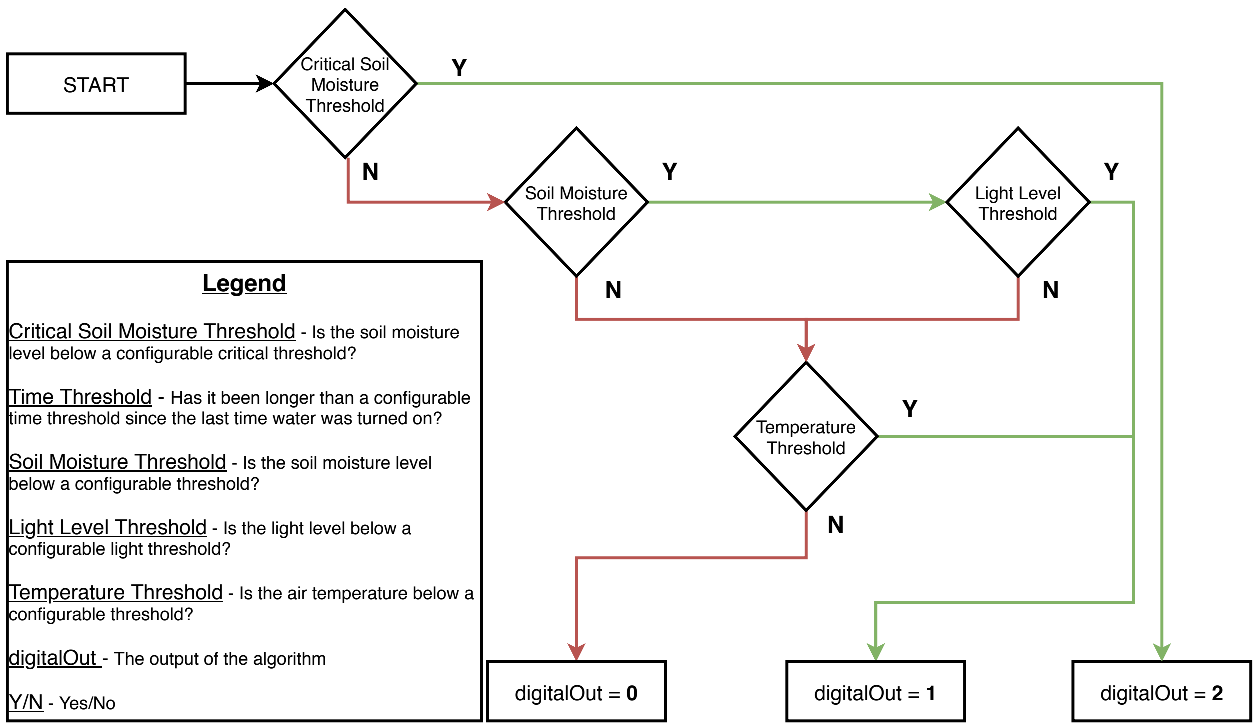


Figure 12: Sensor data analysis algorithm low chart depicting the process of determining if water is needed or not at the location of the sensor node. [6]

2.2.4 Central Hub Water Delivery Algorithm

Once the sensor data reaches the central hub, the central hub needs to transform the data into a signal that controls the latching solenoid valves for water delivery. This process is done by looking at each hose individually and tallying up the digitalOut signals from the sensor data-to-watering algorithm (see Section 2.2.3) and evaluating against a dynamic threshold. The flow chart that outlines this process is shown in Figure 13. This algorithm is called every minute to evaluate the changes in the sensor data and to potentially change the output to the latching solenoid valves (see Section 2.7 for more information on the latching solenoid valves).

[6]This figure was made with Draw.io. [4]

The first task in the water delivery algorithm is to tally up all the digitalOut signals from all the sensor nodes that are mapped to the pre-specified hose. For this entire process, it is important to note that each hose gets evaluated individually so this flow chart process is carried out once per hose per minute minute. In order to tally up the digitalOuts, the program retrieves the most recent data from an array where all new sensor data is stored and the array where sensor nodes are mapped to their respective hoses. If the nodeID from the data matches a nodeID in the hose mapping, then that digitalOut value in the data gets added to the tally.

Once the tally adds up data from all the mapped nodes, the total is compared against a dynamic threshold. This threshold is determined by:

$$\frac{\#\ of\ mapped\ nodes - 1}{2}.$$

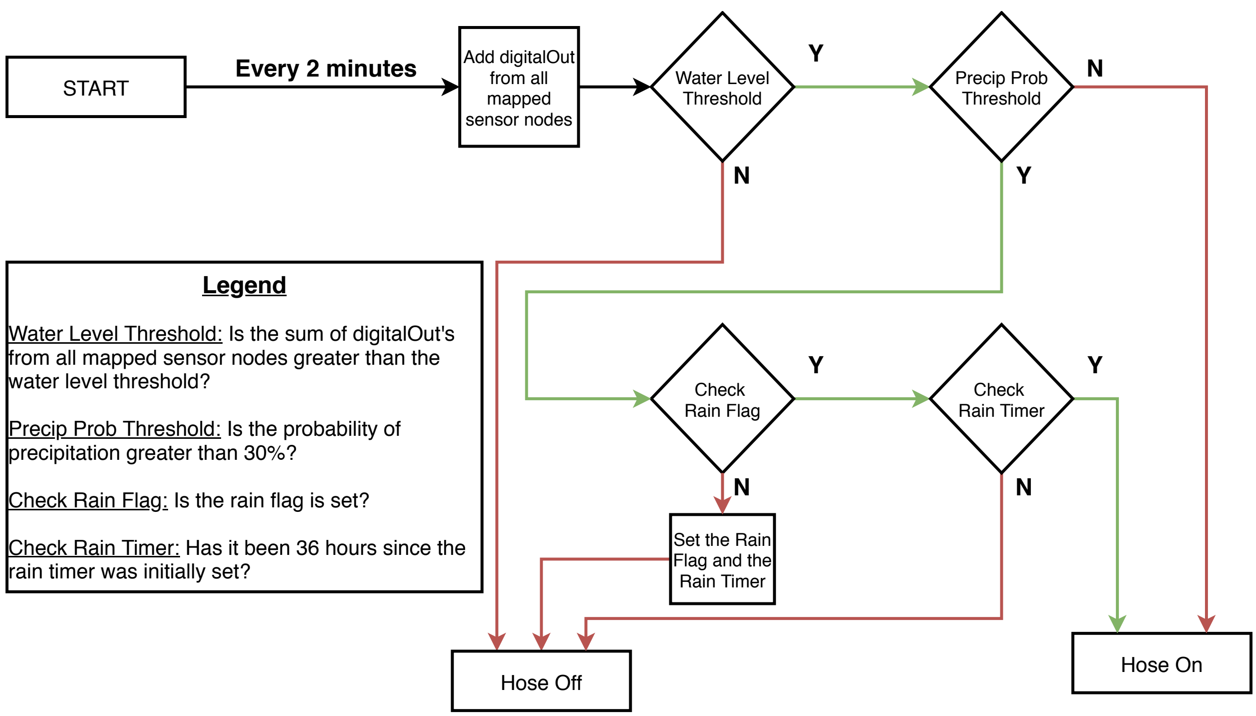


Figure 13: Flow chart depicting the process of converting the sensor data information into a hose output signal.

This formula is selectively chosen to find situations just below a simple majority for the total number of nodes reporting they need water. This process also requires the tally to be greater than zero in order to succeed, so having one or two nodes will still require one of them to report a high digitalOut in order to trigger water delivery. If the digitalOut tally is below the water level threshold, then the hose remains off. But if the tally is greater than the water level threshold, then the program fetches the forecast data (see Section 2.5 for more information on the forecast API). If there is a precipitation probability less than 30%, then the program will turn on the water. This follows as the most simple approach to the water delivery process. However, it becomes slightly more complicated when the precipitation probability is greater than 30%.

In the case where we exceed the precipitation probability threshold, the algorithm first checks the rain flag. The rain flag is unique to each hose and its purpose directly relates to the rain timer.

We only want to set the rain timer once, which is the first time that the digitalOut tally exceeds the water level threshold but the precipitation probability is sufficiently high. Therefore, we need a way to determine if the rain timer has already been set, which is why we use a rain flag; it simply informs the system whether the rain timer has already been set. If the rain flag is not set, then the system sets the rain flag and the rain timer in order to handle the element of chance associated with precipitation.

If the rain timer is set, then the system checks how long it has been since the rain flag was set. We specified a threshold of 36 hours since the rain timer was set, but this can be changed in the user interface. If it has not been 36 hours, then the hose remains off. But if it has been 36 hours, then the hose will be turned on. It is also important to note that both the rain timer and the rain flag are reset whenever the hose is turned on. The bottom line is this algorithm handles when to turn on and off the hose taking into account the sensor data and the forecast data to efficiently control the irrigation system.

2.3 Data Logging

The goal of the data logging portion of the project is both to allow for smooth and efficient logging of the measured sensor data into a comma-separated-values (.csv) file and to transfer the most recent sensor readings into the SQLite database. The user can then quickly move or import the data into a data analysis program (such as Microsoft Excel) to interpret the data in a more useful and effective manner.

2.3.1 Logging the Sensor Data into a .csv File

The data logging portion of the project exists solely on the system's central hub, on the Raspberry Pi. The program operates alongside the entire wireless network and user interface software included within the main central hub program.

Data logging works by taking in the extracted sensor data and printing it out to an output file specified by the user, where it must be in the form of either a .txt file or a .csv file. A .csv file is a special type of .txt file; the main difference between these two file types is that .csv files are commonly used in data analysis software because of its use of a standard delimiter between data values.

Error checking functions initialize the data logging process; these functions ensure the input arguments are valid for the functions that would be used for completing the data logging process. Explicitly, the first error check guarantees the user calls for an output file. If there is no resultant output file, the program returns an error to indicate a failure within the data logging portion of the system. It is crucial for the user to address this error if it arises, otherwise data logging will not be functional.

The next operation of the data logging program opens the output file with the "append" specifier, allowing the program to tack on additional content at the end output file. By utilizing the append feature, the program retains the data from previous sensor measurements, and only updates the file with new readings thereby maintaining a collection of data spanning over an extended period or trial.

The data logging program uses file print statements to print the sensor data from the sensor nodes to the output file in an organized and readable manner. The initial print statement outputs the header of the data columns in the file; this operation is only performed once during the initialization of the central hub program. Each consecutive print statement effectively copies over a specific sensor data element and outputs it into the specified column within the output file. After data logging for this specific loop iteration has completed, the output file is closed to prevent corruption of the data. An example of the resulting .csv file output is presented in Figure 14. Within the output file, line 1 shows the headers of the data columns, each corresponding to one of the different sensor data measurements. The lines below the header represent individual data samples from all sensors, where one would be added every time another measurement of sensor data is taken.

```
1  Soil_Moisture,Ambient_Light,Ambient_Temp,Barometric_Pressure,Precip_Prob,Digital_Output,Node_ID,Battery_Level,Hose_1,Hose_2,Hose_3
2  82.654198,111.527695,0,0,0.000000,0,5,25,0,0,0
3  82.656693,111.909081,0,0,0.000000,0,4,0,0,0,0
4  81.737381,111.146317,0,0,0.000000,0,5,16,0,0,0
5  82.008018,111.909081,0,0,0.000000,0,4,0,0,0,0
6  81.154556,110.002151,0,0,0.000000,0,5,7,0,0,0
7  76.209732,111.909081,0,0,0.000000,0,4,0,0,0,0
8  80.892334,110.764915,0,0,0.000000,0,5,3,0,0,0
9  76.442429,111.909081,0,0,0.000000,0,4,0,0,0,0
10 80.631729,111.909081,0,0,0.000000,0,5,41,0,0,0
11 75.681961,111.909081,0,0,0.000000,0,4,0,0,0,0
12 80.372711,111.146317,0,0,0.000000,0,5,27,0,0,0
13 78.222366,111.909081,0,0,0.000000,0,4,0,0,0,0
14 80.115257,111.146317,0,0,0.000000,0,5,10,0,0,0
15 75.457191,111.909081,0,0,0.000000,0,4,0,0,0,0
16 79.859352,110.764915,0,0,0.000000,0,5,5,0,0,0
17 75.518013,112.297012,0,0,0.000000,0,4,0,0,0,0
18 79.808472,111.146317,0,0,0.000000,0,5,18,0,0,0
19 75.919334,111.909081,0,0,0.000000,0,4,0,0,0,0
20 79.604958,111.527695,0,0,0.000000,0,5,7,0,0,0
21 77.154480,111.909081,0,1015,0.860000,0,4,0,0,0,0
```

Figure 14: Resulting .csv file output from the data logging portion of the central hub program.

2.4 Data Storage

We store sensor data inside an SQLite database on the Raspberry Pi's SD card. The database acts as a backup for sensor data in case there is an overall system issue. SQLite is a running library that implements a self-written SQL database engine, without a server or configuration. We used the SQLite library because it is a C-language library that implements a self-contained and high-reliable file-based SQL database engine which is publicly available. Additionally, SQLite reads and writes regular files directly into memory and the database file format is cross-platform.

2.4.1 Preparation for Sensor Data Transfer to the Database

To keep the database updated, we needed to isolate each 15 minute cycle of sensor data measurements from the entire collection of sensor data. A completely separate .csv file was necessary in order to collect only the most recent sensor data. While we could have used the main .csv file, it would have required significantly more processing power, as we would have had to continuously compare the contents of the database with the contents of the main .csv file. As the contents of the database and .csv file increases, the required amount of processing would increase, potentially resulting in timing issues for other functions within the main program. Instead, the separate .csv file makes the transferring of data to the database significantly smoother and less process-intensive.

This portion of the program begins very much like the main data logging program via error checking functions. These functions are just as essential as before, and the database transfer will not be functional if these errors are not addressed.

The only difference between the code preparing for the database transfer and the main data logging code (explained in Section 2.3.1) is the "write" command when opening the output file. It is important to note that the write command effectively overwrites the entire file, resulting an output file consisting of only the most recent set of sensor data measurements. Once the program closes the file, the program is ready for the next iteration of the loop, where it takes in a new set of updated sensor data measurements.

2.4.2 Accessing the Database

Accessing the database requires the use of the SQLite3 C interface, as there is no other way to use the necessary library routines outside the SQLite3 environment. The C interface allowed us to execute SQLite3 API routines from the main program to perform processing on the .csv file as well as insert data into the database.

2.4.3 Processing the .csv File

Before the .csv file can be processed, the table inside the database needs to be created. After creating and opening the database, the program creates a table inside the database. Construction of the table occurs only once during setup, before any sensor data measurements are taken. The table is where the data is stored and has a header and column for each sensor data measurement.

Processing the .csv file involves opening and reading the created file in order to prepare for the transferring of data. The first line (consisting of column headers) is bypassed as the headers are only used for reference within the .csv file. The second line of the .csv file is tokenized and assigned to a ten-character array. After each sensor data element is assigned, the value is converted to either an integer or a double depending on the expected output.

2.4.4 Inserting the Data into the Database

The data is inserted into the SQLite database after tokenizing the sensor data and converting it into the proper data types. Inserting the data into the database involves binding each data variable to a prepare statement, which is then executed all at once. In this instance, the prepare statement is a character by character command inserted into the SQLite3 shell which allows the program to insert multiple data elements into the database simultaneously. As shown in Figure 15, the contents of the updated database are accessible from the SQLite3 environment running on the Linux terminal. Each row within the database is assigned an ID in the first column, and each column corresponds to the type of sensor data measurements from the central hub. Additional rows are added each time sensor data readings are taken to provide the user with the most updated version of the database.

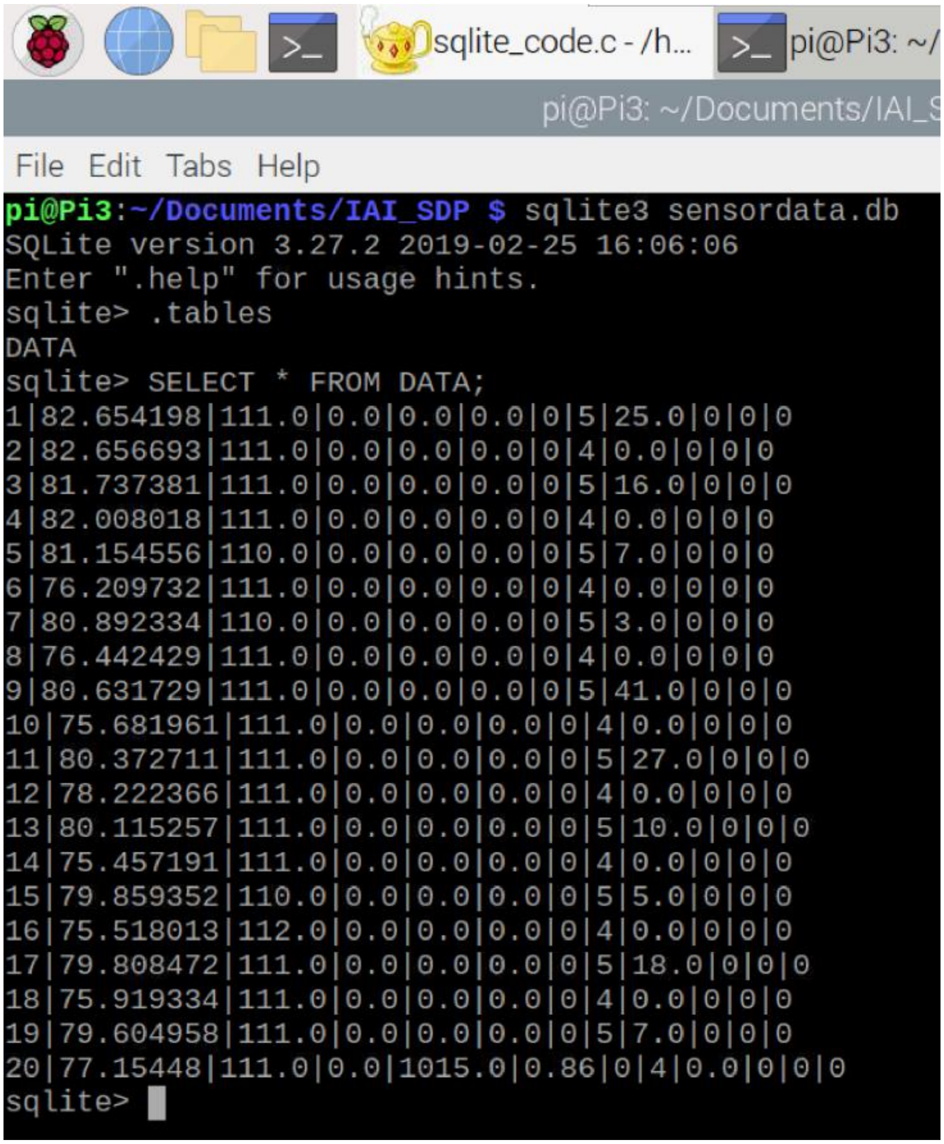


Figure 15: Contents of the database after transfer of the sensor data.

2.5 Weather Forecasting API

The purpose of the Weather Forecasting API is to implement predictive weather data and prevent unnecessary watering. The data from this API is used in the final watering algorithm and provides the user with additional information depending on their needs and preferences. To implement this, we used the Dark Sky Forecast API found at:

`https://darksky.net/dev.`

A weather API is a way for developers to access forecast information provided via radar data for a vast range of locations worldwide. The Dark Sky API is one of a few available weather API options; we decided on the Dark Sky API due to its ease of use and the minimal cost for our design. The Dark Sky API takes radar data from the USA NOAA's NEXRAD system in order to provide a accurate data regarding weather conditions.

2.5.1 Embedding the API Inside the Main System

Ideally, we would prefer to use a C-oriented program to generate forecast data, but our options were severely limited in this regard. Instead, we decided to implement the forecast API in Python. However, our local master file (see Section 6.4) is written in C++ which does not have the ability to embed python code. To get around the issue of conflicting languages, we called the weather API from the Unix shell in the central hub's main C++ program by simply opening a python script where the necessary API code was located.

2.5.2 The Dark Sky API

To determine our location, the Dark Sky API takes in three arguments: an API key, a latitude value, and a longitude value. The API key requires a simple Dark Sky registration, which is free for any user and includes a daily call limit of 1,000. Since our system calls for data every 15 minutes, we will not exceed more than 96 calls per day, which is well below the daily limit. During each call, the API returns a forecast in the form of a .json map containing all of the weather data using the three arguments covered above. This forecast object includes multiple classes including daily, hourly, and current weather forecasts each with over 100 forecast data parameters, most of which are not required for our watering algorithm. To isolate the measurements we want, we filtered the parameters one by one out of the .json map object containing all of the forecast data. The process of filtering involved deciding whether we wanted a daily, hourly, or current forecast measurement, as categorization of the map object is designed in this manner. We searched for the desired parameter location in the remaining data after reducing the forecast data by timescale. Once locating the desired parameter, the next step involves isolating the parameter by printing out the pointer object inside the specific array element containing the desired parameter.

2.5.3 Extracting the Forecast Data

Once all the measurements we wanted are printed out in the shell, they are read in from the main program. The main program then assigns these measurements to specific variables to be used for the system's watering algorithm (see Section 2.2.3).

2.6 Sensor Array

Located at each sensor node, the Sensor Array consists of three unique sensors that monitor soil moisture level, time of day, temperature, and barometric pressure. Figure 16 shows the schematic for the Sensor Array, which consists of a light level sensor, soil moisture sensor, and the Barometric Pressure Sensor modeled as R_PHOTO, R_MOISTURE, and J2, respectively. All three of these sensors are sampled once every 64 seconds by the sensor node's microcontroller and relayed to the central hub to make optimal watering decisions.

Each sampling period starts by closing the switch between the 5V rail and Vcc by using a high-side P-channel MOSFET, the DMP2110UW-7, which is displayed as Q1.[10] Driving this MOSFET's gate with a digital low will ensure operation in the deep triode region, where R_{DS}(on) stays low at approximately 100 mΩ. Once this switch is closed, the microcontroller will wait about 1 ms before sampling. This will give more than enough time for the MOSFET to turn on (t_{ON} of 7.7 ns) and the sensor signals, Analog_Light and Analog_Moisture, to charge (max RC of 10μs). It is important to note that after about 5 RC time constants have passed when charging a capacitor, the voltage across that capacitor will be at about 99.3% of its steady state DC value. Since the time between enabling and sampling is well above 50 μs, the sensor readings for moisture and light levels will be at steady-state values when read by the microcontroller.

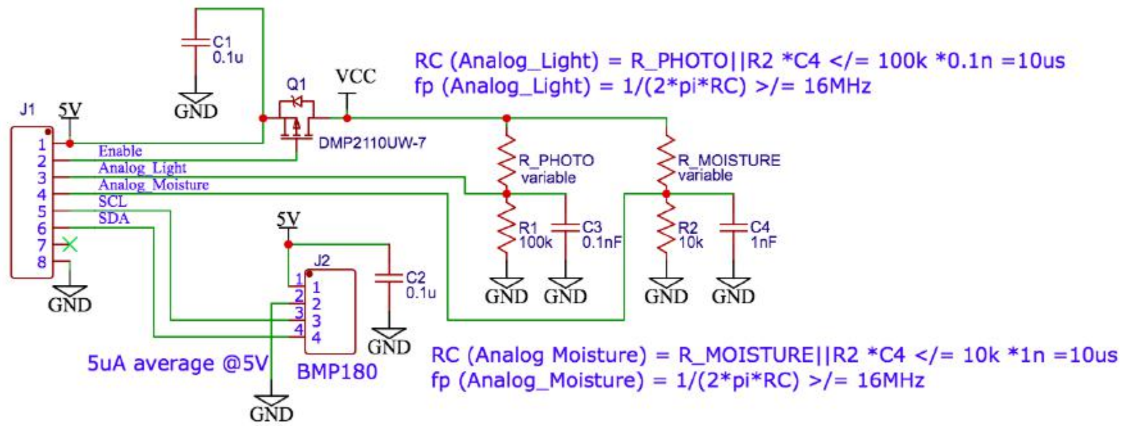


Figure 16: Version 4 of the Sensor Array Schematic. [7]

The Sensor Array only consumes about 900 μA when sampling. By only enabling this switch for 500 ms during each sampling state, the sensor node can preserve power consumption that would otherwise be wasted if the sensors were continuously. Using the power equation with the known current passing through the MOSFET and its characteristic R_{DS}(on), we see the power it consumes is very low, shown in Equation 1.

$$P = I^2 R = (900\,\mu A)^2 (100\,m\Omega) = 81\,nW \tag{1}$$

As for the layout and assembly within the sensor node (shown in Figure 17 and Figure 18), we see the physical design keeps a small footprint, with the surface area dimensions of the PCB at just 1.5 by 1.4 inches.

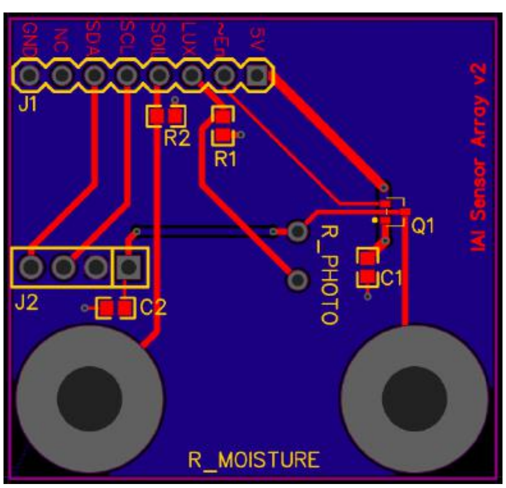


Figure 17: Sensor Array Layout.

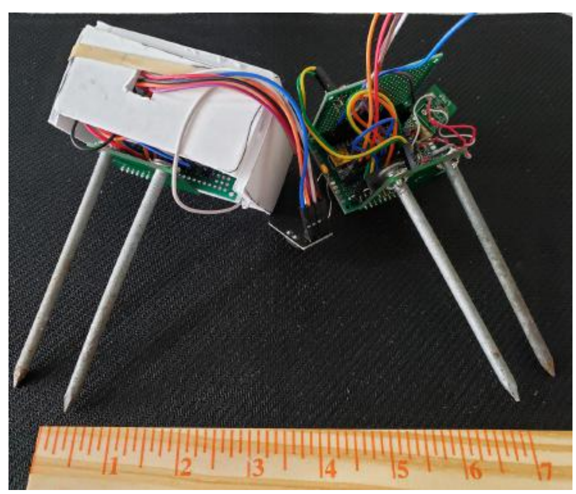


Figure 18: Sensor Array Assembly.

[7]Figure created using EasyEDA [14].

2.6.1 Soil Moisture Sensor

Soil moisture sensing in this project works under one fundamental premise: the consistent relationship between a particular soil's resistivity and its relative moisture content. The moisture content metric of interest is the gravimetric water content - the ratio between the weight of water in a soil sample divided by the weight of that soil when completely dry. It is easy to calibrate the soil moisture sensor for this metric of water content since it only requires a weight scale to determine the soil's change in water volume. Measuring resistivity, on the other hand, is not quite as simple. Equation 2 shows that the resistivity (ρ) of a medium, in this case the soil of interest, is a function of the material's resistance as well as the cross-sectional area (A), and the length of that medium (L).

$$\rho = \frac{AR}{L} \tag{2}$$

Since measuring the area and length of a soil sample can be imprecise, the sensor instead fixes the distance and depth at which two probes are placed in the soil and measures the resistance between them. This maintains the usefulness of the soils' resistivity and moisture content relationship since the resistance becomes a fixed multiple of the resistivity, so long as the depth distance between the probes are constant. Using this relationship, the soil moisture sensor uses two 20D, 4-inch length, common hot-dipped galvanized steel nails as probes. These nails have an exterior zinc coating which provides protection against oxidation over the sensor's lifetime. The area and length of soil between the probes remains fixed by spacing them precisely 1 inch apart.

Under this configuration, the moisture sensor shows a clear exponential trend between its resistance reading and the gravimetric water content of a particular soil so long as soil compactness remains constant. To show this trend we conducted six unique trials following the standard operating procedure in Section 6.3, which measures the sensor's resistance with increments in gravimetric water content for a fixed soil sub-sample with evenly distributed moisture and fixed compactness. These six trials tested the resistance readings and soil moisture content for three soil sub-samples, all taken outside of UC Santa Cruz's Baskin Engineering. Following the same sensor design, we created additional soil moisture sensors and re-tested the same soil sub-samples in order to verify consistency of the design. The experimental results for these six trials are presented in Figure 19, where each individual trial is distinguished by color.

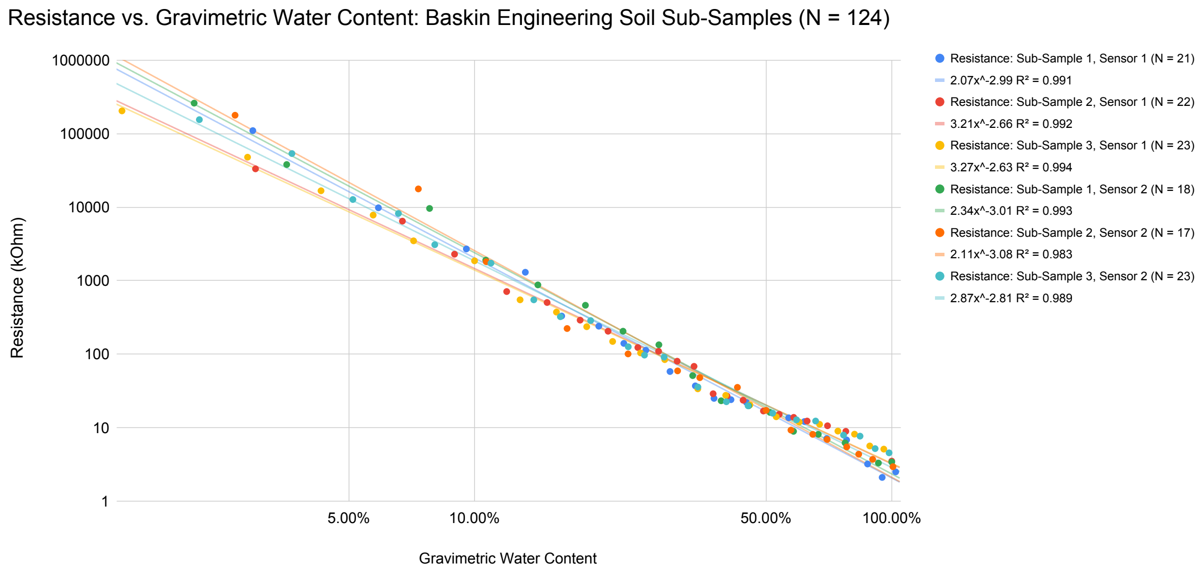


Figure 19: Soil moisture sensor performance using two unique sensors three unique soil sub-samples from the same location just outside Baskin Engineering.

The functional curve that accurately fits the sensor's resistance follows an exponential relationship, given by $R = ax^{-b}$ where R is soil resistance, x is the gravimetric water content within the range of 1% and 100%, and a and b are regression coefficients. Across all six trials, we see the spread between these curves remains small, so by aggregating this data in Figure 20 we form a single trend line which maintains a high correlation over the entire data set.

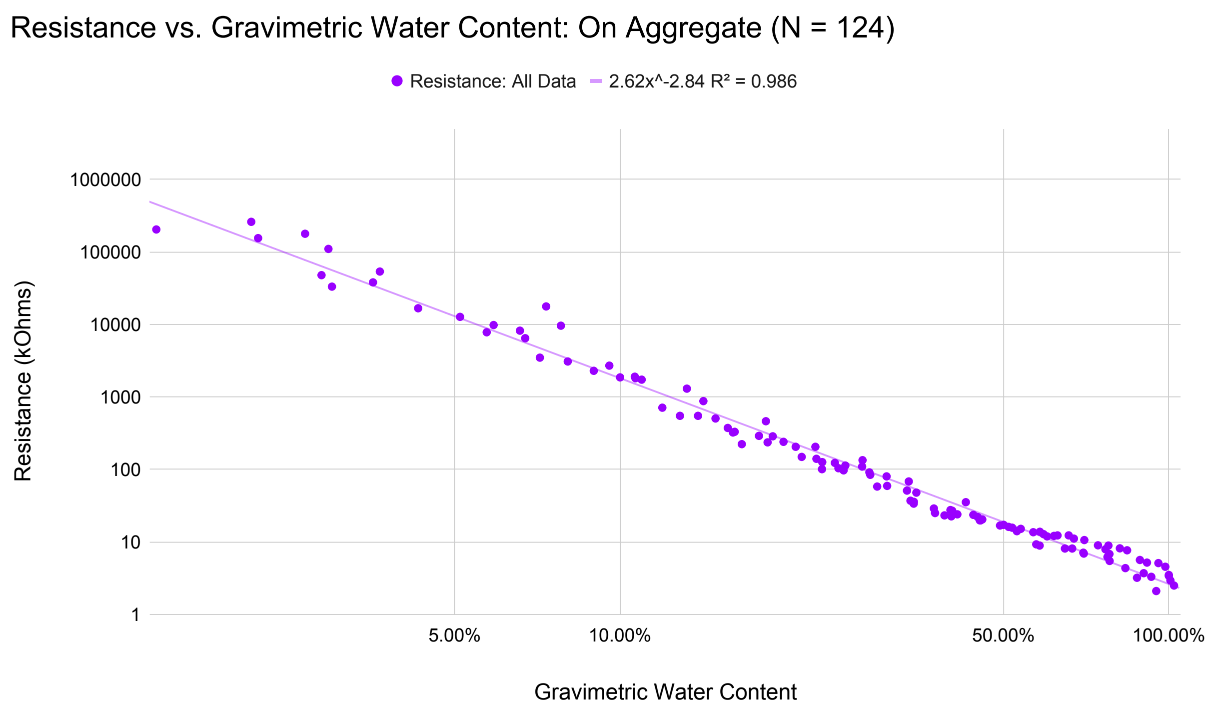


Figure 20: Soil moisture sensor performance and characteristic regression from data on aggregate.

This data of 124 samples nets a functional model of $R = 2.62x^{-2.84}$ and a coefficient of determination, R^2, of 0.986. This value means that with the data collected, we estimate that 98.6% of the differences in moisture sensor readings can be explained by changes in gravimetric water content. However, it is important to note that this coefficient of determination has only been shown under these known caveats: soil samples outside Baskin Engineering, fixed soil compactness, and homogeneous water distribution within the soil. We estimate that a true universal moisture sensor with this accuracy would need to model or calibrate for soil compactness and the specific soil sample it is measuring. Additionally, there is no simple solution to ensure homogeneous moisture distribution in the soil. But in general, the results found thus far show the usefulness of the soil moisture sensor for finding gravimetric water content, so long as these key caveats are met.

While this aggregate model is by no means perfect for soil moisture monitoring, this regression model is ultimately used by the software to map sensor readings to gravimetric water content within our system. To extrapolate the resistance across the probes, we use a voltage divider between the probes (represented by R_MOISTURE) and the fixed load resistor R2 shown earlier in Figure 16. Using a fixed 10 kΩ load resistor under a 5 V rail, the sensor node's microcontroller simply reads the analog 0 V to 5 V output at the voltage divider and solves for the unknown resistance. With the resistance known, it then solves $R = 2.62x^{-2.84}$ for x, effectively mapping the voltage to the gravimetric moisture content in the soil.

2.6.2 Light Level Sensor

The light level sensor is responsible for measuring the sunlight intensity, measured in lux, which is used to distinguish between night and day. Accurately recording light intensity allows the watering algorithm, explored in Section 2.2.3, to water more frequently during periods of lower light levels, effectively minimizing water loss due to evaporation. The light intensity is measured using the WODEYIJIA GM5539 photoresistor. This sensor's resistance decreases as the effective light intensity increases. The implementation of the light level sensor is shown in Figure 16 where R_PHOTO is the photoresistor and R1 is the fixed 100k load resistor. The circuit is almost identical to that of the soil moisture sensor, since both use a voltage divider under a 5V rail with a fixed load resistor to extrapolate the resistance of their corresponding sensors.

Since specifications for the GM5539 give an upper and lower bound for its resistance at varying levels of lux, we mapped these resistance values to corresponding ADC voltages using our voltage divider, shown in Equation 3. For example at 40 lux, the intensity of an overcast sunrise or sunset, the photoresistor sits between 10 kΩ and 26 kΩ which puts the output voltage between 4 and 4.5 V [31].

$$V_{out(max)} = 5V(\frac{100k\Omega}{R\_PHOTO_{(min)} + 100k\Omega}), V_{out(min)} = 5V(\frac{100k\Omega}{R\_PHOTO_{(max)} + 100k\Omega}) \tag{3}$$

Graphing these values gives the plot shown in Figure 21, where blue represents $V_{out(min)}$ and red represents $V_{out(max)}$. As a reference, approximately 1 lux is the brightness of a full moon, while 40 lux is that of an overcast sunrise.

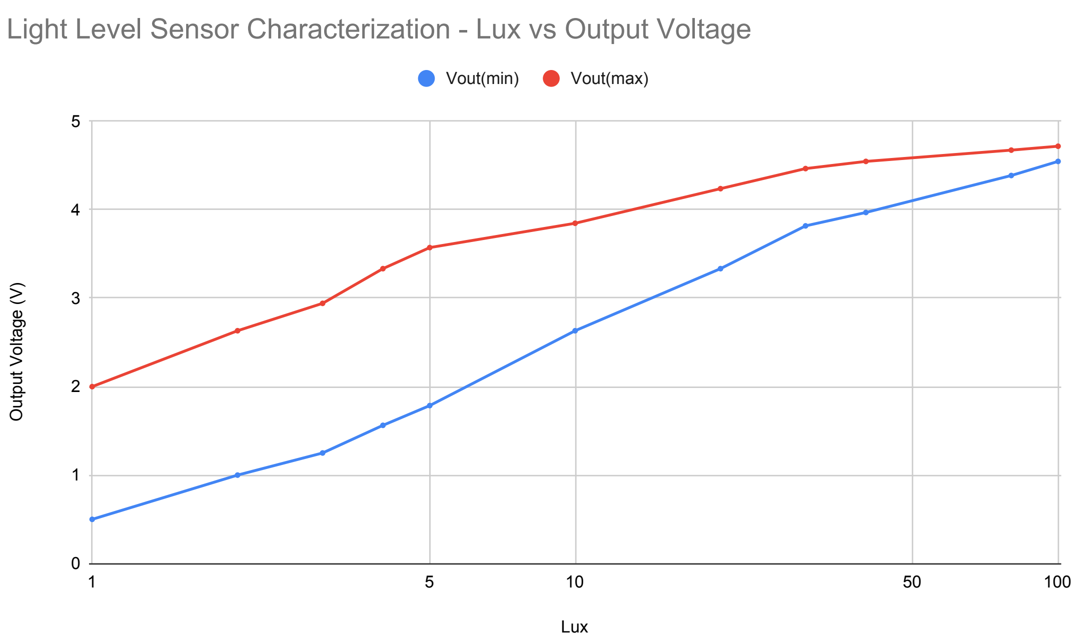


Figure 21: Light Level Sensor Characterization Graph. The reading tolerances between GM5539 photoresistors nets the following range of output voltages at each light level [31].

In the final model, we implemented an exponential regression on the characteristic curve for $V_{out(min)}$, effectively giving software the means to project the maximum lux value observed by the sensor after reading the analog voltage, shown in Equation 4 where the voltage reading at the

ADC is x. This regression model nets an R^2 of 0.978.

$$\text{Light Level (lux)} = 0.502e^{1.14x} \tag{4}$$

In the final algorithm, our system applies this equation to map the raw voltage values into lux. The default light level threshold (used in Section 2.2.4) was set to 30 lux based on empirical data in order to differentiate between night and day. This threshold, however, can be changed by the user through the user interface. We chose the 100 kΩ load resistor since it provided the best resolution at and around the two most important light levels, sunrise and sunset (approximately 40 lux). Being able to identify these two light intensities is critical since the system aims to minimize watering loses due to evaporation. Sunrise is arguably that most ineffective period to water since any recently-supplied water will experience prolonged exposure to daylight, while sunset is an ideal time to water since it represents a prolonged period of no sunlight, and subsequently minimal evaporation. Using the light level sensor, the system is able to recognize periods of low sunlight intensity and time our water distribution such that losses due to evaporation are minimized.

2.6.3 Barometric Pressure and Temperature Sensor

The barometric pressure and temperature sensing at the node level is implemented by the BMP180 board from Adafruit shown in Figure 22. The BMP180 board is simple, compact and offers dedicated libraries for its I2C communication protocol. This sensor allows the system to take in real-time data for barometric pressure and temperature, and look for substantial deviations in weather and subsequently change its watering patterns. These events include reduced watering when rain is imminent to prevent over-watering along with prolonged watering during freezing weather to insulate and protect plants from cold shock.

The specific values for which the barometric pressure readings predict storms still requires additional research, but in general when the barometric pressure acutely decreases, this indicates to the system that there is a high probability of a looming storm. As for preventing damage due to freezing, the goal is to keep the plants well above 0°C. This means at and near freezing temperatures, the system will automatically start watering to prevent the plants' roots from freezing. Water provides insulation which improves the odds of the plant surviving. Using data reported from the BMP180 in tandem with the weather forecasting API (outlined in Sub Section 2.5), the system can accurately determine the climate that the garden is experiencing and optimize its watering algorithm accordingly.

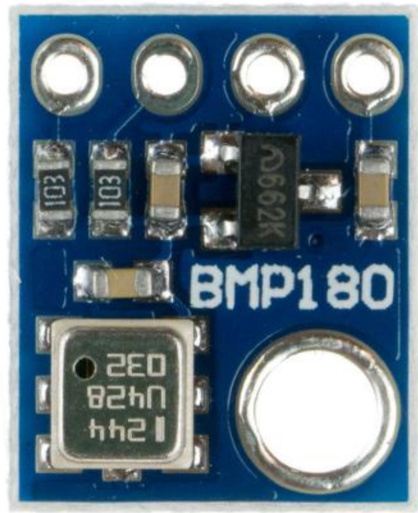


Figure 22: Adafruit BMP180. This breakout board allows the system to read temperature and pressure to determine weather deviations and change watering accordingly

2.7 Automated Water Delivery

In order to precisely and autonomously control the water delivery, we have designed circuits for the central hub system to convert low-powered electrical signals from the microcontroller to actuate a latching solenoid valve (LSV). Since our system aims to support large irrigation setups with multiple hoses, we devised a method to provide electromechanical control for up to four LSVs. With this functionality, users can configure the water delivery system for a wide range of applications.

Along with actuating the solenoid valves, it is important to monitor the amount of water that is dispensed from our irrigation system. This is accomplished through the Digiten FL-608 flow sensors which are added in series with each LSV. The signal for each flow sensor is read through the digital pins of the central hub SBC, where the corresponding tachometer (tach) frequency from the flow sensor can be mapped to flow rate and subsequently total water output. Using these sensors, the central hub tracks the total water consumption at each hose allowing the system to check for faults in the water delivery and relay water usage information back to the user.

The design elements of these sub-components, which together make up the full water delivery system, are discussed in further detail throughout this section.

2.7.1 Electronic Control of Valve Actuators

Electronic valve actuators provide the ability to convert electrical signals into physical actuation (opening/closing) of a valve so that water flow can be either blocked or permitted. For the actuator components, latching solenoid valves, or LSVs (displayed in Figure 23), were chosen since they operate with quick pulses of electrical current, causing an internal piston to magnetically latch in either the open or closed position [23]. Control over the valve's position is determined through the polarity of the pulse used to actuate it, where a +5 V pulse of at least 50 milliseconds across the LSV opens the valve, and a -5 V pulse closes it. This driving method results in significant reductions in energy consumption over long periods of operation when compared to continuous solenoid valves which need to be constantly driven to trigger water flow.

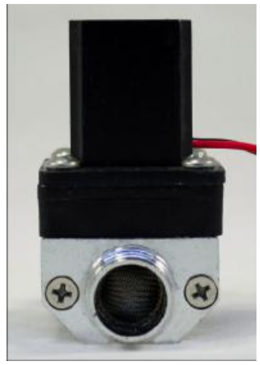


Figure 23: Latching solenoid valve used to control the flow of water for the system.

A major constraint with driving these actuators from our central hub is the limited 53 mW power output from the SBC GPIO (general-purpose input/output) pins. The GPIO pins allow for a maximum current output of 16 mA for digital 3.3 V signals which is not enough to drive the LSVs since they require a ±5 V pulse with at least 289 mA. As a way to get around this issue, we designed an H-Bridge driver to convert low-powered 0-5 V digital signals into ±5 V electrical pulses drawn from the 5 V power rail. Using the power rail to drive the LSV instead of the GPIO pins directly provides a higher current limit of 1 A which is more than enough to drive a single LSV.

Now with enough current to simultaneously drive 4 LSVs, the central hub runs into another problem with its excessive usage of GPIO pins. In order to drive multiple LSVs from the single central hub while reducing the number of the utilized GPIO pins, we designed a circuit which would level-shift and demultiplex the 3.3V digital signals so that multiple LSVs could be controlled using the same four GPIO pins. The block diagram outlining the set up for electrically controlling multiple LSVs is shown in Figure 24. The circuit design for both the H-Bridge Driver and the Level-Shifting is discussed more thoroughly within Section 2.7.2 and Section 2.7.3 respectively.

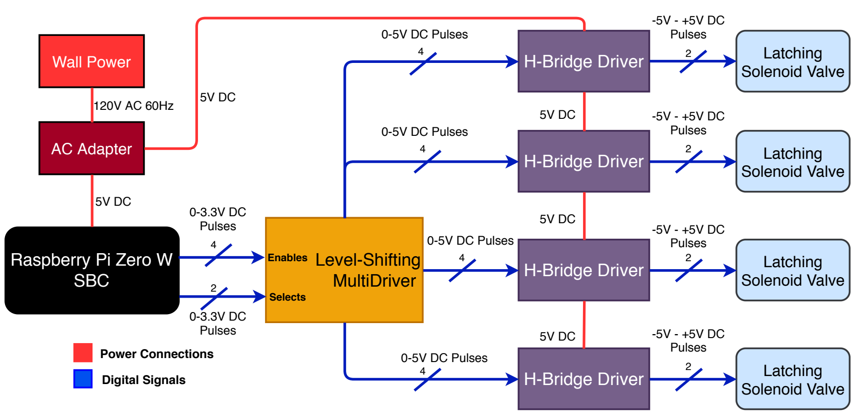


Figure 24: Block diagram outlining electrical signals used to actuate the LSVs. [8]

[7]This figure was made with Draw.io. [4]

2.7.2 H-Bridge Circuit Design

The purpose of the H-Bridge Driver circuit is to provide sufficient power and precise control over the actuation of a single latching solenoid valve. Since these valves operate in response to a ±5 V pulse, four MOSFETs (metal–oxide–semiconductor field-effect transistors) are driven with individual voltage signals at their gates to connect either 0 V or +5 V to the LSV terminals. Depending on which LSV terminal is applied with +5 V, the LSV will either open (+5 V across the LSV) or close (-5 V across the LSV). By choosing which set of transistors are turned on, this method allows the central hub to control the duration and polarity of the actuating pulses. The schematic of the fully designed H-bridge driver circuit V8 which allows for electrical control of a single LSV is shown in Figure 25.

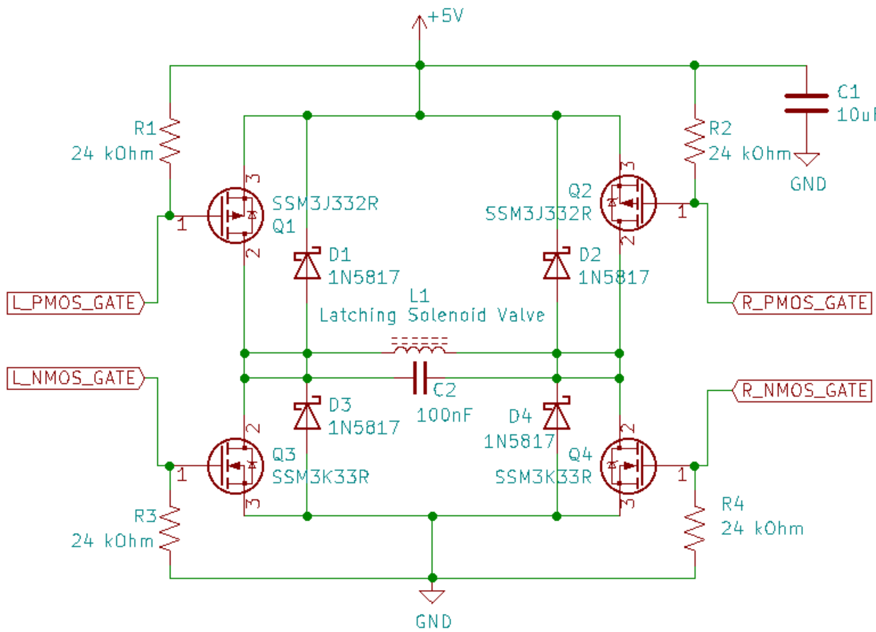
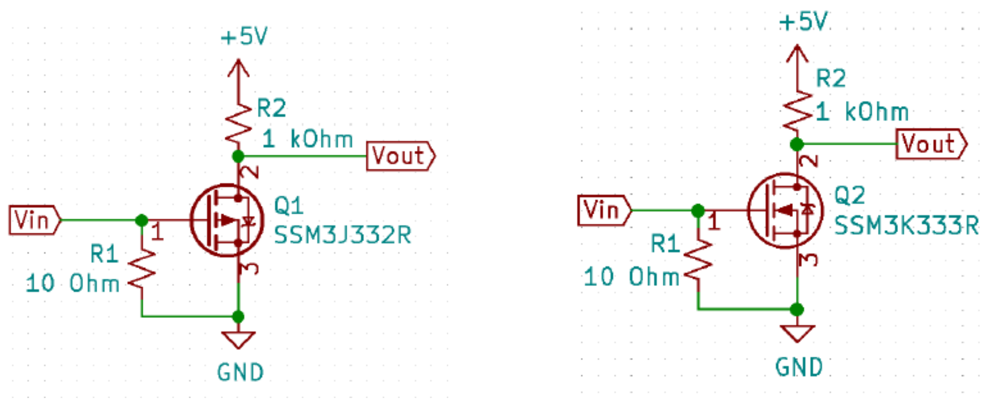


Figure 25: Schematic of H-bridge Driver circuit V8 responsible for actuating the LSVs.

This H-Bridge circuit consists of four MOSFET transistors with Schottky fly-back diodes and pull up/down resistors at the gates to ensure the transistors are turned off when the inputs at the gates are floating. For the transistor components, we chose to utilize the SSM3J332R P-Channel Field Effect Transistor [28] and the SSM3K333R N-Channel Field Effect Transistor [27] for their fast switching times, 4.5 V drive capabilities, and low-on resistances of 50 mΩ and 42 mΩ respectively. The fast switching capability of these transistors is important for precise control of the pulse used to drive the LSV.

To determine the actual switching times of the PFET and NFET experimentally, we wired the transistors according to the testing schematics shown in Figure 26. During this test, the gate of each transistor was driven with a +5 V 1 MHz square wave and the voltage at the drain was scoped using the Analog Discovery 2 (AD2) oscilloscope. The delay observed when transitioning between high and low output signals was then recorded.

[8] Figures were created using KiCad EDA. [14].



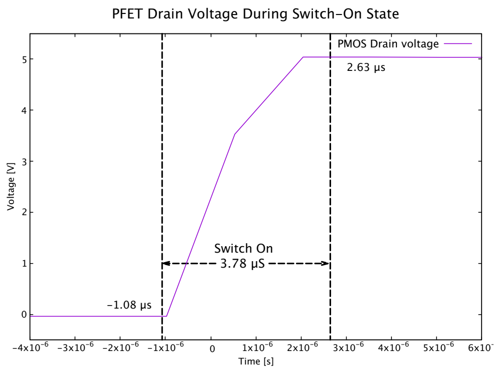
(a) PFET Testing Configuration

(b) NFET Testing Configuration

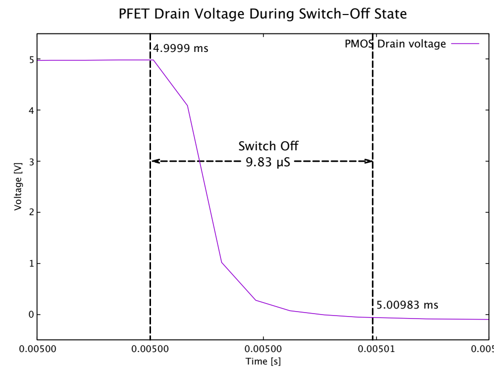
Figure 26: Schematics for the two testing configurations used to determine switch times for the individual MOSFET transistors where the gates are both driven with a 5 V 1 MHz square wave. [9]

From the test, we found that the SSM3J332R PFET had an operating switch-on time of 3.78 μs and switch-off time of 9.83 μs. Similarly, the SSM3K333R NFET exhibited a switch-on time of 7.88 μs and a switch off-time of 3.25 μs. The resulting oscilloscope traces taken from an Analog Discovery 2 (AD2), shown in Figure 27, demonstrate the quick switching time of these MOSFETs where they were all found to operate under 10 μs. Since the central hub SBC is able to precisely control GPIO outputs down to a minimum of 1 millisecond, there will always be sufficient time to allow for the MOSFET transistors to properly switch on or off accordingly.

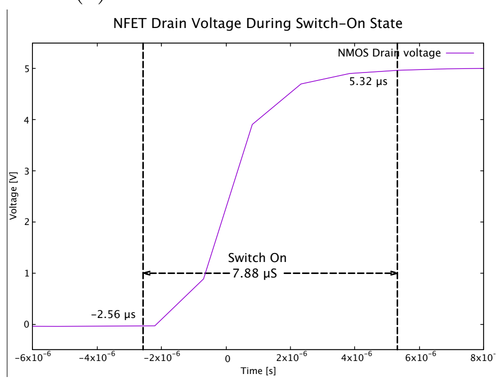
[9] Figures were created using KiCad EDA. [14].



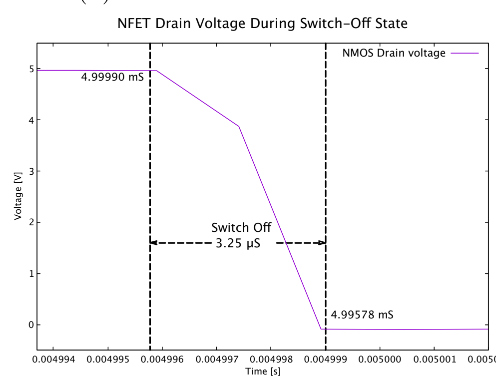
(a) PFET switch-on transition



(b) PFET switch-off transition



(c) NFET switch-on transition



(d) NFET switch-off transition

Figure 27: Drain voltages of the MOSFETs scoped using an Analog Discovery 2 to determine relative switching times.

Another vital consideration we took into account when driving the H-bridge circuit is the prevention of shoot-through current. Shoot-through current is experienced when both the NFET and PFET transistors along the same side of the H-bridge are turned on at the same time and a low-resistance path is thus created between power and ground. This effectively shorts our power supply, and wastes additional power, which could cause excessive amounts of heat to dissipate from the transistors and result in damaging the circuit components themselves. The most secure way of preventing shoot-through current is to ensure that only one transistor is turned on at a time on each side of the H-bridge. To accomplish this, we used software running on the central hub to drive each of the four gates on the H-Bridge individually and manually control the timing. We took into account the observed switch-on and switch-off times for the transistors, and found that all components responded within less than 100 ns. Using these measured switch times, we determined that a delay between gate driver signals of at least 1 ms would be sufficient to ensure that there is no point where two of the FETs are transitioning at the same time.

The timing diagram, shown in Figure 28, specifies the sequence of required driver signals in order to obtain the desired pulse of either +5 V or -5 V at the input of the H-Bridge. The gate driver signals are changed no less than 1 ms apart from each other to ensure no shoot-through current is experienced by the transistors. Software is used to keep track of the current valve position and control the timing of the driver signals depending on which actuation is desired. Figure 29 shows the experimental H-Bridge output with no load when driven to produce 100 ms pulses.

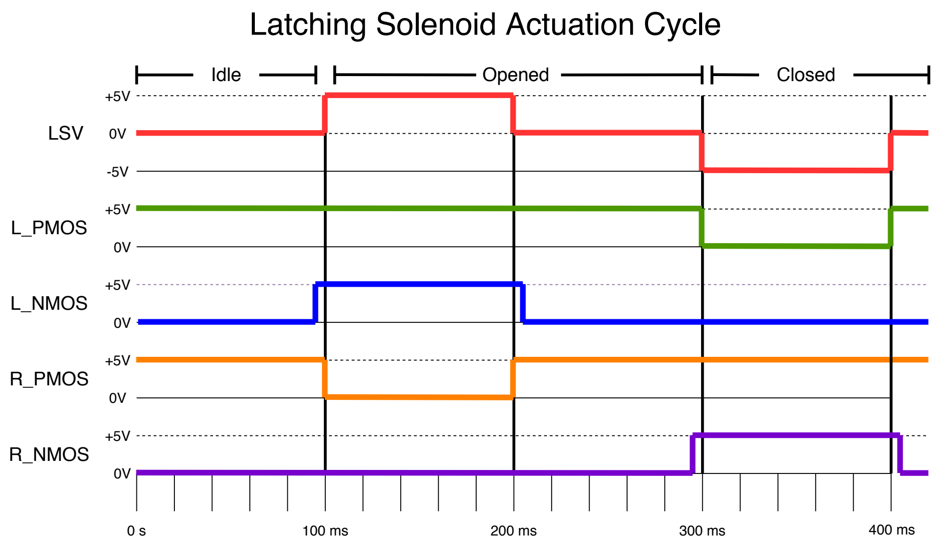


Figure 28: Timing diagram for the gate driver signals into the H-Bridge module corresponding to the cycle of opening and closing the latching solenoid valve. [10]

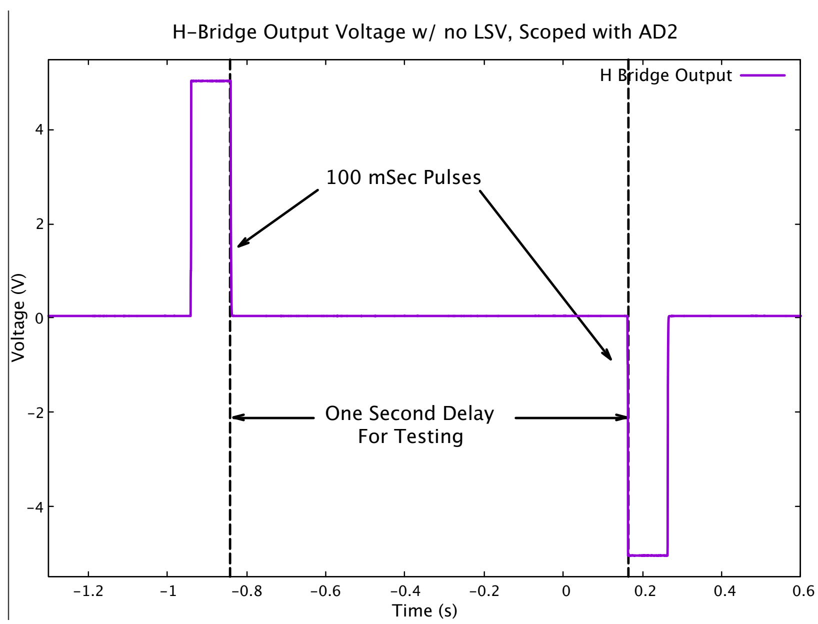


Figure 29: Analog Discovery 2 oscilloscope trace of the voltage across the H-Bridge output with no load when programmed to trigger 100 ms pulses.

Due to the sharp change of current experienced when a pulse is applied or reset, the internal coil within the LSV will induce a magnetic flux in the opposite direction of our coil, resulting in a large inductive voltage spike to compensate for the rapid change in current. If large enough, these voltage spikes can damage the SBC or other components in the circuit. To clamp the voltage and reduce these inductive spikes, the 1N5817 Schottky diodes [18] are used as fly-back diodes by connecting them in parallel with the LSV. When their low forward voltage of 450 mV is exceeded, the diodes

[10]This figure was made with Draw.io. [4]

begin conducting to allow for a current path back to the power supply thereby significantly reducing the large inductive spikes. When scoping the voltage across the latching solenoid valve during actuation, the observed trace shown in Figure 30 demonstrated the fly-back diode's snubbing effect on the inductive spike which was measured to be around 492 mV. The reduced inductive spikes did not present issues with utilizing other components on the central hub and the resulting ±5 V pulses were able to actuate a LSV consistently, thus proving the success of the H-Bridge V8 circuit.

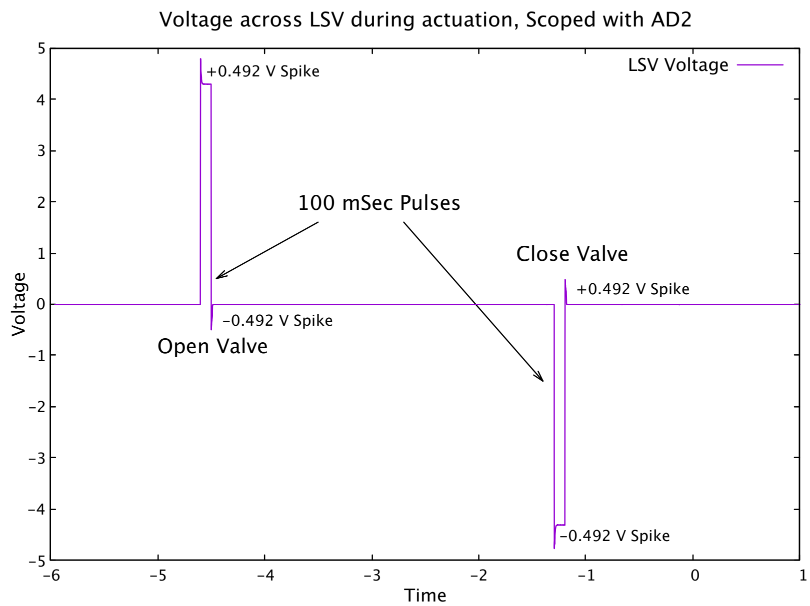


Figure 30: Analog Discovery 2 oscilloscope trace of the voltage across the latching solenoid valve terminals when driven with 100 mS ±5V pulses.

2.7.3 Level-Shifting Multi-Driver Circuit Design

Being able to trigger different paths of water flow is beneficial for users looking to irrigate larger gardens or plants requiring different amounts of soil moisture. The H-Bridge module can safely handle the actuation of a single LSV, however, there are a few complications we encounter when driving multiple H-Bridge modules directly from the SBC GPIO pins.

The first issue is that the Raspberry Pi Zero W has a limit of usable GPIO pins and each H-Bridge module requires four digital pins in order to control the timing of the gate signals. Since many of these pins are used for additional system components, we must limit the number of inputs required to drive multiple H-Bridge modules. Secondly, the Raspberry Pi Zero W is only able to output 3.3 V digital signals, yet the H-Bridge drivers require 5 V digital signals for proper actuation. Thus, to securely actuate multiple LSVs, we needed a solution that could modulate these signals. Modulation of these signals would thus be needed in order to securely actuate multiple LSVs.

Our approach to solving both of these issues came in the form of the Level-Shifting Multi-Driver circuit, the schematic for which is shown in Figure 31. This circuit would be responsible for both converting the 3.3 V signals from the SBC to usable 5 V signals and decoding between four separate sets of H-Bridge inputs. Through this method, up to four H-Bridge modules, each driving a single LSV, can be controlled individually using only 6 GPIO pins from the SBC. Determining which

H-Bridge module to drive is accomplished by switching the select line inputs for the demultiplexer.

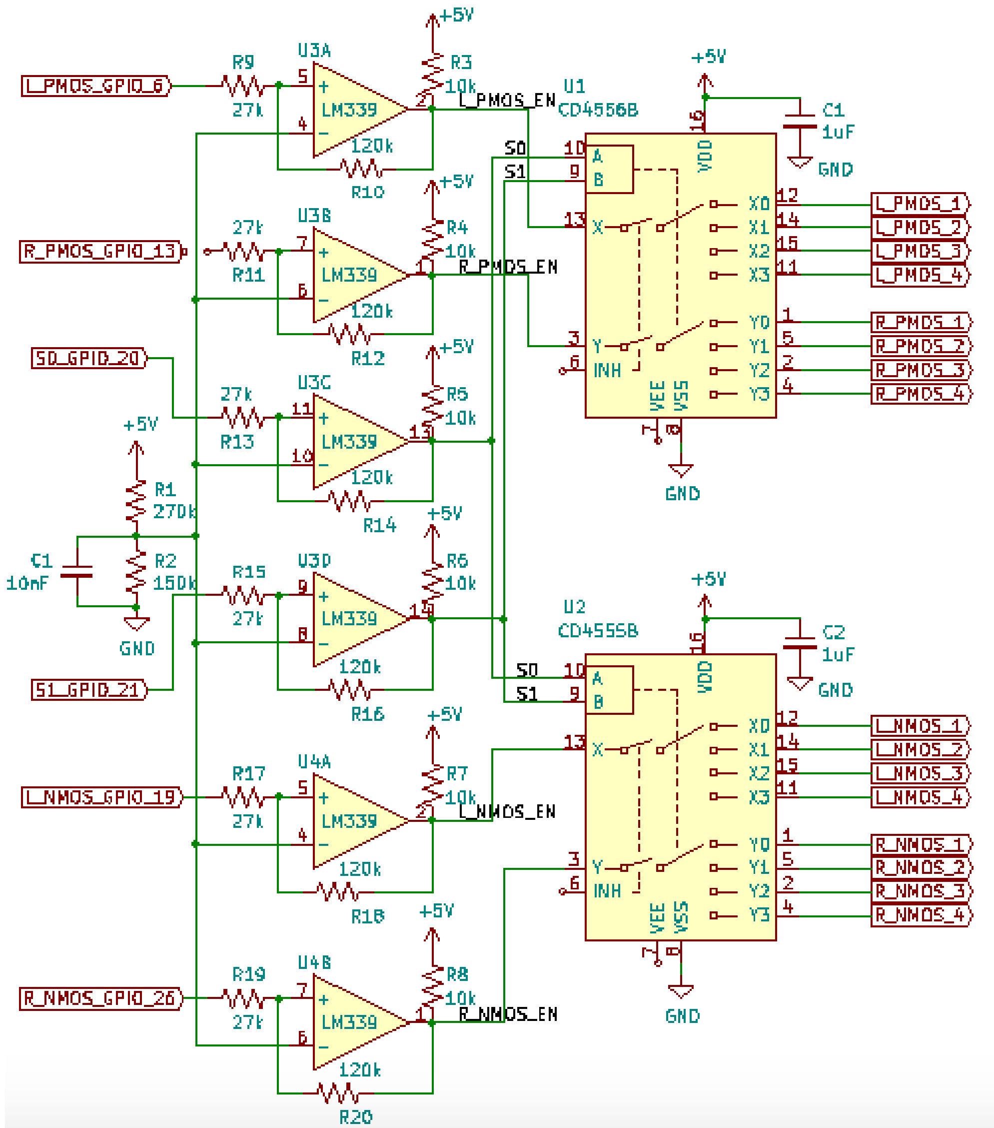


Figure 31: Level Shifting Multi-Driver Schematic V3 used to drive four H-Bridge modules with 3.3V logic-level inputs. [11]

This circuit utilizes the LM339 comparator powered at 5 V to rail the output voltage when the positive input of the component surpasses a the voltage fed into the negative terminal [12]. Conversely, the output is grounded once the positive input becomes lower than the negative input by at least . This comparator was chosen due to its availability and quick response time of 1.3 μs, which is well within the minimum 1 ms timing delay between the MOSFET enable signals.

Since the expected voltage levels at the input of these comparators range from 0V to a maximum

[11] Figures were created using KiCad EDA. [14].

of 3.329V, we implemented positive feedback in the circuit to set hysteresis bounds so that noise in the input signal would not cause a change in the output signal. Each comparator in the circuit utilizes a high voltage threshold of 2.19 V and a low voltage threshold of 1.07 V to provide roughly 1.12 V of noise immunity for each comparator output signal.

The voltage thresholds for these comparators are set so that the 1.12 V of noise immunity provides sufficient headroom from either power supply. Therefore, a logic-level high input signal is comfortably above the high comparator threshold and a logic-level low input signal is below the low comparator threshold. The resistive elements used to set the thresholds were determined using Kirchoff's Voltage Law analysis. Following the comparator, labeled U3C within the circuit(see Figure 32), the voltage thresholds were computed in relation to the reference voltage, V_{ref}, and resistors, R_1, R_2, R_{13} and R_{14}.

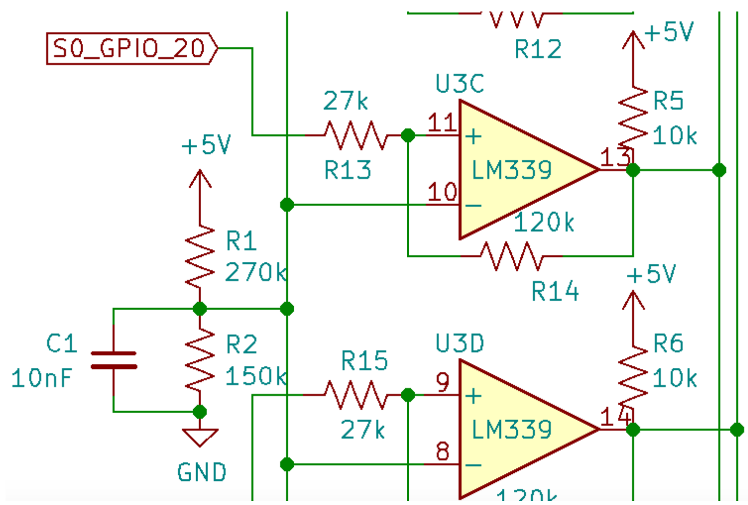


Figure 32: Close-up view of single comparator within the Level-Shifting Multi-Driver V3.. [12]

To provide a reference voltage close to the center of our expected input signal, we set our V_{ref} as

$$V_{ref} = V_{cc}(\frac{R_2}{R_1 + R_2}) = 5\,V(\frac{150\,k\Omega}{150\,k\Omega + 270\,k\Omega}) = 1.79\,V \tag{5}$$

The comparator's upper limit hysteresis threshold, V_{UT}, was set using $R_{13} = 120\,k\Omega$ and $R_{14} = 27\,k\Omega$ resulting in a voltage threshold of

$$V_{UT} = V_{ref}(\frac{R_{13} + R_{14}}{R_{14}}) = 1.79\,V(\frac{27k\Omega + 120k\Omega}{120k\Omega}) = 2.19\,V \tag{6}$$

Using the same method, we computed the lower voltage threshold, V_{LT} , for the comparator to be

$$V_{LT} = \frac{V_{ref}(R_{13} + R_{14}) - V_{cc}(R_{13})}{R_{14}}) = \frac{1.79\,V(27\,k\Omega + 120\,k\Omega) - 5\,V(27\,k\Omega)}{120\,k\Omega)} = 1.06\,V \tag{7}$$

The hysteresis voltage, V_H ,is defined as the difference between these two thresholds which is determined as

$$V_H = V_{UT} - V_{LT} = 2.19\,V - 1.06\,V = 1.13\,V \tag{8}$$

[12] Figures were created using KiCad EDA. [14].

Utilizing this hysteresis voltage of 1.13 V would prevent erratic switching of the output signal in response to slight changes in the input signal. Figure 33 demonstrates proper level shifting functionality from the comparators within the Level-Shifting Multi-Driver V3 circuit where the digital input signal is converted from 3.3 V to 5 V.

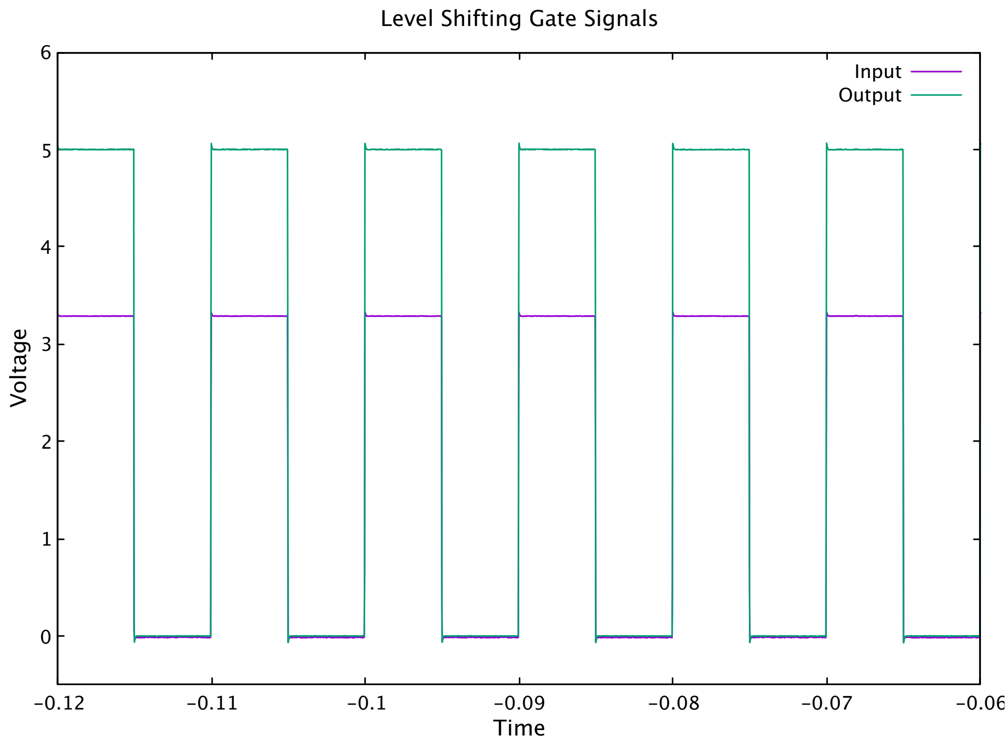


Figure 33: AD2 oscilloscope trace of input and output voltage of the LS Multi-Driver V3 circuit fed with a 100 Hz 0-3.3 V square wave as the enables.

Since the LM339 is an open-collector device, 10 $k\Omega$ pull-up resistors are used so sink current from the power supply when the output of the comparator goes high, effectively grounding the signal. The output into the active low enables of the demultiplexer remains at 5 V until the open-collector digital output goes high and causes the output to become grounded. Figure 34 displays the voltage across two separate LSVs during separate actuation events, where both valves are driven from individual H-Bridge modules and are connected to the same level-shifting demultiplexer circuit. Through the use of this circuit, the central hub consistently triggers multiple latching solenoid valves allowing for more extensive and customizable water delivery setups.

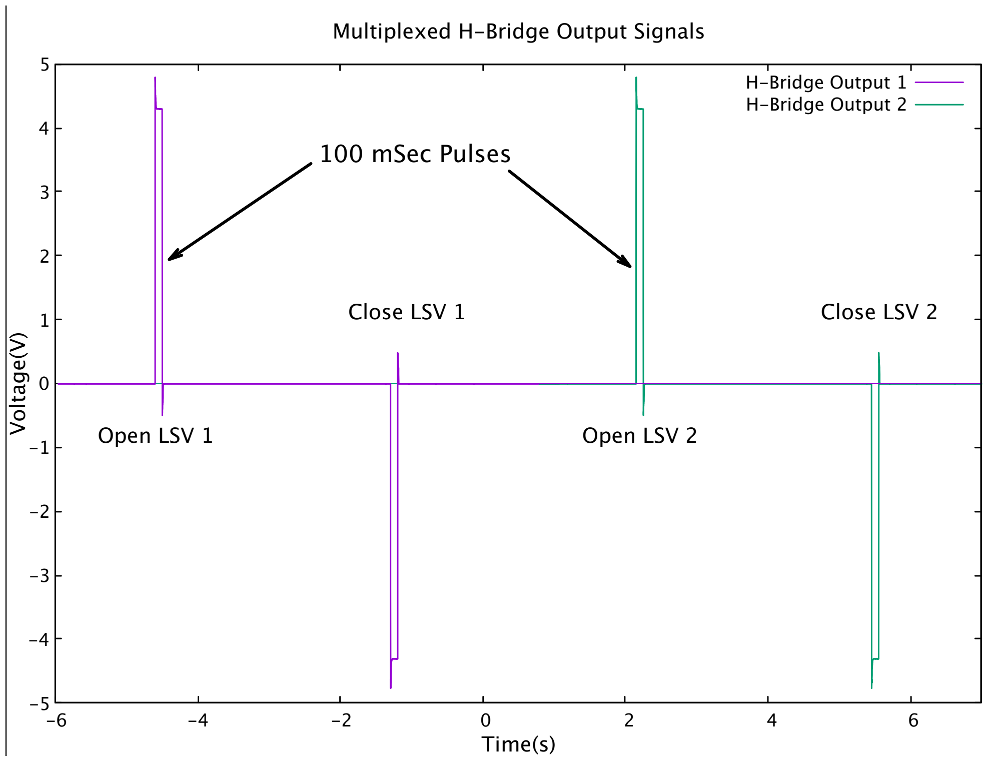


Figure 34: AD2 oscilloscope traces across two separate LSVs when actuated from the Level-Shifting Multi-Driver V3.

2.7.4 Monitoring Water Usage

In order to monitor water output and report this back to the user, we use the Digiten FL-608 (shown in in Figure 35) in series with the latching solenoid valves. This flow sensor has 3 pins, power, ground and tach. The tach signal, which is used to measure water output, sends a 50% positive duty-cycle signal whose frequency is proportional to the flow rate passing through the sensor in liters per second.

Figure 35: FL-608 flow sensor used to measure the total amount of water that is distributed through the water delivery network.

The FL-608 claims to have a a tach with a 330 pulses per liter conversion for flow rates between 1 and 60 L/min, but under our test this conversion factor did not hold, shown in Figure 36. This test puts the flow meter through various watering periods, counts tach pulses and measures the water output in liters. When sorting the data by rising edges tallied, however, we see a much stronger trend, shown in Figure 37. The trend is roughly logarithmic, where after about 6500 pulses the pulses per liter conversion settles to about 400.

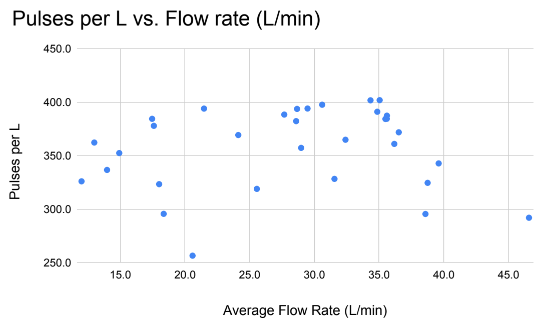


Figure 36: Experimental pulses per liter in FL-608 across average flow rates.

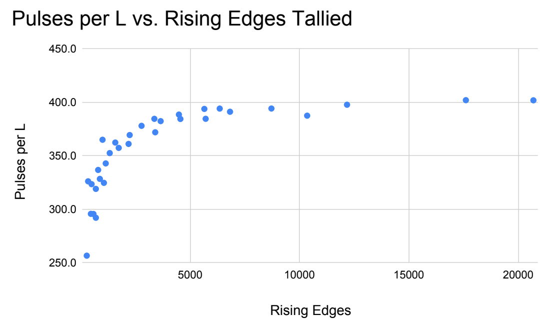


Figure 37: Experimental pulses per liter in FL-608 across tallied rising edges.

To model this varying pulses per liter conversion to an accurate functional curve, we implemented piecewise model, where a natural log regression models pulse per liter conversion for under 6500 pulses, shown in Figure 38, as 46.2 + 40.8ln(x) . This curve shows a relatively strong correlation with an R^2 value of 0.928. When monitoring outputs larger than 6500 pulses, on the other hand, the system will simply use a constant 404 pulses per liter conversion.

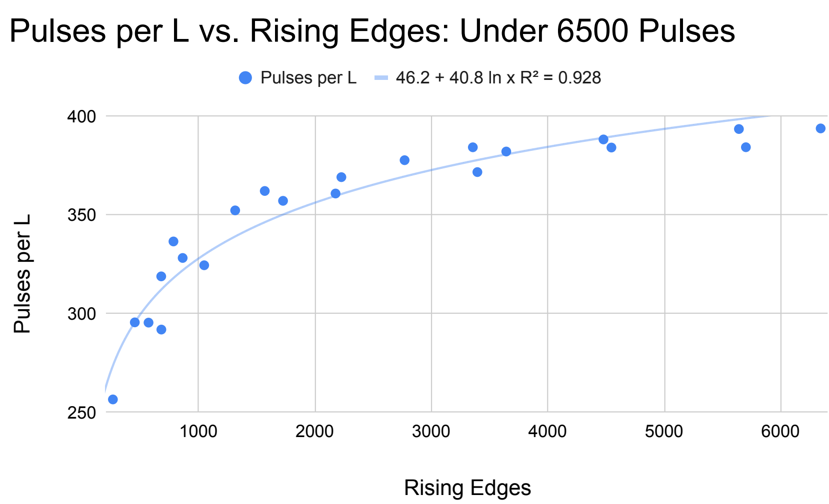


Figure 38: Pulses per liter data and functional curve for under 6500 pulses.

In order to tally water output for the user, the Central Hub software will simply record pulses and store this as a variable, x, then solve the piecewise function shown in Equation 9.

$$\text{Output (L)} = \frac{\text{Pulses}}{\text{Pulses per L}} \begin{cases} \text{Output (L)} = \frac{x}{46.2+40.8ln(x)} & x < 0 \\ \text{Output (L)} = \frac{x}{404} & x \geq 6500 \end{cases} \tag{9}$$

To confirm the accuracy of this piecewise model, we back-tested our data against the functional curve's projections and used a unique trial, which implements this function to display projected water output in liters. By measuring the percent error between the model's projected and actual output in liters, we see the error stays mostly below 5%, as shown in Figure 39 which has back-tested data in blue and the unique trial in red.

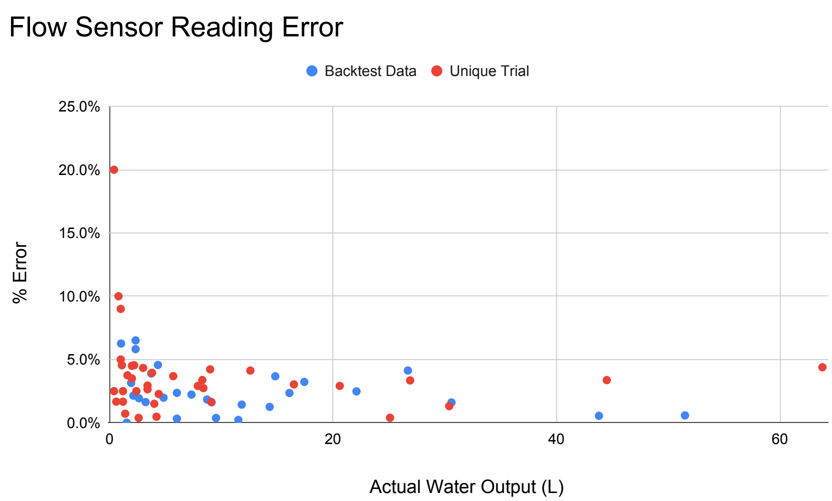


Figure 39: Flow sensor error distribution with back-tested data in blue and the unique trial in red

One issue with this data arises in small total water outputs, where the error term can be high when the sum of total water output passed through the flow meter is under 2.5 liters. This is likely due to the scale's measurement tolerance, 0.1 liters. This means outputs as small as 1 total liter can see error terms as large as 10%. For this reason, we can only expect very accurate water monitoring (under 5% tolerance) once 2.5 liters of water have passed through the flow meter, which is where the majority of watering periods will fall under. Additionally, this inaccuracy only affects the accuracy of water usage reported to the user and will not hurt the performance of the watering algorithm.

2.7.5 Waterproofing Electronics

Considering that water and electronics do not mix well and our irrigation system must operate in wet conditions, we want to be sure that there is clear isolation between the two in order to prevent injuries to the user as well as water-damage or shorting to our electrical components. The mechanical enclosures of the central hub and the sensor nodes, which will house several electronics including the Raspberry Pi microcomputer and Arduino Nano micro-controller, will be designed to act as semi-permeable barriers that will block direct contact with water while also allowing for small amounts of moisture to naturally escape through evaporation. Also, as an added measure of safety, we will be coating our PCBs in a transparent varnish that will protect them from short circuits and corrosion. This section will further describe the waterproofing techniques we plan to implement in our system as well as the material considerations that went into the mechanical design.

To provide a physical barrier between moisture and our electronic components, we plan to design a housing unit for the central hub as well as the individual sensor nodes and 3D print them using material that is suitable for wet environments. This can be quite a challenge considering that 3D printed components can appear to be air-tight, yet the extrusion process can often lead to a porous enclosure with small gaps between the printed layers. Some materials can also degrade due to constant contact with water, which is not ideal for our design since we plan for our system to be placed in a wet environment. For example, PLA (polylactic acid) filament is a usually great for initial prototypes since it provides a quickly made impact resistant mold with superior printing detail, but it tends to attract moisture which can cause the structure to soften and eventually dissolve [20].

As an alternative to these less water-resistant 3D printing filaments, we are looking to use PETT (PolyEthylene Trimethylene Terephthalate) filament for both our central housing [11]. This strong material does not absorb water moisture well and is safe to biological organisms which are both key aspects needed for our design. The major constraint we see within using this type of filament is its need for specialized equipment which is costly and difficult to work with.

Physical protection is not always the most secure method for water-proof design considering that unexpected leakages can occur. Our printed circuit boards are essential to the functionality of our system and their electrical components require secure protection from moisture artifact. We plan to apply a protective water-resistant layer onto our PCBs as an added measure of water-protection. Although, we plan to extensively test our circuits beforehand, it is desired that the coating allows for modifications to be made to our circuit after application in the event that fault behavior does occur.

We found several options for PCB protection, including a selection of conformal coatings that vary in terms of cost, ease-of-removal, and protective performance. For a robust water-resistance layer, silicone resin is commonly used in high humidity environments as it presents a solid barrier preventing moisture from coming in direct contact with the PCB [19]. This of course is at the cost of a concrete-type enclosure that is difficult to remove if repair is required. Previous research has also noted the effectiveness of cheap nail polishes in PCB water security due to the film-forming nitrocellulose polymers [30]. Similar to the silicone resin, a concrete-like effect is attained which is not entirely desired for a system prototype. Other substances, such as acrylic conformal coating, provide less water-protection but allow for easy removal. These less water-resistant options could be appropriate considering that our PCBs will also be protected by a physical enclosure. Since the physical fabrication of the project was put on hold, a future plan to determine the coating that best protects against water damage, consists of testing the waterproofing capabilities of a nail-polish application, an acrylic conformal coating, and a commercial PCB varnish. Once tested, the most protective coating along with the most suitable 3D printing material would then be used to create a secure waterproof housing for subsequent versions of the system.

2.8 User Interface for Central Hub

The goal for the user interface (UI) is to allow the user to designate sensor nodes to hose outputs, control the irrigation system's settings, and observe the sensor data all through a visually pleasing display that is easy to work with. The menu-controlled interface provides users with the ability to configure the one-hop network, actuate water delivery, monitor sensor data, and adjust the moisture threshold levels as desired.

2.8.1 Outline of User Menu

An overview of the user menu directory is shown in Figure 40. Within this flow chart, the general page layout is illustrated, where the home menu presents three available menu pages: Sensor Data, Hose Configuration and Options. Within the Sensor Data menu, the user is provided with four options for real-time sensor data. The UI can either relay individual sensor node data points or provide a recent log of a specific type of measurement from all sensors. Having direct access to the current sensor readings is convenient for our target audience of casual gardeners and homeowners.

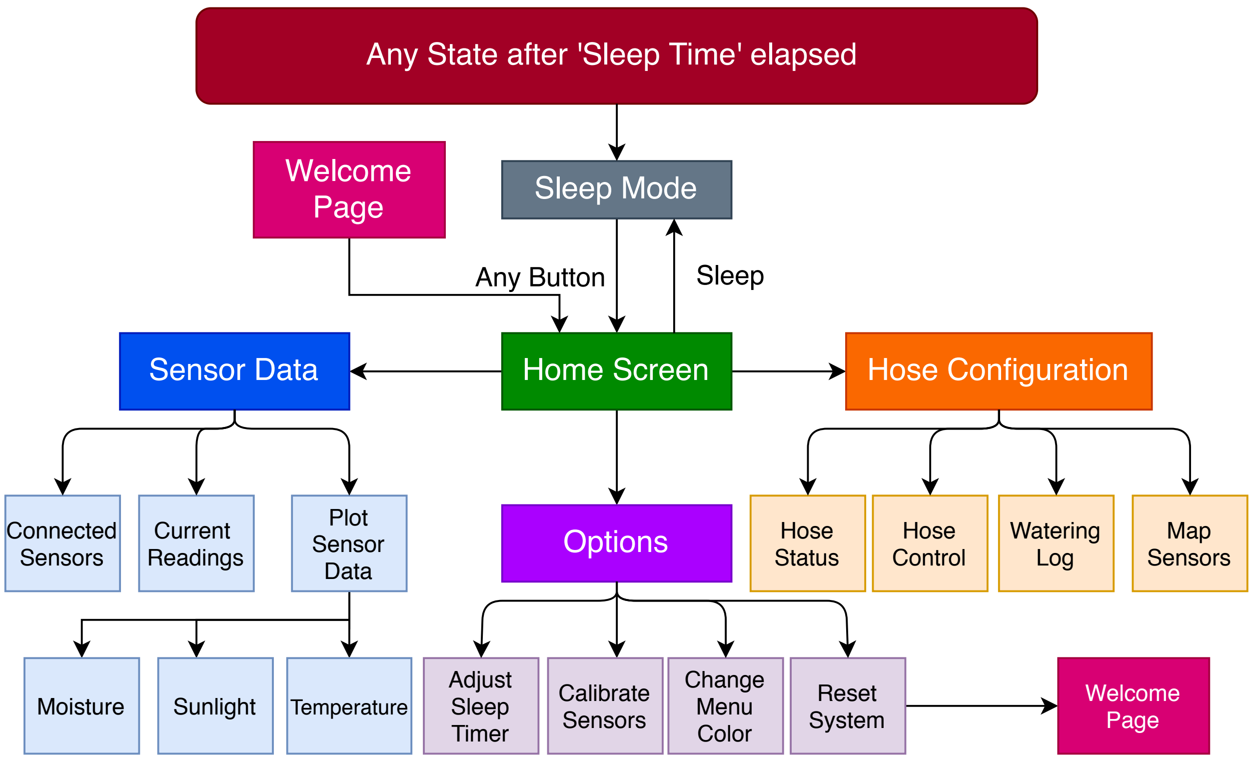


Figure 40: Flowchart describing a high-level page layout of the user interface. [13]

Within the Hose Configuration menu, the user is provided options to configure the sensor nodes to correspond with specific hose outputs. This menu also allows for adjustments to the moisture sensor thresholds which determine how dry the soil must be before watering. Further research would allow for the implementation of calibrated preset options within this menu corresponding to common plant profiles with varying soil moisture requirements. Although the system is set by default to automatic irrigation, the interface also grants the user manual control of the hoses. This utility is helpful for situations in which water is needed for other general purposes while the system is being used to monitor and control irrigation.

The Options menu presents configurable settings more focused on the actual central hub rather than the irrigation system. It provides an option that can configure and edit the central hub's sleep

[13] This figure was made with Draw.io. [4]

mode timer. When a button has not been pressed within the set time specified by the sleep timer, the state machine transitions into sleep mode which turns off the OLED screen in order to conserve energy. Meanwhile, if the user desires a full system reset to reconfigure the sensor network entirely, there is a provided Reset option to disengage the connection from the sensor nodes, essentially resetting all wireless network connections. Therefore, the central hub allows the user to control everything from changing the water-moisture thresholds to observing sensor data in real-time.

2.8.2 Assembly of User Interface

To provide a low-cost display without sacrificing clarity, we used the 1.54" RGB OLED Module acquired from WaveShare industries [5]. This module consists of a SSD1351 128 RGB x 128 Dot Matrix OLED driver which controls a model UG-2828GDEDF11 OEL display. The datasheets were obtained from the manufacturers Solomon Systech [22] and WiseChip Semiconductor Inc. [9] respectively. This display provides RGB functionality and a minimum brightness of 70 cd/m^2, which is clearly visible in most outdoor lighting conditions as seen in Figure 41.

Figure 41: OLED display providing clear visibility of the welcome page in daylight.

Display functionality is controlled through C/C++ programming that will run on the central hub SBC. In order to access the SSD1351 driver, we used the OLED_Driver, OLED_GFX, and DEV_Config libraries provided by WaveShare industries [6]. Similar to the RF radios, we had to configure the SPI bus to allow for serial communication between the OLED driver and the central hub SBC. Using the same methods described in Section 2.2, we used the BCM2835 library to access to the GPIO pins and SPI functionality within the Raspberry Pi. Connecting the OLED module to the same SPI bus as the wireless transceiver first requires physically connecting the OLED module's MISO input (master-in-slave-out) to GPIO pin 10 and the CLK (system clock) to GPIO pin 11 on the central hub. For individual control of the OLED module, the additional DC, reset, and chip select pins must be connected to GPIO pins 23, 24, and 16 respectively and enabled in software during initializing of the SPI bus. The OLED module software implementation works by taking in 16-bit unsigned integers which correspond to coordinate inputs and RGB parameters. These values are then used to update specific pixel locations with desired colors every clock cycle. The provided GFX library includes an assortment of built-in functions that make visual control of the OLED easier to work with.

The user menu interaction is accomplished through a state machine design that transitions based on physical push-button inputs. The schematic used to connect the OLED module and push-buttons to the Raspberry Pi is shown in Figure 42. The button inputs default to low, but once pressed down, the GPIO pins detect a 3.3 V digital high signal. Maximum current draw for each push-button was limited to approximately 0.17 mA with the inclusion of 20 $k\Omega$ pull up resistors in series with the button input. In order to run the OLED with event-driven programming rather than continuous sampling, the buttons inputs update the UI state machine (and therefore the OLED display) only when the input is initially transitions from low (0 V) to high (3.3 V). Debouncing is performed on each button signal to ensure that the inputs are stable before triggering a button event.

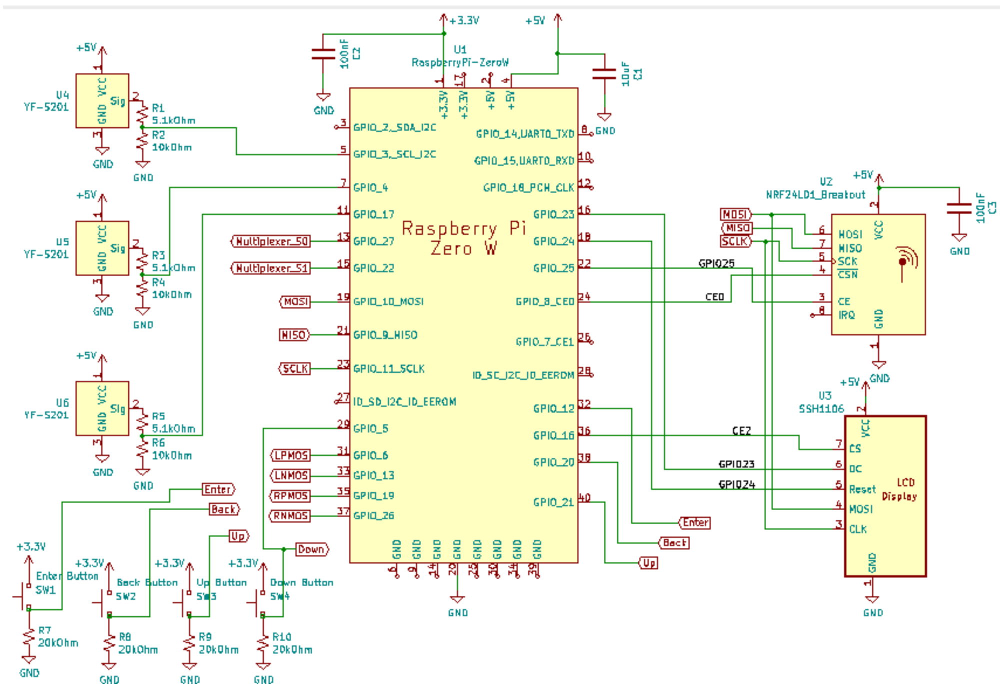
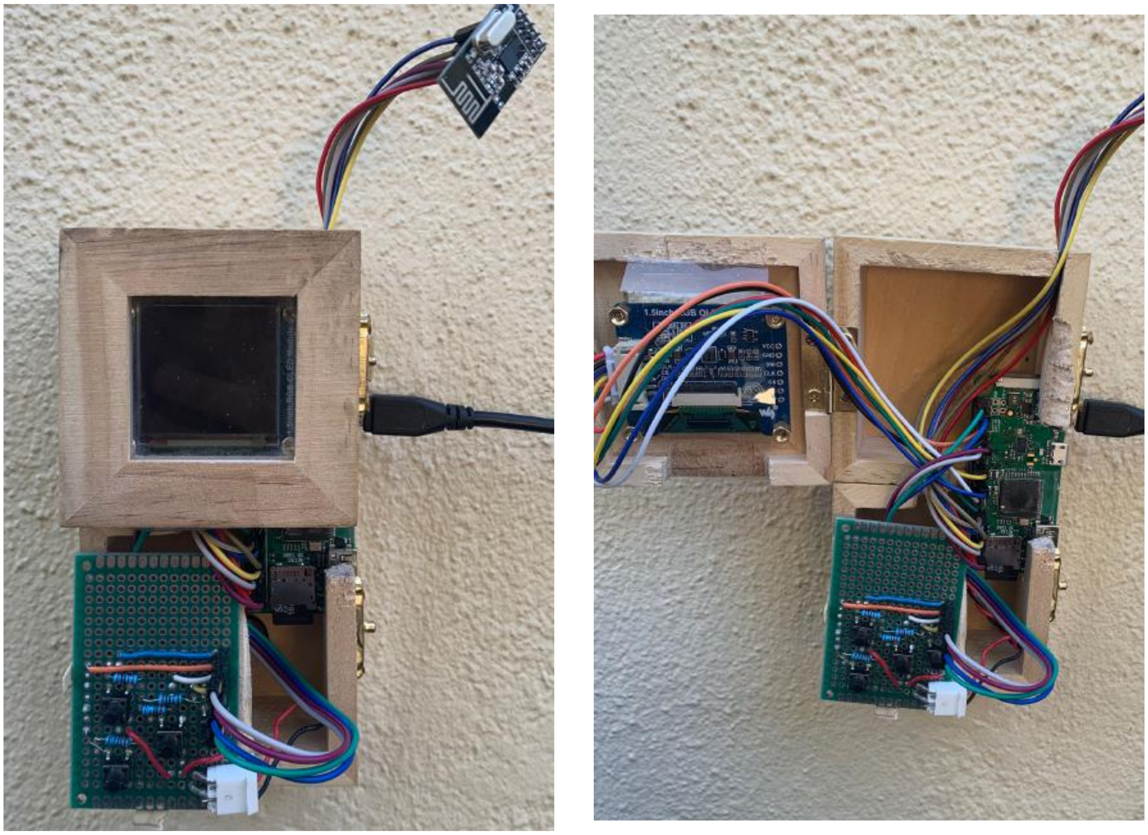


Figure 42: Full Central Hub Schematic V2 outlining connections for OLED module.[14]

As a way to encase the entire central hub into single unit, we designed a wooden assembly that can be mounted on a wall for easy installation. Slots were carved to allow for routing of the wires connected to the central hub. The button interface as well as the OLED module are both mounted on the front of the housing for user interaction with the system. Figure 43 displays an example of the mounted central hub assembly as well as the internal connections within the encasing.

[14] This figure was made with KiCAD [14]



(a) User Interface of Central Hub

(b) Internal Circuitry of Central Hub.

Figure 43: Complete Central Hub mounted on a wall

2.8.3 Data Plotting Functionality

One of the most important features of the user interface is the ability to observe stored sensor data from the OLED screen, since it provides useful information regarding plant and soil conditions. As a way to display many sensor readings in a presentable manner, a plotting function was implemented within the Sensor Data menu pages.

Plotting of the data is accomplished by converting a collection of sensor data points into pixel locations to be set on the OLED module. Using this method, we are able to traverse through arrays containing sensor data structs and plot the past 100 readings pertaining to a specific node ID and data type (soil moisture, sunlight, etc.). The process of traversing through the data struct array and plotting relevant sensor data onto the OLED module is outlined in the flowchart shown in Figure 44.

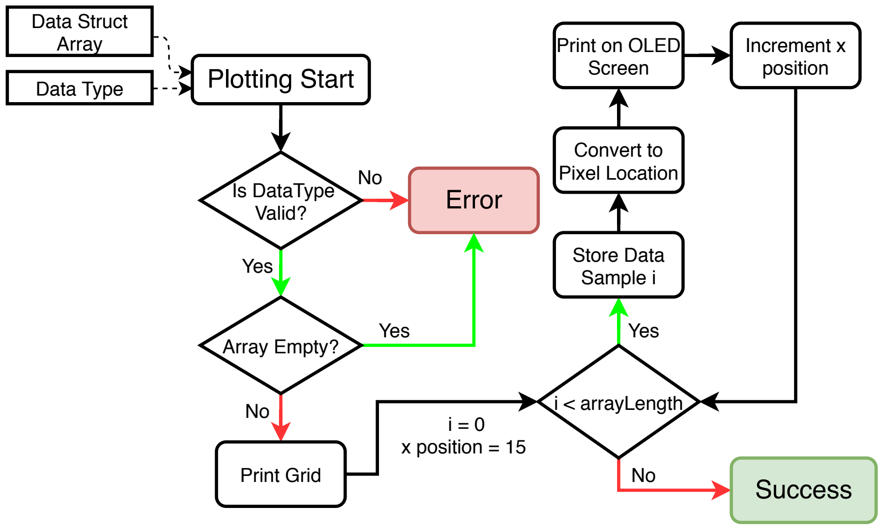


Figure 44: Flow chart outlining user interface plotting function.

Within this plotting function, an array of data structs and a desired data type are passed in as inputs. The array is used to determine which set of data points to print corresponding to a specific sensor node ID. Stored as an enum, the data type input (either moisture, sunlight, or temperature) determines which element in the struct to use for plotting. Once the inputs are checked to be valid, the array index and x-coordinate variable used for printing are both initialized. Since the value of the OLED y-axis is limited to 120 and the top of the screen represents a y-coordinate value of 0, the sensor data output is converted into a printable value for the OLED screen. Mapping sensor data values which range from 0 to 100, is accomplished by subtracted from the y-position value corresponding to the bottom of the grid where the new mapped value can be defined as

$$\text{Mapped Y} = 120 - (\text{Sensor Data Value}) \tag{10}$$

After plotting the data point at the mapped x and y coordinates, the function iteratively increments the x coordinate and repeats the process until all data within the array corresponding the desired data type is plotted. An example of the plotting functionality working on the OLED module to display water moisture data is shown in Figure 45.

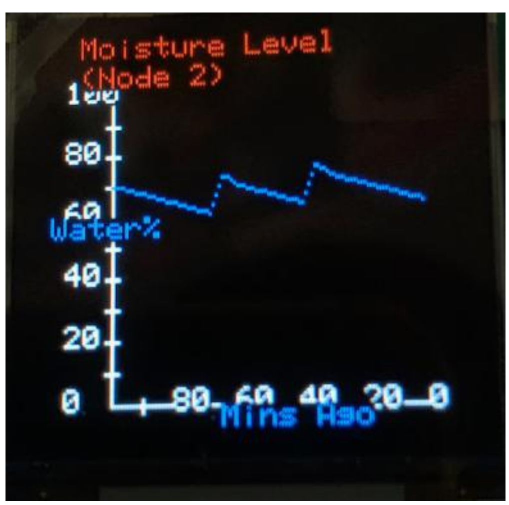


Figure 45: Example of moisture plot displayed on user interface for Sensor Node 2.

2.9 Central Hub Power Distribution

Our design of the central hub requires its power supply to enable continuous operation of the Raspberry Pi Zero W single-board computer and all the peripherals simultaneously. It also needs to supply enough current to control four, parallel latching solenoid valves for the distribution of water. For this reason, power will be provided through a standard 5V wall adapter with a maximum current output of 1 A. As determined by the power budget analysis, shown in Table 5, we found that this standard switching adapter would be sufficient in powering our single-board computer and supplying enough current to drive all of the desired components through the Raspberry Pi 5V rail.

Watering - Open one valve, close another			
Component:	**Current (mA) @ 5V:**	**Supply Voltage (V):**	**Power (mW):**
Raspberry Pi	200	5	1000
4x Pushbuttons	1.03	5	5.15
OLED Display Screen	3.1	5	15.5
NRF24L01 Transceiver	11	3.3	16.5
Solenoid Valve (Opening)	289	5	1445
Solenoid Valve (Closing)	272	5	1360
Total	776.13	5	3842.15
Not Watering			
Component:	**Current (mA) @ 5V:**	**Supply Voltage (V):**	**Power (mW):**
Raspberry Pi	200	5	1000
4x Pushbuttons	1.03	5	5.15
OLED Display Screen	3.1	5	15.5
NRF24L01 Transceiver	11	3.3	36.3
Solenoid Valve (Opening)	0	5	0
Solenoid Valve (Closing)	0	5	0
Total	215.13	5	1056.95

Table 5: Central Hub Power Distribution Information. All currents are normalized to 5V. NRF24L01 Transceiver includes 5V to 3.3V step-down conversion loss.

When evaluating the Central Hub's power characteristics in Table 5, it is important to note that the watering algorithm only actuates one solenoid at a time, as explained in Section 2.7.3. Additionally, once a solenoid is opened or closed, it does not consume any current to stay in its current state. The latching solenoid is actuated by sending a 100 ms ±5 V square wave pulse across the valve where the polarity determines whether the latch will open or close. Since the opening and closing pulses consume roughly 289 mA and 272 mA respectively, the adapter can easily supply these current demands on top of the rest of the system.

2.10 Sensor Node Power Distribution

The sensor nodes' power demands come from the Arduino Nano microcontroller, the nRF24L01+ radio transceiver, and the sensor array. The load characterization for the sensor nodes is shown in Table 6, showing all components and their respective power demand under the microcontroller's sleep and full power states. Regarding the two sensor states, the first and most common is the sleep mode, where the sensor nodes draw about 8 mA. But once every 64 seconds, the nodes will transition to their sampling and messaging state, which consumes 33 mA and will stay there for about 500 ms. This means the sensor nodes spend about 99.2% percent of their time in sleep mode, making their time-weighted average current demand marginally higher than that of their sleep mode at 8.53 mA.

Sampling and Messaging State			
Component:	**Current (mA) @ 5V:**	**Supply Voltage (V):**	**Power (mW):**
Arduino Nano	21	5	105
NRF24L01 Transceiver	11	3.3	55
Sensor Array	0.9	5	4.5
Total	32.9	5	164.5
Sleep Mode State			
Component:	**Current (mA) @ 5V:**	**Supply Voltage (V):**	**Power (mW):**
Arduino Nano	8	5	40
NRF24L01 Transceiver	0.03	3.3	0.13
Sensor Array	0.01	5	0.05
Total	8.04	5	40.2
Time-Weighted Average			
Component:	**Current (mA) @ 5V:**	**Supply Voltage (V):**	**Power (mW):**
Total	8.53	5	42.63

Table 6: Sensor Node Power Distribution Information. nRF24L01+ Transceiver includes 5V to 3.3V step-down conversion loss.

2.10.1 Battery Considerations

Each Sensor Node runs on a 3.7 V, 10 Ah lithium-ion polymer battery. The first consideration in battery management is the depth of discharge (DoD), which is the percentage of the capacity drained from a battery at full charge. The key trade-off in choosing the proper DoD comes down to balancing battery life and capacity retention. The chart in Figure 46 shows capacity retention over the battery's lifetime across different states of charge (SoC) bandwidths, where SoC is the level of charge of a battery relative to its capacity. Using the average load current from Table 6 in Section 2.10 and the nominal battery voltage of 3.7, a full discharge of this battery nets an ideal battery life of 36.2 days. This does not factor in supply efficiency or the battery management characteristics, but will be used to estimate the time between discharge cycles and better interpret this capacity retention curve in the context of time. To give plenty of headroom, we assume our battery will be recharged every 10 days, less than a third of that ideal lifetime. Using 10 days as our average time for a single discharge and recharge cycle, this means 500 discharge cycles occurs at approximately 5,000 days, or 13.7 years.

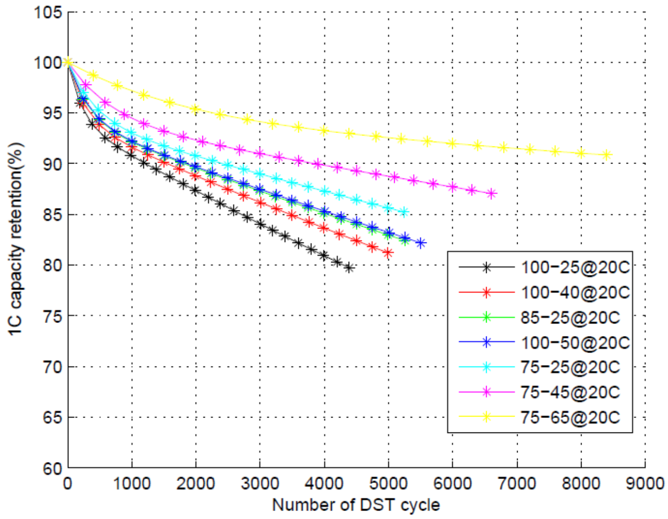


Figure 46: Capacity retention curves for lithium-ion polymer batteries at various SoC Bandwidths [1].

In general, we see that with smaller discharge bandwidths comes improved capacity retention. This, of course, comes at the cost of battery life in a single discharge. The 100-25% SoC (or 0-75% DoD) in black nets about a 93% capacity retention after 500 cycles. A full battery discharge of 100-0% SoC is not shown in this plot, but nets about an 84% capacity retention under the same conditions [29]. Note that by cutting to a 100-50% SoC (or 0-50% DoD) shown in blue, we take a significant cut into battery life with minimal improvements in capacity retention. Using known capacity retention rates, we chose the 0-75% DoD since it was the best of balance lifetime and capacity retention.

It is important to address that when charging the battery under its full capacity, such as the 85-25% SoC curve in green, the capacity retention sees a meaningful boost while draining a larger proportion of the battery's capacity than the 100-50% SoC in blue. This characteristic needs to be explored for larger discharge bandwidths, such as our chosen 75% bandwidth, and compare its capacity retention to that of 0-75% DoD before confirming its benefits and subsequently adding this to the design. With a 0-75% DoD and a peak C-rate well below 0.2C, we project our battery voltage to follow the orange curve for a 0.2 C-rate, shown in Figure 47. Under these conditions, the battery voltage will swing from 4.2 Volts at full charge to 3.6 Volts at 75% DoD.

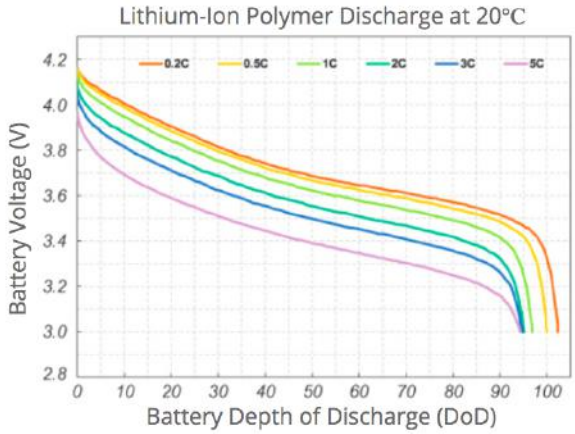


Figure 47: Discharge graph dependent on C-rate for lithium-ion polymer batteries[3].

2.10.2 Sensor Node Power Supply

To take our battery output and convert this to a fixed 5 V output, the sensor node power supply, shown in Figure 48 uses the TPS61222 boost converter, which is designed for low current applications with lithium-ion polymer batteries. The voltage divider (R1 and R2) allows for the microcontroller to monitor when the battery voltage goes below 3.6 V to enter deep sleep, and alert the user to charge the sensor node's battery. With just 7.5 μA of quiescent current, the boost converter can reach up to 95% efficiency in ideal conditions [13]. While many power supplies will taut their high efficiency, the datasheet often will not specify the exact conditions under which this high efficiency holds. For this reason, it is paramount that high efficiency holds at both the load current conditions and the battery input voltages.

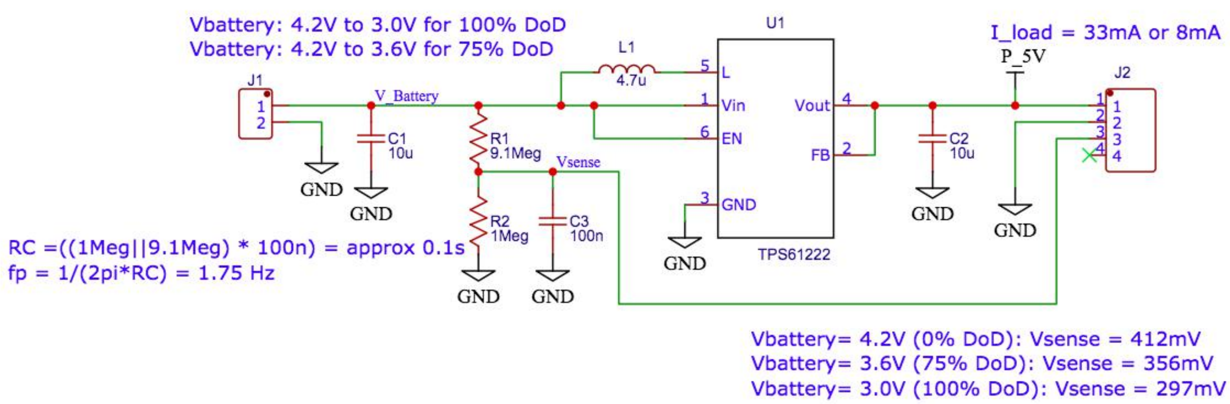


Figure 48: Sensor node power supply schematic version implementing the TPS61222 boost converter. [15]

To test the boost converter's efficiency rating at the battery input voltages and load currents, the power supply's output was connected to a resistive load corresponding to the sensor node load currents of 8mA and 33mA (taken from Table 6). Within these load currents, the input voltage to

[15] This figure was made with EasyEDA [14]

the power supply is swept in increments of 100 mV across the battery voltages for a 0-75% depth of discharge, 4.2 to 3.6 Volts. At each data point, efficiency is calculated using voltage and current measurements at both the input and output. The results are shown in Table 7, where all conditions have at least a 90% efficiency.

TPS61222 Efficiency Results		
Input Voltage:	**Efficiency for 8 mA load current:**	**Efficiency for 33 mA load current:**
4.2	94.0%	92.1%
4.1	93.7%	91.6%
4.0	93.6%	91.3%
3.9	93.6%	90.9%
3.8	92.9%	90.4%
3.7	92.4%	90.3%
3.6	92.1%	90.1%

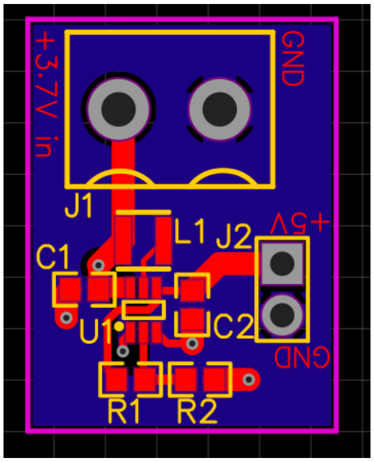
Table 7: TPS61222 Power Supply Efficiency for sleep mode (8 mA) and Sampling and Messaging mode (33 mA).

Using the lowest efficiency rating (measured at 3.6 V) and time weighting the 8 mA and 33 mA load currents based on expected duty cycles discussed in Section 2.10, the power supply's average efficiency was measured to be 92%. Taking this average efficiency and a 0-75% depth of discharge, we project a battery life of over 3 weeks, so long as the battery stays around 20° C, shown in Equation 11.

$$\begin{aligned}&\text{Battery Life} \approx \left(\frac{\text{Battery Energy}}{\text{Average Power Demand}}\right)(\text{Supply Efficiency})(\text{Discharge Bandwidth})\\ &\text{Battery Life} \approx \left(\frac{(3.6\,V)\,(10000\,mAh)}{(5\,V)\,(8.53\,mA)}\right)(92\%)\,(75\%) = 582.4 \text{ hours} = 24.3 \text{ days}\end{aligned} \tag{11}$$

With a long battery life, this requires minimal maintenance recharging the sensor nodes, only requiring the user to do so periodically. Additionally, this lifetime holds over several years; by factoring in our depth of discharge and its affect on capacity retention (Section 2.10.1), this will put the sensor node's battery life at about 93% of its original capacity at 500 discharge cycles, which corresponds to a single 0-75% DoD cycle of 22 days.

As for the power supply layout and assembly (Figure 49a and Figure 49b respectively), we see this design keeps the footprint small at just 0.6 by 0.8 inches, which is great for a seamless application on the sensor nodes.



(a) Sensor node Power supply layout.

(b) Sensor node power supply physical assembly.

Figure 49: The sensor node power supply converts 3.7V to 5V at a minimum of 90% efficiency.

2.11 Parts List and Budget

The purpose of this final sub-section is to list all the parts and give a final estimate for the budget. The parts list (see Table 8) is broken down by the sub-sections in Section 2 with the components listed by unit price and quantity needed. Although we focused on an inexpensive design with suitable performance, improving this system with higher grade parts is an available option if budgeting is not a concern.

Item	Cost/Item	Quantity	Total Cost
Microcontrollers			
Arduino Nano	$4.66	3	$13.98
Raspberry Pi Zero W	$24.50	1	$24.50
Wireless Communication			
HiLetgo NRF24L01+	$1.97	3	$5.91
Sensor Array			
Photoresistor 5 mm	$0.15	30	$4.65
BMP180 GY-68 Digital BPS	$5.39	3	$16.17
Galvanized Steel Probes	$0.09	100	$8.59
DMP22110UW-7	$0.44	3	$1.36
Printed Circuit Boards	$2.00	5	$10.00
Water Delivery			
Drip Irrigation Kit	$14.87	1	$14.87
1/2" Latching Solenoid Valve	$17.99	3	$53.97
FL-608 Liquid Flow Sensor	$9.95	3	$19.85
Printed Circuit Boards	$2.00	5	$10.00
User Interface			
1.5inch RGB OLED Module	$19.35	1	$19.35
Pushbuttons	$0.36	4	$1.44
Power			
5Ah rechargable Li-Ion Polymer	$14.19	3	$42.57
TPS61222 5V Boost Converter	$0.17	25	$4.25
Printed Circuit Boards	$2.00	5	$10.00
TOTAL			**$261.46**

Table 8: This table outlines the required parts for the project as well as price estimates based on our purchases.

3 Testing and Results - Full System

Testing and validation of the Intuitive Auto-Irrigation project was carried out in Spring 2020 through two distinct phases to ensure our system is functional and to resolve any critical errors. The first testing phase verified that the wireless network accurately reported the sensor data and stored that data on the master node using an incomplete subset of the full irrigation system. The second testing phase, on the other hand, implemented the full irrigation system and validated that all individual component blocks properly worked together. Having multiple phases of incremental testing allowed us to verify proper functioning of individual system blocks prior to testing the full integrated system. This made debugging easier and allowed testing to begin earlier while we continued to improve other components throughout the duration of the testing phases. Specifically looking at the first testing phase, all tested components had specific failure indicators, making it easy to test all of these components as an initial prototype system. This first testing phase did not implement water delivery, instead we took measurements with the sensor nodes and recorded the data on the central hub. We effectively simulated water delivery by logging indicators variables to the output text file that clearly showed when water delivery would be triggered. Performing these sanity checks earlier lowered the chance of undesired consequences arising from water delivery related malfunctions later on in the second testing phase.

3.1 Testing Phase 1: Prototype Testing

Throughout the initial prototype testing phase, we gathered a great deal of data from the soil moisture sensor and ambient light sensor - which are the two most important sensors in the array. The results of the data are plotted in Figure 50, Figure 51, and Figure 52. These tests were performed at Grant's house using two potted plants in his backyard. The first test began at 5:30pm and lasted roughly 3.5 hours (shown in Figure 50). The data shows two trends, one for each sensor node, showing the logged gravimetric water content and the ambient light level. Regarding the soil moisture level, we identify a trend that shows the water content slowly drop over the duration of the test. This is what we expect; without water delivery or rain, there is no way for the soil to replenish its moisture levels. As for the ambient light level, we see an interesting trend due to the fact that the test was started when there was full sun and lasted until it was totally dark. The light sensor was able to distinctly identify between night and day and respond to sharp changes in light level correlating to sunlight peeking through clouds or trees. Overall, the data shows that both the soil moisture content sensor and light level sensor can accurately and reliably record soil moisture content and light level respectively.

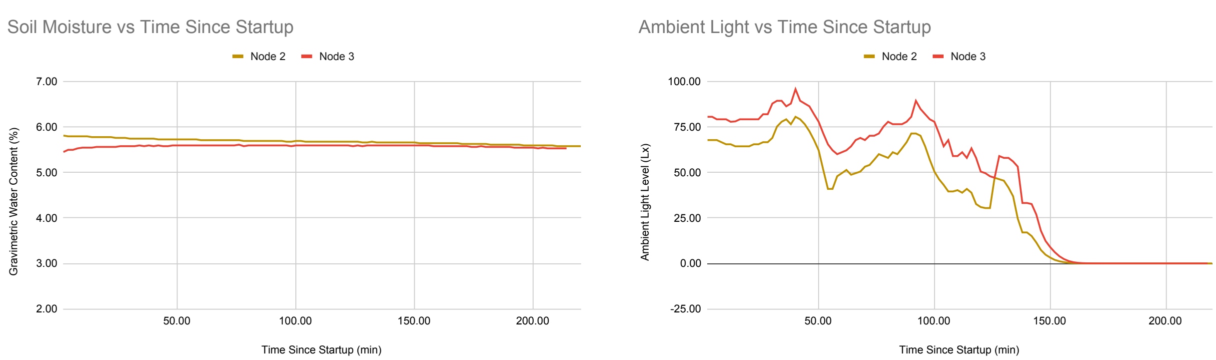


Figure 50: Soil moisture and light level data from 2 nodes during test 1 of testing phase 1.

The second test followed the same principles as the first test, but the starting time was changed to 10:00am (shown in Figure 51). This allowed the test to run throughout the day and resulted in new data trends. The soil moisture content follows a similar pattern to the first trial but curiously begins to rise roughly 2.5 hours into the test. While only a minimal change of about 0.15%, this trend is a bit unusual because it should not be possible for the water level in the soil to increase without directly adding water into the system. We know this is not an error with the soil moisture sensor readings because both sensors spike simultaneously even though they reside in separate soil containers. While not impossible for both sensors to report faulty sensor data at the exact same time, it is much more likely that the soil moisture content increased due to some other phenomenon. Initially confused by the data, we performed more tests to obtain a better understanding of the slight rise in water levels in response to an increase in sunlight.

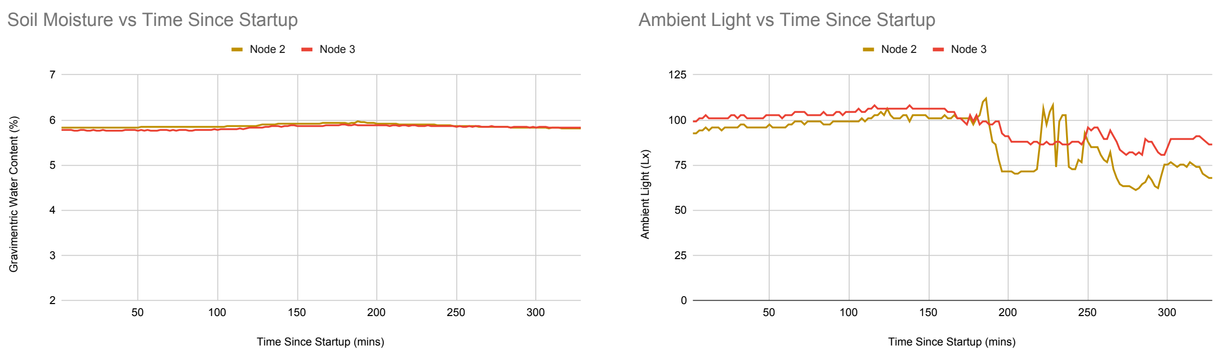


Figure 51: Soil moisture and light level data from 2 nodes during test 2 of testing phase 1.

Regarding the ambient light level data, we see a slowly increasing slope for both sensor nodes within the first half of the trial and then varied in the second half of data. We believe that sharp variation had to do with both the formation of the clouds throughout the day as well as shading (eg. trees, the house) due to the location of the sun relative to the light sensor. Although both plants were in the same general area, they experienced varying amounts of sunlight throughout the day. The high sensitivity of the light sensors were able to identify direct sunlight versus cloud cover and even when drifting in and out of the shade. The proven higher sensitivity allows us to design a more nuanced algorithm with precise, user-defined thresholds for ambient light levels.

The final test in the first phase of testing lasted from 3:00pm to 11:00am the following day (shown in Figure 52). Test number three only used a single node because we ran out of 9V batteries and only had a single Lithium polymer battery at the time. Regardless, the test follows previous trends for the soil's water content: descending overnight as water is not added to the system. However, as morning comes around, we are able to identify the same pattern as seen in the first test, Figure 50. The soil moisture again rises in the morning when seemingly no water is added to the system. The data recorded in this test proves that our sensor nodes did not exhibit faulty behavior, rather there is some other phenomenon occurring that leads to to a rise in soil moisture content in the morning. Without much to go on, our leading hypothesis at this point was that the elevation of water in the soil changed with the temperature of soil. This is supported by our light level data because we see that the soil moisture only begins to rise in the morning and continues to do so throughout the day. However, without additional data, this hypothesis would remain unproven until later.

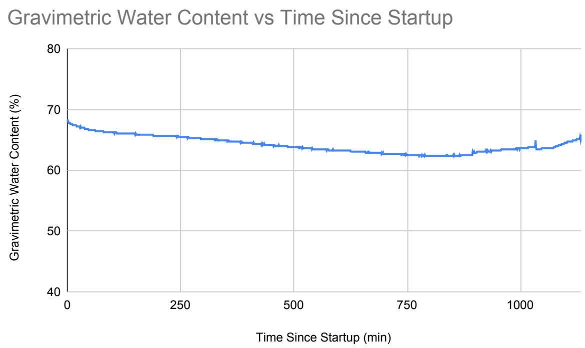


Figure 52: Soil moisture and light level data from 1 node during test 3 of testing phase 1.

The light level data also resembles previous tests, where we see a sharp difference between the day and night. The data in this test is also much more consistent than previous tests yet still maintains a high level of sensitivity (seen as the fluctuating light level at night). Basically, by applying a digital filter to remove much of this noise, we can make the light sensor a more precise instrument.

The main takeaway from this first testing phase is that the sensor nodes are able to record consistent sensor data and transmit it to the central hub where that data is logged to a csv file. This successful process verified that our first testing phase worked as intended, so we moved on to implement the next phase of testing.

3.2 Testing Phase 2: Validation Testing

Upon completing the first testing phase, we fixed several bugs and implemented all system components for phase 2. This meant adding in the user interface and water delivery systems to test a near-complete version of the product. This second phase of testing was expected to take about two weeks longer than the first phase due to the added complexity of the system and the extra data we wanted to capture, but because of weather complications, we had to shorten the test substantially. While obviously undesirable, the inability to create water-tight enclosures for the sensor nodes (due to a lack of access to a 3D printer) meant we had to restart the test after a couple days. The rain we experienced also caused damage to the sensor nodes, stripping two more days of testing in order to fix the problems stemming from water-related short circuits. By the time we fixed the rain-related issues, we only had 5 days left to test, which is shorter than we would have liked.

Validating the full integration of all system-level components is essential to the completion of our project. The primary aspect we were looking to validate was whether the system worked properly based on visual inspections of the plants and by observation of the data logged from the sensor nodes. Essentially, we wanted to validate that the system works through data and through qualitative analysis. Specifically for the data validation, we looked to record enough data to cross-check the expected water delivery outputs, based on sensor measurements, with the observed outputs of the hose signals to verify that the system could accurately trigger water distribution when the appropriate conditions were met. The bottom line is that we succeeded in meeting that criteria; the data, discussion, and final visual inspection of the plants in this section will show how we came to that conclusion.

The test for phase 2 lasted a total of 5 days; due to limitations in the fabrication process and battery accessibility, we were limited to two nodes. We were hoping to test more, however, this still provided plenty of data to help with validating the system. The data for the test is shown in Figure 53, where the data is plotted over time to show a time-lapse of the entire test. The data for this test was super-imposed to give a better sense of how the soil moisture content and light level correlated with each other. For reference, the soil moisture content is plotted on y-axis 1 (on the left) and the light level is plotted on y-axis 2 (on the right).

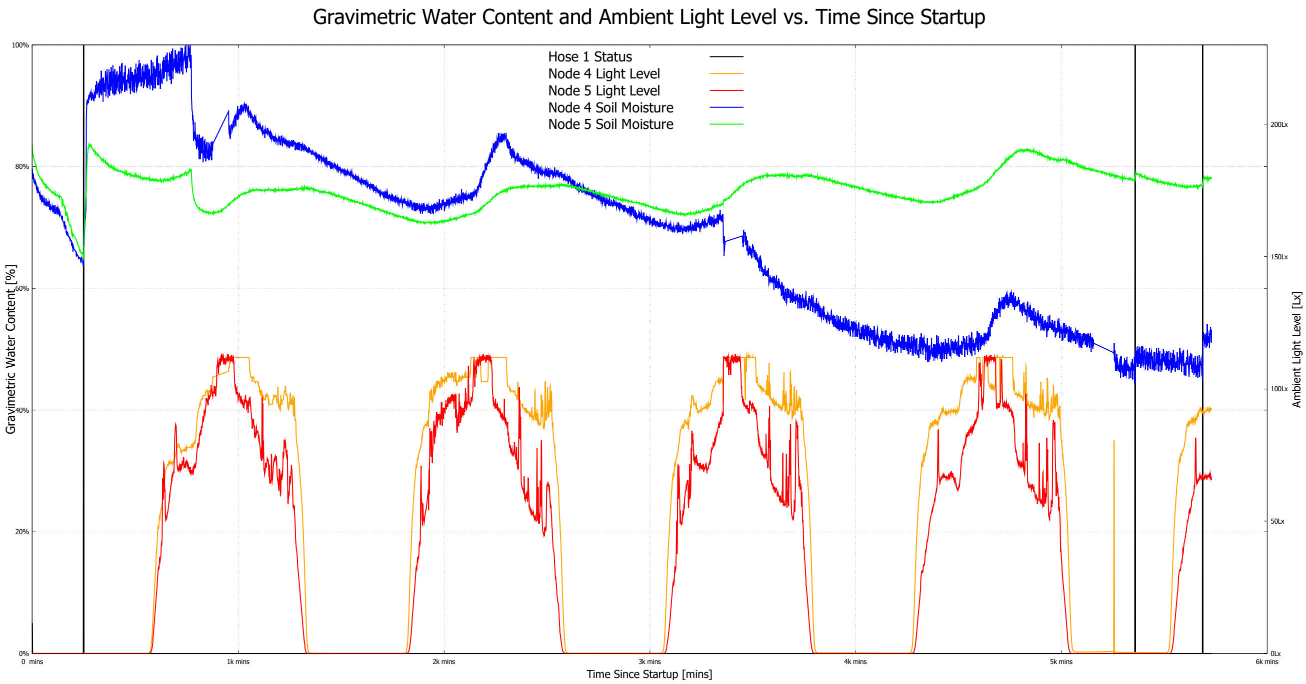


Figure 53: Phase 2 validation testing results shows sensor data from the final stage of testing.

In Figure 53, the solid, vertical black lines indicate when the hose output was triggered while the blue and green lines represent the soil moisture content of sensor node 4 and sensor node 5 respectively, the yellow and red lines show the light level data again from nodes 4 and 5 respectively.

Looking at when the hose triggered (ie. when watering occurred), we can see that it largely depends on the soil moisture content. The first time the hose triggered, indicated by the furthest black line on the left, occurred when both sensor nodes reported a soil moisture level below 65% and the light level was low. Since we set the soil moisture threshold to 65% based on empirical data from prior tests, if we refer back to Figure 12, the sensor nodes are supposed to report that they need water when the soil moisture level is below its standard threshold and when light level is low. Therefore, this black line shows the sensor nodes sent a digital high signal to the master node indicating they needed water when soil moisture levels dropped below the preset threshold. As soon as both nodes reported that they needed water, the central hub triggered water delivery, after verifying that no rain was predicted in the next 36 hours. This aligned with our expected results as we see that the soil moisture levels shoot up after the water delivery was triggered. The reason the soil moisture content rises so fast is because the water pressure in the hose was set too high, this was promptly lowered for the remainder of the test.

The test progressed after the initial water delivery as it waited a couple days for the water level to drop again. The next two times the water delivery process was triggered, we see it happen for a different reason than the first black line. With the critical soil moisture level threshold set to 45%, if either of the two nodes has a water level below 45%, the hose will immediately turn on. This is shown when node 4 dropped below 45% twice in relatively rapid succession. The result is the hose turned on for short periods of time to ensure that the water-deprived plant got enough water to prevent dehydration. With the critical soil moisture threshold, this process happens regardless of the light level which is why we see the water delivery process also trigger during the day for the last black line.

One specific thing to notice here with regards to the soil moisture levels is how varied the recordings for the two sensor nodes were. While assumptions may lead to a conclusion that the soil moisture sensor was inaccurate, the light level data here shows why the two nodes had such different soil values. During the day, the light level of node 4 (yellow) was almost always higher than that of node 5 (red) implying that node 4 received more sun than node 5. This further implies that the soil for node 4 should be drier than the soil for node 5 because increased temperature directly correlates with higher evaporation rates. The data supports this notion too; we see that soil moisture content for node 4 dropped much more than node 5 confirming our suspicions. Therefore, the soil moisture data results are not attributed to variability in the sensors themselves, but rather as a result of the different environmental conditions each node is exposed to.

Another trend we see in the soil moisture content is the water level rising each morning when the sun comes out. Each time the light level started to rise, the soil moisture content followed shortly after. This is the same trend we saw in the prototype testing phase which we attributed to heat causing the water in the soil to rise and fall throughout the day. Previously we did not have enough data to reasonably verify this was the case, but now we have enough samples to show a continuous trend. We can confidently say that the measured soil moisture directly correlates with the light absorbed by the soil, resulting in the sinusoidal pattern evidently displayed in node 5's soil moisture content (the green line).

With data that confirms the working Intuitive Auto-Irrigation system, and the fact that the plants looked healthy after the completion of the test (see Figure 54 for the end result of the plants), we effectively completed a prototype design of an active sensor-based automatic irrigation system. Each phase of testing was integral to this conclusion as each test builds upon the previous one, until our final test where we verified and validated that the system works as intended to lower water consumption without sacrificing plant health.

Figure 54: The final visual inspection of the plants after testing showing that they are indeed, not dead.

4 Conclusion

After analyzing all subsystems for the project and integrating them within the full sensor-based irrigation system prototype, we conclude that our design goals were met; we successfully created a system which can autonomously monitor and control plant irrigation without sacrificing plant health. Although there are still ways to improve our system, the wireless functionality, water delivery actuation, data storage, and user-interface capabilities are all fully functional. Regarding the wireless network, we surpassed the required distance of 400 feet per network layer by about 50 feet while still being able to maintain reliable communication. As for the sensor array, we verified our soil moisture readings to be precisely correlated with the soil's gravimetric water content for a fixed soil type. The sensor nodes also displayed appropriate levels of energy efficiency as the internal microcontroller was in a low-power state for more than 99% of the operating time and only switched on when sampling and transmitting sensor data to the central hub. Over the course of our testing period, we noticed that the soil parameters, both moisture and ambient light levels, effectively determine when irrigation is needed and trigger water delivery at the central hub. Therefore, by adjusting the relevant parameters to specific gardening setups, we believe that our fully autonomous irrigation system can benefit casual gardeners who are looking to save both time and natural resources.

4.1 Lessons Learned

Throughout the design process, we came across many situations where our original design malfunctioned or we ran into unexpected issues. These situations forced us to learn from our mistakes and debug accordingly to ensure we were able to complete our deliverables on time.

The first significant hurdle of our project was the inability to use an Arduino Uno as the central hub of the system; the amount of memory on the microcontroller was insufficient for the number of software libraries required. The Arduino Uno was also limited in its built-in utility, requiring external peripheral and WiFi modules to provide SD card functionality and access to internet connection, both of which were components that had both compatibility and reproducibility issues. These reasons pushed us to transfer our central hub from the Arduino Uno to a Raspberry Pi SBC.

Another important discovery we made during the implementation process was the inefficiency of continuous solenoid valves to control water delivery. Because of the importance placed on energy efficiency, operating multiple continuous solenoid valves would not work because they require substantial energy when driven for prolonged periods of time. The design change became apparent alongside the calculation of the power budget which revealed a massive reduction in energy consumption with the application of the significantly more efficient latching solenoid valves.

When working on the project's software implementation, team members with Apple computers encountered issues connecting to certain knockoff Arduino modules through the UART bus because of the lack of backwards compatibility with outdated bootloaders. Newer Arduino Nanos and Unos had no such issue connecting through USB, highlighting the importance of correct component selection.

Throughout the project, we learned the sheer importance of documentation and its service in recording essential engineering information such as circuit schematics, block diagrams, and flowcharts. After being more meticulous with our system documentation, the amount of time spent debugging fell considerably. Working on this project ultimately taught us how to properly manage an extensive system through consistently, well-documented subsystems, which is vital in the field of engineering where collaboration is expected.

4.2 Future Work

If allotted more time to work on this project, there are several improvements and design alterations that we would make to the irrigation system. These changes involve either optimizing the system for a broader application or continuing with the small scale model, which would involve making improvements and adding further scalability. Within this section, we focused on a few specific adjustments to improve this system for residential or small-scale gardens if given a few more months of prototyping.

Although our sensor nodes communicate effectively, more efficient and compact wireless modules are readily available which would enhance the wireless communications aspect of our system. The Particle Argon, for example, is a development board that acts as an all-in-one transceiver with built-in 2.4 GHz radio transmission, WiFi, and low-power Bluetooth capability [16]. Using the Particle Argon in place of the Arduino Nanos and nRF24L01+ radio transceivers would eliminate unnecessary complexity by removing extra peripherals by integrating the microcontroller and transceiver. The other added benefit is the additional Bluetooth capabilities that would allow users to connect to the system through their mobile devices. Interfacing our system with a mobile application would allow the users to check on the system's status and easily control it remotely from their cellular devices.

Throughout our design process, we noted that the battery life of the sensor nodes was a significant issue for long term usage of our irrigation system and would need to be improved in future revisions. The addition of solar cells to the sensor nodes' power supply as a supplementary energy source addresses battery life concerns by charging the battery and providing power to the sensor nodes. This modification would drastically extend the time we operate our sensor nodes if efficient solar power generation is achieved. The main issue with adding the solar panels is their large physical footprint relative to the sensor nodes. The large area of the solar cells could affect the growth of the surrounding plants as well as complicate both the mechanical fabrication and light sensor implementation. Solving these issues requires careful consideration of the placement of the sensor nodes since the location and orientation of the solar panel is crucial to its effectiveness. If implemented correctly, this addition could prove to be beneficial for potential users. Since the sensor nodes are designed to be left outdoors, the solar panels would generate a constant stream of energy given adequate exposure to sunlight. Using these solar cells would also align with the goal of our system being environmentally friendly.

As our project neared conclusion, we noted smaller additions that could be made assuming we were allotted three more months. With regards to the user interface, this includes a timer-based irrigation option and customization of all system parameters (sensor thresholds, forecast data, database control). Revisions to the Level-Shifting Multi-Driver PCB meant fixing the grounding plane to avoid ground splitting and creating a finalized board that encompasses all relevant central hub components and peripherals pins. Further soil testing would look to quantify the effects of soil compactness and composition on the conductive moisture readings. With this testing data, user calibration could be improved with preset soil profiles that account for different readings in different types of soil. This would also need to be accompanied by extensive research on the electrical properties of unique soil types.

Although there are many other changes we could make for future iterations of this project, those mentioned above are the most notable features that can be feasibly applied to our system and provide substantial benefits to the users.

5 Acknowledgements and References

We would like to acknowledge Professors Tela Favoloro and Stephen Petersen for their invaluable help throughout the course of this project. We also want to thank Sushmita Joardar and Andrea David, graduate students at UC Santa Cruz, for their assistance and mentorship during the design process.

We want to throw a special thanks to Russel Evans and BSOE BELS team for ensuring safety and supplying our team with all of the components, parts, and equipment we need. Without their support, this project would be stuck in the ground; we would be unable to plant any seeds without it.

Furthermore, we would like to express our gratitude to Kresge College and the Kresge Garden Co-op for funding the project and providing a testing space. This project would not have been possible without their financial support and cooperation. We deeply appreciate it.

References

[1] Accubattery. *Capacity Retention Curve*. URL: https://accubattery.zendesk.com/hc/en-us/articles/360016286793-Re-Modeling-of-Lithium-Ion-Battery-Degradation-for-Cell-Life-Assessment (visited on 05/26/2020).

[2] Environmental Protection Agency. *How We Use Water*. URL: https://www.epa.gov/watersense/how-we-use-water (visited on 03/03/2020).

[3] LiPol Battery. *Battery Discharge Curve*. URL: https://www.lipobattery.us/high-rate-lithium-ion-battery-15c-2/ (visited on 02/14/2020).

[4] Diagrams.net. *Draw.io*. URL: app.diagrams.net (visited on 03/19/2020).

[5] Waveshare Electronics. *1.5inch RGB OLED Module*. URL: https://www.waveshare.com/wiki/1.5inch_RGB_OLED_Module (visited on 02/27/2020).

[6] Waveshare Electronics. *File:1.5inch RGB OLED Module Code.7z*. URL: https://www.waveshare.com/wiki/File:1.5inch_RGB_OLED_Module_Code.7z (visited on 02/28/2020).

[7] Y. Kim R. G. Evans and W. M. Iversen. "Automated Irrigation System Using a Wireless Sensor Network and GPRS Module." In: *IEEE Transactions on Instrumentation and Measurement* 64 (1 Jan. 2014), pp. 166–176. URL: https://www.researchgate.net/publication/260303884_Automated_Irrigation_System_Using_a_Wireless_Sensor_Network_and_GPRS_Module.

[8] Y. Kim R. G. Evans and W. M. Iversen. "Remote Sensing and Control of an Irrigation System Using a Distributed Wireless Sensor Network." In: *IEEE Transactions on Instrumentation and Measurement* 57 (7 July 2008). URL: https://ieeexplore.ieee.org/abstract/document/4457920/authors#authors.

[9] WiseChip Semiconductor Inc. *UG-2828GDEDF11 OEL Display Module Product Specification*. URL: https://www.waveshare.com/w/upload/4/43/UG-2828GDEDF11.pdf (visited on 02/29/2020).

[10] Diodes Incorporated. *P-Channel Enhancement Mode MOSFET*. URL: https://www.diodes.com/assets/Datasheets/DMP2110UW.pdf (visited on 03/22/2020).

[11] 3D Insider. *16 Different Types of 3D Printing Materials*. URL: https://3dinsider.com/3d-printing-materials/ (visited on 03/18/2020).

[12] Texas Instruments. *LM339, LM239, LM139, LM2901 Quad Differential Comparators*. URL: http://www.ti.com/lit/ds/symlink/lm339.pdf?ts=1591473017584&ref_url=https://www.google.com/ (visited on 04/25/2020).

[13] Texas Instruments. *LOW INPUT VOLTAGE, 0.7-V BOOST CONVERTER WITH 5.5-microA QUIESCENT CURRENT*. URL: http://www.ti.com/lit/ds/symlink/tps61222-ep.pdf?ts=1591657551072&ref_url=https://www.google.com/ (visited on 04/19/2020).

[14] KiCad. *KiCad EDA - A Cross Platform and Open Source Electronics Design Automation Suite*. URL: https://www.kicad-pcb.org/ (visited on 03/13/2020).

[15] Newegg. *NRF24L01+ 2.4GHz Antenna Wireless Transceiver RF Module ISM For Arduino & Raspberry Pi Compatible*. URL: https://www.newegg.com/p/2A7-00D0-00035 (visited on 03/06/2020).

[16] Particle. *Particle Argon*. URL: https://docs.particle.io/argon/ (visited on 03/17/2020).

[17] City of Santa Barbara. *Santa Barbara Water Waste Report*. URL: https://www.santabarbaraca.gov/gov/depts/pw/resources/conservation/reportwater/default.asp (visited on 04/02/2020).

[18] ON Semiconductor. *Axial Lead Rectifiers*. URL: https://www.onsemi.com/pub/Collateral/1N5817-D.PDF (visited on 03/13/2020).

[19] Tech Spray. *The Essential Guide to Conformal Coating*. URL: https://www.techspray.com/the-essential-guide-to-conformal-coating#types (visited on 03/14/2020).

[20] Kerry Stevenson. *Waterproofing Your 3D Prints*. URL: https://www.fabbaloo.com/blog/2017/10/19/waterproofing-your-3d-prints (visited on 03/18/2020).

[21] US Geological Survey. *Estimated Use of Water in the United States in 2000*. URL: https://pubs.usgs.gov/circ/2004/circ1268/htdocs/table07.html (visited on 03/03/2020).

[22] SOLOMON SYSTECH. *128 RGB x 128 Dot Matrix OLED/PLED Segment/Common Driver with Controller*. URL: https://www.waveshare.com/w/upload/a/a7/SSD1351-Revision_1.5.pdf (visited on 02/29/2020).

[23] TLX Technologies. *Latching Solenoid Theory*. URL: https://www.tlxtech.com/understanding-solenoids/theory-operation/latching-solenoid-theory (visited on 03/11/2020).

[24] TMRh20. *RF24 Library*. URL: http://tmrh20.github.io/RF24/index.html (visited on 03/06/2020).

[25] TMRh20. *RF24 Mesh Library*. URL: https://tmrh20.github.io/RF24Mesh/index.html (visited on 03/06/2020).

[26] TMRh20. *RF24 Network Library*. URL: https://tmrh20.github.io/RF24Network/ (visited on 03/06/2020).

[27] Toshiba. *TOSHIBA Field-Effect Transistor Silicon N-Channel MOS Type (U-MOS VII-H) SSM3K333R*. URL: https://www.mouser.com/datasheet/2/408/SSM3K333R_datasheet_en_20140301-1150953.pdf (visited on 03/13/2020).

[28] Toshiba. *TOSHIBA Field-Effect Transistor Silicon P-Channel MOS Type (U-MOSVI) SSM3J332R*. URL: https://www.mouser.com/datasheet/2/408/SSM3J332R_datasheet_en_20181015-1150575.pdf (visited on 03/13/2020).

[29] Battery University. *How to Prolong Lithium-based Batteries*. URL: https://batteryuniversity.com/learn/article/how_to_prolong_lithium_based_batteries (visited on 03/15/2020).

[30] Volt-log. *How to Waterproof Your Electronics or PCBs*. URL: https://www.instructables.com/id/How-to-Waterproof-Your-Electronics-or-PCBs/ (visited on 03/12/2020).

[31] WODEYIJIA. *GM5539 PDF - Documentation language EN*. URL: https://www.tme.eu/Document/01ba1573cea1124ddd9a55cccc53ed63/all.pdf (visited on 03/07/2020).

6 Appendix

6.1 Full Schematic

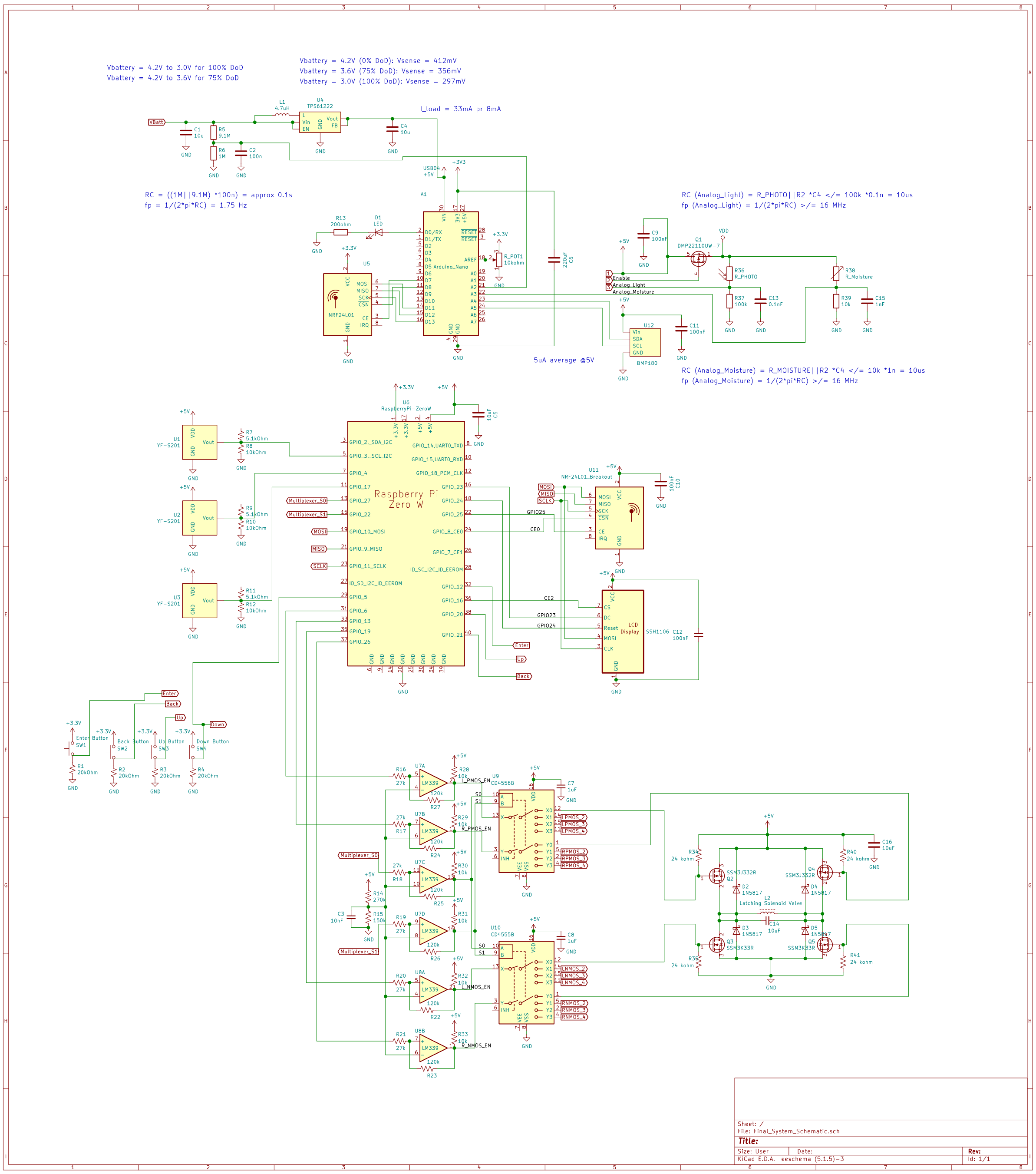


Figure 55: This is the full schematic for the Intuitive Auto-Irrigation System. [16]

[16]This figure was made in KiCAD [14].

6.2 Wireless Range Testing SOP

```
Intuitive Auto Irrigation | Wireless Height Test
Wireless Height Test
Last Updated: 29 Feb 2020
Version 1
----------------




Purpose
Due to the nature of the RF signals, the Height from the Ground (HfG) to the RF module is
important when considering the maximum range of node-to-node wireless communication. This
test is designed to highlight the differences in the range limitations when the RF modules
are placed at different heights from the ground. In specific terms, this test looks at the
limitations of the exact testing space for different heights of the modules.


Expectations
We expect to see a significantly reduced range from nodes that are placed close to the
ground while nodes located up higher will have much better long-range communication. In
addition to the grounding effect on the RF signals, the line-of-sight also plays a big
role when the ground is not perfectly flat. Below, the expectations are given in the form:


(Distance_Master Node_HfG - Distance_Sensor Node_HfG:         Works/Fails)
With expected results that usually vary a lot in italics.
100ft_5cm - 100ft_5cm:                  Works
100ft_50cm - 100ft_5cm:                 Works
100ft_50cm - 100ft_50cm:                Works
100ft_125cm - 100ft_125cm:              Works
--
150ft_5cm - 150ft_5cm:                  Works
150ft_50cm - 150ft_5cm:                 Works
150ft_50cm - 150ft_50cm:                Works
150ft_125cm - 150ft_125cm:              Works
--
200ft_5cm - 200ft_5cm:                  Works
200ft_50cm - 200ft_5cm:                 Works
200ft_50cm - 200ft_50cm:                Works
200ft_125cm - 200ft_125cm:              Works
--
250ft_5cm - 250ft_5cm:                  Fails
250ft_50cm - 250ft_5cm:                 Works
250ft_50cm - 250ft_50cm:                Works
250ft_125cm - 250ft_125cm:              Works
--
300ft_5cm - 300ft_5cm:                  Fails
300ft_50cm - 300ft_5cm:                 Works
300ft_50cm - 300ft_50cm:                Works
300ft_125cm - 300ft_125cm:              Works
--
350ft_5cm - 350ft_5cm:                  Fails
350ft_50cm - 350ft_5cm:                 Works
350ft_50cm - 350ft_50cm:                Works
350ft_125cm - 350ft_125cm:              Works
--
400ft_5cm - 400ft_5cm:                  Fails
```

```
400ft_50cm - 400ft_5cm:               Works
400ft_50cm - 400ft_50cm:              Works
400ft_125cm - 400ft_125cm:            Works
--
450ft_5cm - 450ft_5cm:                Fails
450ft_50cm - 450ft_5cm:               Fails
450ft_50cm - 450ft_50cm:              Works
450ft_125cm - 450ft_125cm:            Works
--
500ft_5cm - 500ft_5cm:                Fails
500ft_50cm - 500ft_5cm:               Fails
500ft_50cm - 500ft_50cm:              Fails
500ft_125cm - 500ft_125cm:            Fails

Procedure
1. Set up two nRF24L01 wireless transceivers with complimentary wireless protocols on
   different devices
   1. Each sensor node should have a unique nodeID
   2. Extra nodes can be left on an external power supply if sufficient personnel are
      not available
   3. Sensor array is not needed
2. Verify proper connection between devices by ensuring valid communication at short
   distances between nodes
3. Move out into an open area where distance between nodes can be measured
4. Place the master node in a stationary, visible location
5. Orient each antenna to face the other node
6. Create distance between the connected nodes by walking away from the stationary node
   1. Attempt to keep the moving node at the desired height the entire time
7. Walk out to the next distance/range checkpoint, given as a certain number of feet from
   the master node, noted in the expectations segment
8. Once there, cycle through the varying height tests while verifying reliable connection
   1. For each range-height test, ensure at least 5 consecutive, successful packets
      delivered from the sensor node to the master
   2. If reliable connection cannot be established within 1 minute, that test is
      considered ‘Failed’
   3. A direct line-of-sight must be ensured for each test to guarantee accurate
      test results
9. After cycling through the various height tests at a single distance, move out to the
   next distance by returning to step 7
   1. Once the last distance checkpoint has been tested, compile all the results
```

6.3 Soil Moisture Sensor Testing SOP

```
Intuitive Auto Irrigation | Soil Moisture Sensor Test
Soil Moisture Sensor Testing
Last Updated: 9 March 2020
----------------

Purpose
The IAI moisture sensor uses the moisture-dependent variable resistance of soil to determine when
the soil is dry and when it is wet. Through informal testing of wet and dry soil, the group has
determined that the sensor can tell the difference between subjectively wet and dry soil. But, the
real resolution for small increments of watering has yet to be determined. This experiment aims to
begin to form a characteristic graph for both resistance and subsequent voltage readings for the
moisture sensor under otherwise fixed conditions. The test will compare gravimetric water content
of the soil vs. its resistance.

Expectations
1. Once soil is completely dry, measured resistance has consistently been measured as an
   open circuit on the multimeter
2. Data should follow a form of resistance =  ax-b where a and b are regression
   constants and x is the gravimetric moisture value. The constant a can vary by up to 50%
   between separate trials under the same soil sample while b only varies at most 15% under the
   same conditions. The constant b has been observed to be  more dependent on the soil type than a.
   Correlation coefficients, fixing for soil specific soil samples,  have been observed to be
   R2 = 0.97 +/- 0.02

Procedure
1. Extract a sample of soil and specify where the soil is from and when it was extracted. Remove
any non-soil components (grass, leaves, etc.).
2. Dry the soil sample (follows NDSU soil testing lab, but with stricter rules)
   1. Lay out newspaper on floor or table
   2. Spread the soil layer on the newspaper or paper towels so that the layer is at most 0.25
    inch thick. Break up large clods of soil (golf ball size or larger)
   3. If possible use a fan to blow on sample
   4. Stir sample periodically (every 8 hours)
   5. Allow sample at least 48 hours to dry
3. Acquire an appropriate container to allow soil dimensions to be a depth of over 4 inches and
   width of over 1.25 inches. This will allow the probes to be fully submerged.
4. Using a scale, measure the container's weight. This will be the offset value when measuring
   gravimetric water content
5. Add the now dry soil to the container with at least the dimensions listed in step 3 and pat
   down the soil to keep the compaction fixed. Mark this height with a pen to maintain the
   soil compactness.
6. Record the weight of the dry soil and subtract the weight of the empty container
7. Place probes 1" apart in the soil and submerged up to the head of the probe and measure the
   resistance of the soil. This will be the first data point for gravimetric water potential
   at 0%.
8. Remove the soil and add to to a flat surface
9. With a scale, measure a small portion of water (1-20g).  Add this water to the sample and
   mix the water into the soil with a spoon or other stirring method. Mix for at least 30 seconds
   or until the soil reaches a homogeneous color/appearance. Best results occur when using small
   increments early on and increasing them slightly as more water is added. This is because data
```

```
   is more scarce early on in a log-log plot.
10. Plot the data in the form of table A below. This will determine the gravimetric water
    content at each data point by taking the water content of the soil divided by the dry
    soil measurement.

         __________________________________________
        |Water|      Gravimetric      |  Resistance   |
        |added|        Water          |   from 1"     |
        | (g) |     Content (%)       | Probes (kOhms)|
        |_____|_______________________|_______________|
        |  0  | =0/dry soil mass (g) |       x       |
        |  2  | =2/dry soil mass (g) |       y       |
        |  4  | =4/dry soil mass (g) |       z       |
        |_____|_______________________|_______________|
                Table A: Example data logging


11. Once again place probes 1" apart in the soil at the same compactness and submerged up to the head of the probe
12. Repeat steps 8-10 until the desired data set is reached. 100% gravimetric soil moisture
    content has commonly been used as the stopping point but can be passed.
```

6.4 Appendix: Local Master Code

```
#ifndef __cplusplus
#define __cplusplus
#endif

// **********INCLUDES ***********
#include <RF24/RF24.h>
#include <RF24Network/RF24Network.h>
#include <RF24Mesh/RF24Mesh.h>
#include <RF24/utility/RPi/bcm2835.h>
#include <iostream>
#include <cstdio>
#include <vector>
#include "OLED_GFX.h"
#include "OLED_Driver.h"
#include <stdio.h>
#include <stdlib.h>             //exit()
#include <signal.h>     //signal()
#include <math.h>
#include <time.h>
#include <sqlite3.h>
#include "obj/Debug.h"

/****GLOBALS ****/
#define LED RPI_BPLUS_GPIO_J8_07
#define pushButton RPI_BPLUS_GPIO_J8_29
#define SPI_SPEED_2MHZ 2000000
#define TRUE 1
#define FALSE 0
#define MAX_ELEMENTS 100
#define MAX_SENSORS 20
#define NUM_HOSES 3
#define MOISTURE 0
#define SUNLIGHT 1
#define TEMP 2

// Water Delivery
#define WATER_OFF 0
#define WATER_ON 1
#define PMOS_ON 0       //States for MOS gates
#define PMOS_OFF 1      //1 -> 5V, 0 -> 0V
#define NMOS_ON 1
#define NMOS_OFF 0
#define DEMUX_OFF 1     //Set to Low Enable
#define DEMUX_ON 0
#define LPMOS_Pin 6
#define LNMOS_Pin 13
#define RPMOS_Pin 19
#define RNMOS_Pin 26

// Buttons
#define ENTER_Pin 12
#define BACK_Pin 5
#define DOWN_Pin 20
#define UP_Pin 21

//Flow Sensor Pins
#define FLOW_SENSOR_1_Pin 4
```

```
#define FLOW_SENSOR_2_Pin 3
#define FLOW_SENSOR_3_Pin 2

//Select Pins
#define SEL_1_Pin 17
#define SEL_0_Pin 27

//Flow Sensor Conversions
#define FS_CAL_A 46.2   //Variables used for Characterized FS Regression
#define FS_CAL_B 40.8
#define FS_CAL_STEADY 404       //Calibration factor used when FS signal is steady
#define LITERS_TO_GAL 0.264172

// Time
#define FORECAST_CALL 1800000
#define HOURS_36 129600000
#define MIN_10 600000
#define MIN_5 300000
#define MIN_3 180000
#define MIN_2 120000
#define MIN_1 60000
#define FIVE_SECONDS 5000
#define EIGHT_SECONDS 8000
#define ONE_SECOND 1000
#define PULSE_DURATION 3000
#define FET_DELAY 10
#define MUX_DELAY 50
#define HUNDRED_MILLI 100

// CSV Files
#define CSVFILENAME "Data_Log_to_db.csv"

// MISC
#define DEBUG_ON 1
#define DATA_PARAM_SIZE 10
#define DATA_PARAM_NUM 11

/*Avialable Colors
#define BLACK   0x0000
#define BLUE    0x001F
#define RED     0xF800
#define GREEN   0x07E0
#define CYAN    0x07FF
#define MAGENTA 0xF81F
#define YELLOW  0xFFE0
#define WHITE   0xFFFF
*/

/****Configure the Radio ****/
/*Radio Pins:
 *          CE: 22
 *          CSN: 24
 *          MOSI: 19
 *          MISO: 21
 *          CLK: 23   */
RF24 radio(RPI_BPLUS_GPIO_J8_22, RPI_BPLUS_GPIO_J8_24, BCM2835_SPI_SPEED_8MHZ);
RF24Network network(radio);
RF24Mesh mesh(radio, network);

```

```
// C_Struct stores relevant thresholds
typedef struct
{
    float sM_thresh;
    float sM_thresh_OO;
    float lL_thresh;
    uint16_t tC_thresh;
    uint16_t time_thresh;
}
C_Struct;

// D_Struct stores the relevant sensor data
typedef struct
{
    float soilMoisture;
    float lightLevel;
    uint16_t temp_C;
    uint8_t digitalOut;
    uint8_t nodeID;
    uint8_t battLevel;
}
D_Struct;

typedef struct
{
    float precipProb;
    int temperature;
    int humidity;
    int pressure;
    int windSpeed;
    int windBearing;
}
Forecast;

typedef struct
{
    uint8_t status;
    uint8_t sensors[MAX_SENSORS];
    uint8_t waterLevel;
    uint8_t tally;
    uint8_t flowRate;
    uint8_t rainFlag;
    uint32_t rainTimer;
    uint8_t control;
}
Hoses;

//States for Water Delivery SM
typedef enum
{
    HOSE_IDLE,
    HOSE_ON_S1,
    HOSE_ON_S2,
    HOSE_ON_S3,
    HOSE_OFF_S1,
    HOSE_OFF_S2,
    HOSE_OFF_S3,
}
w_State;
```

```
//States for OLED SM
typedef enum
{
    WELCOME_PAGE,
    SLEEP,
    HOME_PAGE,
    SENSORS_HOME,
    SENSORS_LIST,
    SENSORS_CURRENT,
    SENSORS_PLOT_START,
    SENSORS_PLOT,
    HOSES_HOME,
    HOSES_STATUS,
    HOSES_CONTROL,
    HOSES_WATER,
    HOSES_MAP,
    HOSES_MAP_SELECT,
    SETTINGS_HOME,
    SETTINGS_SLEEP,
    SETTINGS_CAL,
    SETTINGS_COLOR,
    SETTINGS_RESET,
}
OLED_State;

// Enum for hose specification
typedef enum
{
    HOSE0,
    HOSE1,
    HOSE2,
}
HOSE_NUM;

// Enum for control types
enum
{
    OFF,
    ON,
    AUTOMATIC,
};

// Data Vars
D_Struct D_Dat;
D_Struct current_Dat_1; //Most recent Sensor Data
D_Struct current_Dat_2;
D_Struct current_Dat_3;
//D_Struct current_Dat_4;
//D_Struct current_Dat_5;
D_Struct sensor_data[MAX_ELEMENTS];
D_Struct sensor1_data[MAX_ELEMENTS];    //Recent Sensor Data for Plotting
D_Struct sensor2_data[MAX_ELEMENTS];
D_Struct sensor3_data[MAX_ELEMENTS];
//D_Struct sensor4_data[MAX_ELEMENTS];
//D_Struct sensor5_data[MAX_ELEMENTS]
uint8_t dFlag = 0;
uint8_t dataDat = 1;
uint8_t column_flag = 0;
```

```
int sd_index = -1;
int sd_index_1 = -1;
int sd_index_2 = -1;
int sd_index_3 = -1;

//Struct Declarations
OLED_State oledState = WELCOME_PAGE;
OLED_State nextPage = WELCOME_PAGE;
static w_State waterState = HOSE_IDLE;  //Water Deliver SM state var

//Array Declarations
static int mappedSensors[MAX_SENSORS];  //Intialize Mapping Variables
static int unmappedSensors[MAX_SENSORS];
//static uint8_t prev_waterLevel[3];
static char timeBuffer[20];     //Buffers used for format Variables into strings
static char intBuffer[20];
static char sensorIDBuffer[100];
static char hoseBuffer[100];
//static char testBuffer2[100];
//static char testBuffer3[100];
static char currentBuffer1[100];        //Used for printing current sensor readings
static char currentBuffer2[100];
static char currentBuffer3[100];
static char currentBuffer4[100];
static char currentBuffer5[100];

//Variable Declarations
static uint8_t dataType = 0;    //Determines type of data to be  printed
static uint16_t oledColor;

static int oledHour;    //Store System Time
static int oledMinute;
//static int tempVal = 0;
static int wholeVal = 0;        //Converting
static int decimalVal = 0;
static int hose0_elements = 0;  //keeps track of number of sensors mapped to hose
static int hose1_elements = 0;
static int hose2_elements = 0;
static int new_Data = FALSE;    //Flag set when new data received to update OLED
static int selected_Node = 0;   //Determine which set of Sensor node Data to plot
static int nodeUnmapped = TRUE; // Set True if at least one unmapped sensor
static int sensorToMap = 0;     // Var keeping track of which sensor to map

//Flow Sensor Support
static int tach_fs[4];  //Used for individually tracking flow rates
static int prevTach_fs[4];      //and Water Output using YF-S201 Flow Sensors
static int pulseCount_fs[4];
static float water_liters_fs[4];
static float prev_Liters_fs[4];
static float water_gal_fs[4];
static float moisture_s_thresh[MAX_SENSORS];    //Moisture Thresholds
//static float testFloat = 64.757065;
static float temp_wholeVal = 0;
static float temp_decimalVal = 0;
static float test_Moisture = 0;

//Button Event Detection Variables
static int prevArrowState, arrowState = 0;
static int lastUpButtonState, lastDownButtonState, lastBackButtonState, lastEnterButtonState,
```

```c
upButtonValue, downButtonValue, backButtonValue, enterButtonValue,
upButtonValue2, downButtonValue2, backButtonValue2, enterButtonValue2,
ENTER_PRESSED, UP_PRESSED, DOWN_PRESSED, BACK_PRESSED = 0;

// Timers
uint32_t forecastTimer = 0;
uint32_t waterDeliveryTimer = 0;
uint32_t wTimer = 0;    //Timer used for driving Water Delivery testing
uint32_t oledSleepTimer = 0;    //Timer used for updating OLED testing
uint32_t sleepTime = 0; //Variable to set time to wait for button press until SLEEP
uint32_t connectionTimer = 0;

// RF24 Vars
const static uint8_t nodeID = 0;        // 0 = master
uint8_t num_nodes = 0;

// Forecast Support
Forecast Forecast1;
char buffer[10];
double data[6];
FILE * fp;

// Water Delivery Support
Hoses Hose0, Hose1, Hose2;
Hoses Hose[3];
uint8_t hose_statuses = 0;
uint8_t prev_hose_statuses = 0;

/****Helper Fxn Prototypes ****/
int Timer(uint32_t, uint32_t);
void setup(void);
void checkButtons(void);
void printHoseStatus(int16_t x, int16_t y, uint8_t status);
int printGrid(int16_t x0, int16_t x1, int16_t y0, int16_t y1, int16_t xtics,
        int16_t ytics);
int printAxesLabels(int16_t x0, int16_t y0);
int plotSampleData(D_Struct data[], uint8_t dataType, int16_t size);
int WaterDeliverySM(uint8_t status, uint32_t delayP_N, uint32_t pulseTime);
void OLED_PrintArrow(int x, int y);
void OLED_SM(uint16_t color);
void LPMOS_Set(uint8_t status);
void RPMOS_Set(uint8_t status);
void LNMOS_Set(uint8_t status);
void RNMOS_Set(uint8_t status);
void recordPulses_FS(int i);
float convertPulse_Liters(int pulseCount);
float convertLiters_Gals(float liters);
int convertFloat_String(float in, char buffer[100]);
void Reset_System(void);
void Set_Select(uint8_t hose_selected);
uint8_t WaterDelivery(HOSE_NUM);
void insert_into_database(sqlite3 *mDb, double soil_moisture, int light,
        int temp, double pressure, double precip_prob, int output, int nodeID,
        double battery_lvl, int hose1, int hose2, int hose3);
void processCSV(sqlite3 *db);
int createTable(sqlite3 *db);
static int callback(void *NotUsed, int argc, char **argv, char **azColName);
void DEBUG_LOG(const char*);

```

```
/***************************************************************************/
/****Void Setup ****/
void setup(void)
{
    DEBUG_LOG("Starting setup");
    //DEBUG("Starting setup.\n");

        // Initialize the Hose array
    Hose[0] = Hose0;
    Hose[1] = Hose1;
    Hose[2] = Hose2;
    Hose[0].waterLevel = 1;
    Hose[1].waterLevel = 1;
    Hose[2].waterLevel = 1;
    Hose[0].control = AUTOMATIC;
    Hose[1].control = OFF;
    Hose[2].control = OFF;
    Hose[0].status = WATER_ON;
    Hose[1].status = WATER_OFF;
    Hose[2].status = WATER_OFF;

        // Init the GPIO Library
        DEBUG_LOG("Initializing the GPIO Library");

    DEV_ModuleInit();
    Device_Init();
    bcm2835_init();
    bcm2835_spi_begin();

        // Set Pins to Output
        DEBUG_LOG("Setting GPIO Pin Modes");

    DEV_GPIO_Mode(LPMOS_Pin, 1);
    DEV_GPIO_Mode(RPMOS_Pin, 1);
    DEV_GPIO_Mode(LNMOS_Pin, 1);
    DEV_GPIO_Mode(RNMOS_Pin, 1);

        // Set SELECT Pins to Output
    DEV_GPIO_Mode(SEL_1_Pin, 1);
    DEV_GPIO_Mode(SEL_0_Pin, 1);

        // Set Pins to Input
    DEV_GPIO_Mode(ENTER_Pin, 0);
    DEV_GPIO_Mode(BACK_Pin, 0);
    DEV_GPIO_Mode(DOWN_Pin, 0);
    DEV_GPIO_Mode(UP_Pin, 0);

        // Set Flow Sensor Pins to Input
    DEV_GPIO_Mode(FLOW_SENSOR_1_Pin, 0);
    DEV_GPIO_Mode(FLOW_SENSOR_2_Pin, 0);
    DEV_GPIO_Mode(FLOW_SENSOR_3_Pin, 0);

        // Turn off the H-Bridge
    LPMOS_Set(PMOS_OFF);        //Initial States for MOS devices
    RPMOS_Set(PMOS_OFF);
    LNMOS_Set(NMOS_OFF);        // Notice the diff b/t PMOS and NMOS states
    RNMOS_Set(NMOS_OFF);

        // Set this node as the master node
```

```
    mesh.setNodeID(nodeID);
    printf("Node ID: %d\n", nodeID);
    radio.setPALevel(RF24_PA_MAX);

        // Initialize the mesh and check for proper chip connection
    if (mesh.begin())
    {
        printf("\nInitialized: %d\n", radio.isChipConnected());
    }
        // Print out debugging information
    radio.printDetails();

        //Initialize OLED variables
    oledColor = WHITE;  //
    sleepTime = MIN_3;  //Set default sleep time to 3 minutes
    oledSleepTimer = bcm2835_millis();  //Start Oled Sleep Timer

        //Testing Current Struct Plotting
    current_Dat_1.nodeID = 2;
    current_Dat_1.soilMoisture = 36.8;
    current_Dat_1.lightLevel = 90.2;
    current_Dat_1.temp_C = 85;

    current_Dat_2.nodeID = 5;
    current_Dat_2.soilMoisture = 67.4;
    current_Dat_2.lightLevel = 72.6;
    current_Dat_2.temp_C = 85;

    current_Dat_3.nodeID = 4;
    current_Dat_3.soilMoisture = 56.5;
    current_Dat_3.lightLevel = 76.3;
    current_Dat_3.temp_C = 85;
    selected_Node = 2;

    moisture_s_thresh[0] = 42;
    moisture_s_thresh[1] = 32;
    moisture_s_thresh[2] = 55;

    int i = 0;
    test_Moisture = 60;

        //Plotting Moisture Testing
    for (i = 0; i < MAX_ELEMENTS; i++)
    {
        sensor2_data[i].soilMoisture = test_Moisture;
        if (((i > 30) && (i < 35)) || ((i > 60) && (i < 65)))
        {
            test_Moisture += 3;
        }
        else if (((i > 36) && (i < 40)) || ((i > 66) && (i < 70)))
        {
            test_Moisture -= 0.8;
        }
        else
        {
            test_Moisture -= 0.3;
        }
    }

```

```
    DEBUG_LOG("Setup Complete");

    return;
}

/*******************************************************************************/
int main(void)
{
    setup();
    sqlite3 * db;
    int rc;

        //Access Local Time from RPi
    time_t t = time(NULL);
    struct tm tm = *localtime(&t);

        //Store as variables for comparison
    oledHour = tm.tm_hour;
    oledMinute = tm.tm_min;

        //Store Time variables into strings for printing
    if (oledHour == 0)
    {
        //Takes Care of 12:00 AM Error
        sprintf(timeBuffer, "%02d:%02d AM", (oledHour + 12), oledMinute);
    }
    else if (oledHour < 12)
    {
        sprintf(timeBuffer, "%02d:%02d AM", oledHour, oledMinute);
    }
    else if (oledHour == 12)
    {
        //Takes care of prev error w/ 12:00 PM
        sprintf(timeBuffer, "%02d:%02d PM", oledHour, oledMinute);
    }
    else
    {
        sprintf(timeBuffer, "%02d:%02d PM", (oledHour - 12), oledMinute);
    }

    printf("now: %d-%02d-%02d %02d:%02d:%02d\n", tm.tm_year + 1900, tm.tm_mon
                + 1, tm.tm_mday, tm.tm_hour, tm.tm_min, tm.tm_sec);

        DEBUG_LOG("Begin Loop");
    while (1)
    {
        // Keep the network updated
        mesh.update();

        // Since this is the master node, we always want to be dynamically assigning
        //    addresses the new nodes
        mesh.DHCP();

        /****Check For Available Network Data ****/
                DEBUG_LOG("Checking for Available Network Data");

        // Check for incoming data from other nodes
        if (network.available())
        {
```

```
                // Create a header var to store incoming network header
            RF24NetworkHeader header;
                // Get the data from the current header
            network.peek(header);

                // First ensure the message is actually addressed to the master
            if (header.to_node == 0)
            {
                // Switch on the header type to sort out different message types
                switch (header.type)
                {
                        // Retrieve the data struct for D type messages
                    case 'D':
                        printf("Message Received\n");
                        // Use the data struct to store data messages and print out the result
                        network.read(header, &D_Dat, sizeof(D_Dat));
                        // Set the flag that indicates we need to respond to a new message
                        dFlag = 1;

                        // Here is where we add the sensor data to the sensor data array
                        // But first we want to see if the sensor data array is full
                        if (sd_index >= MAX_ELEMENTS)
                        {
                                // checks if the index is at the max # of elements
                            int i, j = 0;
                                // Now we transfer the 10 most recent data values to the bottom of the list
                            for ((i = MAX_ELEMENTS - 10); i < MAX_ELEMENTS; i++)
                            {
                                sensor_data[j] = sensor_data[i];        // j is the bottom, i is the top
                                j++;
                            }
                                // Reset the sensor data index
                            sd_index = 10;
                        }
                        // Increment the sensor data index for the new value
                        sd_index++;
                        // Then place the new data into the array
                        sensor_data[sd_index] = D_Dat;
                        break;

                        // Do not read the header data, instead print the address inidicated by the header type
                    default:
                        break;
                }
            }
            else
            {
                // Generally will never get here
                // This basically just removes the message from the input buffer
                network.read(header, 0, 0);
            }
        }
                DEBUG_LOG("Checking for Available Network Data Complete");

        /****Update List of Nodes ****/
                DEBUG_LOG("Updating Node List");

        if (Timer(MIN_2, connectionTimer))
        {
```

```
            connectionTimer = millis();
                // Other option is to create a dict after receiving a message
            if (num_nodes != mesh.addrListTop)
            {
                num_nodes = mesh.addrListTop;
                printf("\nConnected nodes: ");
                int i = 0;
                for (i = 0; i < mesh.addrListTop; i++)
                {
                        // Add sensor nodes to the list of sensors mapped to the hose
                    Hose[HOSE0].sensors[i] = mesh.addrList[i].nodeID;
                    if (i == (mesh.addrListTop - 1))
                    {
                        printf("%d\n", mesh.addrList[i].nodeID);
                    }
                    else
                    {
                        printf("%d, ", mesh.addrList[i].nodeID);
                    }
                }
                // Reset the water level threshold according to the # of sensors
                Hose[HOSE0].waterLevel = i / 2;
                printf("Water Level Threshold: %d\n\n", Hose[HOSE0].waterLevel);
            }
        }
                DEBUG_LOG("Finished Updating List of Nodes");

        /****Data Logging ****/

        if (dFlag)
        {
                // This should be the last thing that gets done when data is received
            dFlag = 0;

            /****Write Data Values to SD Card ****/
                        DEBUG_LOG("Begin Logging Data");

                /* create/open the file to append to (this is the file that stores
                                all the sensor data) */
            FILE *dataLog_fp = fopen("Data_Log.csv", "a");

                // prints out main column headers for the data file.
                // conditional here: output if first loop, dont afterward, controlled by column_flag
            if (column_flag == 0)
            {
                fprintf(dataLog_fp, "Soil_Moisture, Ambient_Light, "
                                "Ambient_Temp, Barometric_Pressure, Precip_Prob, "
                                "Digital_Output, Node_ID, Battery_Level, Hose_1, Hose_2, "
                                "Hose_3\n");
                column_flag = 1;
            }

                // prints out elements of the sensor data struct to the file
            fprintf(dataLog_fp, "%13f,   ", D_Dat.soilMoisture);
            fprintf(dataLog_fp, "%13f,   ", D_Dat.lightLevel);
            fprintf(dataLog_fp, "%19d,   ", D_Dat.temp_C);
            fprintf(dataLog_fp, "%19d,   ", Forecast1.pressure);
            fprintf(dataLog_fp, "%11f,   ", Forecast1.precipProb);
            fprintf(dataLog_fp, "%14d,   ", D_Dat.digitalOut);
```

```
            fprintf(dataLog_fp, "%7d,   ", D_Dat.nodeID);
            fprintf(dataLog_fp, "%14d,   ", D_Dat.battLevel);
            fprintf(dataLog_fp, "%5d,   ", Hose[0].status);
            fprintf(dataLog_fp, "%5d,   ", Hose[1].status);
            fprintf(dataLog_fp, "%5d\n", Hose[2].status);

                // close the file
            fclose(dataLog_fp);

                /* create/open the file to write to (this is the file that stores
                        only the last dataset, which is then transferred to the database) */
            FILE *dataLogToDb_fp = fopen("Data_Log_to_db.csv", "w");

                // prints out main column headers for the data file.
            fprintf(dataLogToDb_fp, "Soil_Moisture, Ambient_Light, "
                                    "Ambient_Temp, Barometric_Pressure, Precip_Prob, "
                                    "Digital_Output, Node_ID, Battery_Level, Hose_1, Hose_2, "
                                    "Hose_3\n");

                // prints out elements of the sensor data struct to the file
            fprintf(dataLogToDb_fp, "%13f,", D_Dat.soilMoisture);
            fprintf(dataLogToDb_fp, "%13f,", D_Dat.lightLevel);
            fprintf(dataLogToDb_fp, "%19d,", D_Dat.temp_C);
            fprintf(dataLogToDb_fp, "%19d,", Forecast1.pressure);
            fprintf(dataLogToDb_fp, "%11f,", Forecast1.precipProb);
            fprintf(dataLogToDb_fp, "%14d,", D_Dat.digitalOut);
            fprintf(dataLogToDb_fp, "%7d,", D_Dat.nodeID);
            fprintf(dataLogToDb_fp, "%14d,", D_Dat.battLevel);
            fprintf(dataLogToDb_fp, "%5d,", Hose[0].status);
            fprintf(dataLogToDb_fp, "%5d,", Hose[1].status);
            fprintf(dataLogToDb_fp, "%5d\n", Hose[2].status);

                // close the file
            fclose(dataLogToDb_fp);

                        DEBUG_LOG("Data Logging Completed");

            /****SQLite Database ****/

                        DEBUG_LOG("Accessing Database");
                // creates and opens database
            rc = sqlite3_open("sensordata.db", &db);

            if (rc)
            {
                fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
            }
            else
            {
                fprintf(stdout, "Opened database successfully\n");
            }

                // creates the table in the database
            createTable(db);

                        DEBUG_LOG("Updating Database.");

                /* takes in the data from the csv file (ignores the first line of
                        headers), and places the data into the table */
```

```
            processCSV(db);

            /*Close database */
            rc = sqlite3_close(db);

                        DEBUG_LOG("Database Update Complete.");

                //Update Struct Variables for Data
            if (D_Dat.nodeID == 2)
            {
                // Update Recent Sensor Node Value
                if (sd_index_1 >= MAX_ELEMENTS)
                {
                        // checks if the index is at the max # of elements
                    int i, j = 0;
                        // Now we transfer the 10 most recent data values to the bottom of the list
                    for ((i = MAX_ELEMENTS - 10); i < MAX_ELEMENTS; i++)
                    {
                        sensor1_data[j] = sensor1_data[i];      // j is the bottom, i is the top
                        j++;
                    }
                        // Reset the sensor data index
                    sd_index_1 = 10;
                }
                // Increment the sensor data index for the new value
                sd_index_1++;
                // Then place the new data into the array
                sensor1_data[sd_index_1] = D_Dat;
                current_Dat_1 = D_Dat;
                new_Data = TRUE;
            }
            else if (D_Dat.nodeID == 5)
            {
                if (sd_index_2 >= MAX_ELEMENTS)
                {
                        // checks if the index is at the max # of elements
                    int i, j = 0;
                        // Now we transfer the 10 most recent data values to the bottom of the list
                    for ((i = MAX_ELEMENTS - 10); i < MAX_ELEMENTS; i++)
                    {
                        sensor2_data[j] = sensor2_data[i];      // j is the bottom, i is the top
                        j++;
                    }
                        // Reset the sensor data index
                    sd_index_2 = 10;
                }
                // Increment the sensor data index for the new value
                sd_index_2++;
                // Then place the new data into the array
                sensor2_data[sd_index_2] = D_Dat;
                current_Dat_2 = D_Dat;
                new_Data = TRUE;
            }
            else if (D_Dat.nodeID == 4)
            {
                if (sd_index_3 >= MAX_ELEMENTS)
                {
                        // checks if the index is at the max # of elements
                    int i, j = 0;
```

```
                        // Now we transfer the 10 most recent data values to the bottom of the list
                    for ((i = MAX_ELEMENTS - 10); i < MAX_ELEMENTS; i++)
                    {
                        sensor3_data[j] = sensor3_data[i];      // j is the bottom, i is the top
                        j++;
                    }
                        // Reset the sensor data index
                    sd_index_3 = 10;
                }
                // Increment the sensor data index for the new value
                sd_index_3++;
                // Then place the new data into the array
                sensor3_data[sd_index_3] = D_Dat;
                current_Dat_3 = D_Dat;
                new_Data = TRUE;
            }

            /****'S' and 'C' Type Message Responses ****/

                // Here we condition on if the node should be sent a configure message instead
                // Send to the message stored in the fromNode nodeID, message type 'S'
            RF24NetworkHeader p_header(mesh.getAddress(D_Dat.nodeID), 'S');
                // Data_Dat is just a 1 telling the node to go to sleep
            if (network.write(p_header, &dataDat, sizeof(dataDat)))
            {
                printf("Message Returned to %d\n\n", D_Dat.nodeID);
            }
        }

        /****Water Delivery ****/

        if (Timer(MIN_1, waterDeliveryTimer))
        {
                // reset the timer
            waterDeliveryTimer = millis();
            printf("Checking Water Delivery\n");
                // Then call WaterDelivery to see if we need to turn on each hose
            if (Hose[0].control == AUTOMATIC)
            {
                hose_statuses = WaterDelivery(HOSE0);
            }
            else if (Hose[0].control == ON)
            {
                if (Hose[0].status == WATER_OFF)
                {
                        // Call the state machine to open the solenoid valve
                    Set_Select(HOSE0);
                    DEV_Delay_ms(MUX_DELAY);
                    printf("Select to Hose 1\n");
                    printf("Hose 1 Turned On\n");
                    while (!WaterDeliverySM(WATER_ON, FET_DELAY, PULSE_DURATION));
                }
                Hose[0].status = WATER_ON;
                hose_statuses |= 0x01;
            }
            else
            {
                if (Hose[0].status == WATER_ON)
                {
```

```
                    Set_Select(HOSE0);
                    DEV_Delay_ms(MUX_DELAY);
                    printf("Select to Hose 1\n");
                    printf("Hose 1 Turned Off\n");
                        // Call the state machine to open the solenoid valve
                    while (!WaterDeliverySM(WATER_OFF, FET_DELAY, PULSE_DURATION));
                }
                Hose[0].status = WATER_OFF;
                hose_statuses &= 0xFE;  //Clear Hose Status
            }
            if (Hose[1].control == AUTOMATIC)
            {
                hose_statuses = WaterDelivery(HOSE1);
            }
            else if (Hose[1].control == ON)
            {
                if (Hose[1].status == WATER_OFF)
                {
                    Set_Select(HOSE1);
                    DEV_Delay_ms(MUX_DELAY);
                    printf("Select to Hose 2\n");
                    printf("Hose 2 Turned On\n");
                        // Call the state machine to open the solenoid valve
                    while (!WaterDeliverySM(WATER_ON, FET_DELAY, PULSE_DURATION));
                }
                Hose[1].status = WATER_ON;
                hose_statuses |= 0x02;
            }
            else
            {
                if (Hose[1].status == WATER_ON)
                {
                    Set_Select(HOSE1);
                    DEV_Delay_ms(MUX_DELAY);
                    printf("Select to Hose 2\n");
                    printf("Hose 2 Turned Off\n");
                        // Call the state machine to open the solenoid valve
                    while (!WaterDeliverySM(WATER_OFF, FET_DELAY, PULSE_DURATION));
                }
                Hose[1].status = WATER_OFF;
                hose_statuses &= 0xFC;  //Clear Hose Status

            }
            if (Hose[2].control == AUTOMATIC)
            {
                hose_statuses = WaterDelivery(HOSE2);
            }
            else if (Hose[2].control == ON)
            {

                if (Hose[2].status == WATER_OFF)
                {
                    Set_Select(HOSE2);
                    DEV_Delay_ms(MUX_DELAY);
                    printf("Select to Hose 3\n");
                    printf("Hose 3 Turned On\n");
                        // Call the state machine to open the solenoid valve
                    while (!WaterDeliverySM(WATER_ON, FET_DELAY, PULSE_DURATION));
                }
```

```
                Hose[2].status = WATER_ON;
                hose_statuses |= 0x04;
            }
            else
            {
                if (Hose[2].status == WATER_ON)
                {
                    Set_Select(HOSE2);
                    DEV_Delay_ms(MUX_DELAY);
                    printf("Select to Hose 3\n");
                    printf("Hose 3 Turned Off\n");
                        // Call the state machine to open the solenoid valve
                    while (!WaterDeliverySM(WATER_OFF, FET_DELAY, PULSE_DURATION));
                }
                Hose[2].status = WATER_OFF;
                hose_statuses &= 0xFB;  //Clear Hose Status
            }
        }

        /****Forecast Data API Call ****/

        if (Timer(FORECAST_CALL, forecastTimer))
        {
            DEBUG_LOG("Opening call to forecast API...");
            forecastTimer = millis();
                // Opens and runs the python script in the terminal
            fp = popen("python RFpython_test.py", "r");

                // error checking
            if (fp == NULL)
            {
                printf("Failed to run command.\n");
                break;
            }

            DEBUG_LOG("Call to forecast API success");
            int tmp = 0;

                // loop that extracts the outputted data from the shell and places it in an array
            while (fgets(buffer, sizeof(buffer), fp) != NULL)
            {
                sscanf(buffer, "%lf", &data[tmp]);
                ++tmp;
            }

                // moves the extracted data from the array to the struct
            Forecast1.precipProb = data[0];
            printf("Forecast1.precipProb = %f.\n", Forecast1.precipProb);
            Forecast1.temperature = round(data[1]);
            printf("Forecast1.temperature = %d.\n", Forecast1.temperature);
            Forecast1.humidity = round(data[2]);
            printf("Forecast1.humidity = %d.\n", Forecast1.humidity);
            Forecast1.pressure = round(data[3]);
            printf("Forecast1.pressure = %d.\n", Forecast1.pressure);
            Forecast1.windSpeed = round(data[4]);
            printf("Forecast1.windSpeed = %d.\n", Forecast1.windSpeed);
            Forecast1.windBearing = round(data[5]);
            printf("Forecast1.windBearing = %d.\n\n", Forecast1.windBearing);

```

```
            pclose(fp);
        }

        /**Flow Sensor Management ****/

        int i;
        // pulseCount_fs2 = 100;        // for testing
        for (i = 0; i > 3; i++)
        {
            recordPulses_FS(i); //Record Flow Sensor Tach Signals
            water_liters_fs[i] = convertPulse_Liters(pulseCount_fs[i]);
            water_gal_fs[i] = convertLiters_Gals(water_liters_fs[i]);
            prev_Liters_fs[i] = water_liters_fs[i];
        }

        /*Testing Current Data Plotting*/
        //convertFloat_String(76.5, currentBuffer1);
        //convertFloat_String(23.75, currentBuffer2);
        //sprintf(currentBuffer3, "%d", 45);
        //sprintf(currentBuffer4, "%d", 2);
        //new_Data = TRUE;

        /****UI Menu Control ****/
        //Continously Update System Time each loop
        t = time(NULL);
        tm = *localtime(&t);

        if (tm.tm_min != oledMinute)
        {
                //if new minute, update time
            oledHour = tm.tm_hour;
            oledMinute = tm.tm_min;
            if (oledHour == 0)
            {
                //Takes care of error w/ 12:00AM
                sprintf(timeBuffer, "%02d:%02d AM", (oledHour + 12), oledMinute);
            }
            else if (oledHour < 12)
            {
                sprintf(timeBuffer, "%02d:%02d AM", oledHour, oledMinute);
            }
            else if (oledHour == 12)
            {
                //Takes care of prev error w/ 12:00 PM
                sprintf(timeBuffer, "%02d:%02d PM", oledHour, oledMinute);
            }
            else
            {
                sprintf(timeBuffer, "%02d:%02d PM", (oledHour - 12), oledMinute);
            }
        }

        // First check the buttons to inform the oled
        checkButtons();
        // Then call the oled function to operate the UI
        OLED_SM(oledColor);
    }   // Loop

        // Should NEVER get here
```

```
    return (1);
}

/******************************************************************************/
/**** HELPER FXNS ****/
void DEBUG_LOG(const char* msg)
{
    if (DEBUG_ON)
    {
        fprintf(stdout,"%s.\n", msg);
    }
}


/*@name: Timer
   @param: delayThresh - timer duration
   @param: prevDelay - time in millis() when the timer started
   @return: digital high/low depending if timer elapsed or not
   This is a non-blocking timer that handles uint32_t overflow,
   it works off the internal function millis() as reference
*/
int Timer(uint32_t delayThresh, uint32_t prevDelay)
{
        // Checks if the current time is at or beyond the set timer
    if ((bcm2835_millis() - prevDelay) >= delayThresh)
    {
        return 1;
    }
    else if (millis() < prevDelay)
    {
        //Checks and responds to overflow of the millis() timer
        if (((4294967296 - prevDelay) + bcm2835_millis()) >= delayThresh)
        {
            return 1;
        }
    }
    return 0;
}

/*
        @name: insert_into_database
        @desc: This function takes in the tokenized values from .csv file, binds
                each one to a prepare statement, and executes every statement.
        @param: *mDb - pointer to the sqlite3 database
        @param: soil_moisture - tokenized soil moisture value from the .csv file
        @param: light - tokenized ambient light value from the .csv file
        @param: temp - tokenized ambient temperature value from the .csv file
        @param: pressure - tokenized barometric pressure value from the .csv file
        @param: precip_prob - tokenized rain probability value from the .csv file
        @param: output - tokenized watering algorithm value from the .csv file
        @param: nodeID - tokenized nodeID value from the .csv file
        @param: battery_lvl - tokenized node battery level value from the .csv file
        @param: hose1 - tokenized first hose output value from the .csv file
        @param: hose2 - tokenized second hose output value from the .csv file
        @param: hose3 - tokenized third hose output value from the .csv file
*/
void insert_into_database(sqlite3 *mDb, double soil_moisture, int light,
        int temp, double pressure, double precip_prob, int output, int nodeID,
        double battery_lvl, int hose1, int hose2, int hose3)
```

```
{
    char *errorMessage;
    sqlite3_exec(mDb, "BEGIN TRANSACTION", NULL, NULL, &errorMessage);

    char buffer[] = "INSERT INTO DATA (Soil_Moisture,Ambient_Light,"
                "Ambient_Temp,Barometric_Pressure,Precip_Prob,Digital_Output,Node_ID,"
                "Battery_Level,Hose_1,Hose_2,Hose_3) VALUES (?1, ?2, ?3, ?4, ?5, ?6, "
                "?7, ?8, ?9, ?10, ?11)";
    sqlite3_stmt * stmt;
    sqlite3_prepare_v2(mDb, buffer, strlen(buffer), &stmt, NULL);

        // binds the values to the prepare statement
    sqlite3_bind_double(stmt, 1, soil_moisture);
    sqlite3_bind_int(stmt, 2, light);
    sqlite3_bind_int(stmt, 3, temp);
    sqlite3_bind_double(stmt, 4, pressure);
    sqlite3_bind_double(stmt, 5, precip_prob);
    sqlite3_bind_int(stmt, 6, output);
    sqlite3_bind_int(stmt, 7, nodeID);
    sqlite3_bind_double(stmt, 8, battery_lvl);
    sqlite3_bind_int(stmt, 9, hose1);
    sqlite3_bind_int(stmt, 10, hose2);
    sqlite3_bind_int(stmt, 11, hose3);

        // error checking to ensure the command was committed to the database
    if (sqlite3_step(stmt) != SQLITE_DONE)
    {
        printf("Commit Failed!\n");
        printf("Error: %s.\n", sqlite3_errmsg(mDb));
    }
    else
    {
        printf("Commit Successful.\n");
    }

    sqlite3_reset(stmt);

        // execute the prepared statements
    sqlite3_exec(mDb, "COMMIT TRANSACTION", NULL, NULL, &errorMessage);
    if (errorMessage != NULL)
    {
        printf("Error: %s\n", errorMessage);
    }
    sqlite3_finalize(stmt);
}

/*
        @name: processCSV
        @desc: This function opens and reads the .csv file, tokenizes the values
                from line 2 inside the csv file, changes the datatypes of the values to
                their proper type, and calls the insert_into_database function, and
                closes the file.
        @param: *db - the sqlite3 database where data is inserted into
*/
void processCSV(sqlite3 *db)
{
    int result, light, temp, output, nodeID, hose1, hose2, hose3;
    double soil_moisture, pressure, precip_prob, battery_lvl;
    char data_param1[DATA_PARAM_SIZE], data_param2[DATA_PARAM_SIZE],
```

```
                 data_param3[DATA_PARAM_SIZE], data_param4[DATA_PARAM_SIZE],
                 data_param5[DATA_PARAM_SIZE], data_param6[DATA_PARAM_SIZE],
                 data_param7[DATA_PARAM_SIZE], data_param8[DATA_PARAM_SIZE],
                 data_param9[DATA_PARAM_SIZE], data_param10[DATA_PARAM_SIZE],
                 data_param11[DATA_PARAM_SIZE];
    char line[256];
    FILE * fp;
    fp = fopen(CSVFILENAME, "r");

    if (fp == NULL)
    {
        fprintf(stderr, "File not found.\n");
    }
        // bypasses the first line of headers
    fgets(line, sizeof(line) - 1, fp);
    while (fgets(line, sizeof(line) - 1, fp) != NULL)
    {
        result = sscanf(line, "%[^','],%[^','],%[^','],%[^','],%[^','],%[^','],"
                        "%[^','],%[^','],%[^','],%[^','],%[^',']",
            data_param1, data_param2, data_param3, data_param4, data_param5,
                        data_param6, data_param7, data_param8, data_param9, data_param10,
                        data_param11);

        if (DEBUG_ON)
        {
            fprintf(stdout, "Result: %d.\n", result);
        }

        if (result == DATA_PARAM_NUM)
        {
            fprintf(stdout, "Line correctly read from csv.\n");
        }
        else
        {
            fprintf(stderr, "Error: Incorrect number of values read from csv.\n");
        }

        soil_moisture = atof(data_param1);
        light = atoi(data_param2);
        temp = atoi(data_param3);
        pressure = atof(data_param4);
        precip_prob = atof(data_param5);
        output = atoi(data_param6);
        nodeID = atoi(data_param7);
        battery_lvl = atof(data_param8);
        hose1 = atoi(data_param9);
        hose2 = atoi(data_param10);
        hose3 = atoi(data_param11);
        //printf("%d\n %d\n %d\n %d\n %d\n %d\n %d\n", i, j, k, l, m, n, o);
        insert_into_database(db, soil_moisture, light, temp, pressure, precip_prob, output, nodeID, battery_lvl, h
    }

    fclose(fp);
}

/*
        @name: createTable
        @desc: Creates the table in the database in which the data will be stored
        @param: *db - the database where the table is created
```

```
*/
int createTable(sqlite3 *db)
{
    int rc;
    char *zErrMsg = 0;

    /*Create SQL statement */
    const char *sql = "CREATE TABLE DATA("
    "ID INTEGER PRIMARY KEY AUTOINCREMENT,"
    "SOIL_MOISTURE REAL,"
    "AMBIENT_LIGHT REAL,"
    "AMBIENT_TEMP REAL,"
    "BAROMETRIC_PRESSURE REAL,"
    "PRECIP_PROB REAL,"
    "DIGITAL_OUTPUT INT,"
    "Node_ID INT,"
    "Battery_Level REAL,"
    "Hose_1 INT,"
    "Hose_2 INT,"
    "Hose_3 INT);";

        //fprintf(stdout,"sql: %s \n",sql);

    /*Execute SQL statement */
    rc = sqlite3_exec(db, sql, callback, 0, &zErrMsg);

        // error checking
    if (rc != SQLITE_OK)
    {
        fprintf(stderr, "SQL error: %s\n", zErrMsg);
        sqlite3_free(zErrMsg);
    }
    return 0;
}

/*
        @name: callback
        @desc: calls for each row after returning from execute statement
        @param:
        @return:
*/
static int callback(void *NotUsed __attribute__((unused)), int argc,
    char **argv, char **azColName)
{
    int i;
    for (i = 0; i < argc; i++)
    {
        printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");
    }
    printf("\n");
    return 0;
}

/*@name: WaterDelivery
   @param: HOSE_NUM - an enum that specifies which hose to evaluate
   @return: uint8_t - a bit array of values that indicate which hoses are on/off
   This function determines if a hose needs to be turned on or off based on sensor data.
   The function also handles the control of the water delivery SM to turn on/off the H-bridge
 */
```

```
uint8_t WaterDelivery(HOSE_NUM HOSE_IN)
{
        // First reset the hose tally
    Hose[HOSE_IN].tally = 0;
    int prevstatus = Hose[HOSE_IN].status;

        // Then need to tally up the digital outs on the hose
    int i, j = 0;
    for (i = 0; i <= MAX_SENSORS; i++)
    {
        // This just shuts down the for loop if the list of sensors is exhausted
        if ((Hose[HOSE_IN].sensors[i] <= 0) || (sd_index == -1))
        {
            break;
        }
        for (j = sd_index; j >= 0; j--)
        {
                // Check if the data item is a sensor mapped to the hose
            if ((sensor_data[j].nodeID == Hose[HOSE_IN].sensors[i]) && (sensor_data[j].nodeID))
            {
                // If it is, increase the tally
                Hose[HOSE_IN].tally += sensor_data[j].digitalOut;
                break;
            }
        }
    }

        // Next check if the tally is above the water level threshold
    if (Hose[HOSE_IN].tally > Hose[HOSE_IN].waterLevel)
    {
        // Check the forecast data
        if (Forecast1.precipProb <= 0.3)
        {
            Hose[HOSE_IN].rainFlag = 0;
                // Go ahead and turn on the water
            Hose[HOSE_IN].status = WATER_ON;
        }
        else
        {
                // Checks if the rain flag is set
                // This prevents the rainTimer from being set more than once
            if (!Hose[HOSE_IN].rainFlag)
            {
                // Sets the rain flag
                Hose[HOSE_IN].rainFlag++;
                // Then the rain timer
                Hose[HOSE_IN].rainTimer = millis();
                // Turns off the hose to wait for the precip prob to take affect
                Hose[HOSE_IN].status = WATER_OFF;
            }
                // If it has been more than 36 hours since the rain timer was set...
            else if (Timer(HOURS_36, Hose[HOSE_IN].rainTimer))
            {
                // ...Go ahead and turn on the water
                Hose[HOSE_IN].status = WATER_ON;
                // Also resets the rain Flag
                Hose[HOSE_IN].rainFlag = 0;
            }
        }
```

```
    }
        // Turn off the hose if the sensors indicate it is not dry enough to water
    else
    {
        Hose[HOSE_IN].status = WATER_OFF;
    }
    printf("Hose %d Water Delivery:\nHose Status: %d;  Prev State = %d\n\n", HOSE_IN, Hose[HOSE_IN].status, prevst
        // Now we actually turn on or off the Hose
    if (prevstatus != Hose[HOSE_IN].status)
    {
        // If statements to control terminal printing
        if (Hose[HOSE_IN].status == WATER_ON)
        {
            printf("Turning ON hose...\n");
        }
        else
        {
            printf("Turning OFF hose...\n");
        }
        // Call the state machine to open the solenoid valve
        while (!WaterDeliverySM(Hose[HOSE_IN].status, FET_DELAY, PULSE_DURATION));
        // More if statments to control terminal printing
        if (Hose[HOSE_IN].status == WATER_ON)
        {
            printf("Hose successfully turned ON\n\n");
        }
        else
        {
            printf("Hose successfully turned OFF\n\n");
        }
    }
        // Create a bit array of hose states to return
    uint8_t hose_status = Hose[2].status *4 + Hose[1].status *2 + Hose[0].status;

    return hose_status;
}

/*@name: LPMOS_Set
   @param: status - whether to turn off or on MOSFET
   @return:void
*/
void LPMOS_Set(uint8_t status)
{
    DEV_Digital_Write(LPMOS_Pin, status);
}

/*@name: RPMOS_Set
   @param: status - whether to turn off or on MOSFET
   @return:void
*/
void RPMOS_Set(uint8_t status)
{
    DEV_Digital_Write(RPMOS_Pin, status);
}

/*@name: LNMOS_Set
   @param: status - whether to turn off or on MOSFET
   @return:void
*/
```

```c
void LNMOS_Set(uint8_t status)
{
    DEV_Digital_Write(LNMOS_Pin, status);
}

/*@name: RNMOS_Set
   @param: status - whether to turn off or on MOSFET
   @return:void
*/
void RNMOS_Set(uint8_t status)
{
    DEV_Digital_Write(RNMOS_Pin, status);
}

/*@name: recordPulses_FS
 *@param: i, determines which flow Sensor to Record
   *Updates Flow Sensor Pulse Count
   @return: return
*/
void recordPulses_FS(int i)
{

    if (i == 0)
    {
        tach_fs[i] = DEV_Digital_Read(FLOW_SENSOR_1_Pin);
    }
    else if (i == 1)
    {
        tach_fs[i] = DEV_Digital_Read(FLOW_SENSOR_2_Pin);
    }
    else if (i == 2)
    {
        tach_fs[i] = DEV_Digital_Read(FLOW_SENSOR_3_Pin);
    }
    else
    {
        printf("Error, Improper Flow Sensor Recorded");
    }

    if (tach_fs[i] != prevTach_fs[i] && tach_fs[i] == 1)
    {
        pulseCount_fs[i] += 1;
        prevTach_fs[i] = tach_fs[i];
        printf(" Pulse Trigger 1 \n ");
    }
    prevTach_fs[i] = tach_fs[i];

    return;
}

/*@name: convertPulse_Liters
   @param: pulseCount - var keeping track of fs pulses
   @return: liters - var keeping track of fs liters
   *
   @return: Liters as a float
*/
float convertPulse_Liters(int pulseCount)
{
    float liters = 0;
```

```
    if (pulseCount < 6500)
    {
        // FS_CAL_A  = 46.2   FS_CAL_B = 40.8
        liters = pulseCount / (FS_CAL_A + (FS_CAL_B* log(pulseCount)));
    }
    else
    {
        liters = pulseCount / FS_CAL_STEADY;    //FS_CAL_STEADY = 404
    }

    return liters;
}

/*@name: convertLiters_Gals
   @param: liters - var keeping track of fs liters
   @param: gallons - var keeping track of fs gallons
   *
   @return: Gallons as a float
*/

float convertLiters_Gals(float liters)
{
    float gallons = 0;

    gallons = liters * LITERS_TO_GAL;   //LITERS_TO_GAL = 0.264172

    return gallons;
}

/*@name: WaterDeliverSM
   @param: status - whether to turn on or off WD
   @param: delayP_N - delay time between turning ON/OFF PFET and NFET
   @param: pulseTime - Time for +/-5V Pulse, Delays time between ON and OFF
   @return: 1/0 depending on whether drive was completed
*/
int WaterDeliverySM(uint8_t status, uint32_t delayP_N, uint32_t pulseTime)
{
    w_State nextState = waterState;     //initialize var to current state
    int hoseSet = FALSE;        // Set to TRUE(1) once done Driving

    switch (waterState)
    {
        case HOSE_IDLE:
                // If the hose is supposed to be turned on
            if (status == WATER_ON)
            {
                nextState = HOSE_ON_S1;
                wTimer = bcm2835_millis();
                hoseSet = 0;
                printf("Leaving Hose Idle: On  \n");
            }
                // If the hose is supposed to be turned off
            else if (status == WATER_OFF)
            {
                nextState = HOSE_OFF_S1;
                wTimer = bcm2835_millis();
                hoseSet = 0;
                printf("Leaving Hose Idle: off  \n");
```

```
            }
            break;

                // Breaks down the function into two parts
                // This first part handles turning on the H-bridge
        case HOSE_ON_S1:
            LNMOS_Set(NMOS_ON);
                //LNMOS_Set(DEMUX_ON);
                // Waits for the P_N delay before moving to the next state
            if (Timer(delayP_N, wTimer))
            {
                wTimer = bcm2835_millis();
                nextState = HOSE_ON_S2;
                printf("Leaving Hose On S1 \n");
            }
            break;

        case HOSE_ON_S2:
            RPMOS_Set(PMOS_ON);
                //RPMOS_Set(DEMUX_ON);
                // Waits for the pulse delay before moving to the next state
            if (Timer(pulseTime, wTimer))
            {
                wTimer = bcm2835_millis();
                nextState = HOSE_ON_S3;
                printf("Leaving Hose On S2 \n");
            }
            break;

        case HOSE_ON_S3:
            RPMOS_Set(PMOS_OFF);
                //RPMOS_Set(DEMUX_OFF);
                // Waits for the P_N delay before moving to the next state
            if (Timer(delayP_N, wTimer))
            {
                wTimer = bcm2835_millis();
                printf("Leaving Hose On S3 \n");
                LNMOS_Set(NMOS_OFF);
                //LNMOS_Set(DEMUX_OFF);
                nextState = HOSE_IDLE;
                hoseSet = 1;
                printf("Leaving Hose On S4 \n");
            }
            break;

                // This second part handles turning off the H-bridge
        case HOSE_OFF_S1:
            RNMOS_Set(NMOS_ON);
                //RNMOS_Set(DEMUX_ON);
                // Waits for the P_N delay before moving to the next state
            if (Timer(delayP_N, wTimer))
            {
                wTimer = bcm2835_millis();
                nextState = HOSE_OFF_S2;
                printf("Leaving Hose Off S1 \n");
            }
            break;

        case HOSE_OFF_S2:
```

```
            LPMOS_Set(PMOS_ON);
                //LPMOS_Set(DEMUX_ON);
            if (Timer(pulseTime, wTimer))
            {
                wTimer = bcm2835_millis();
                nextState = HOSE_OFF_S3;
                printf("Leaving Hose Off S2 \n");
            }
            break;

        case HOSE_OFF_S3:
            LPMOS_Set(PMOS_OFF);
                //LPMOS_Set(DEMUX_OFF);
                // Waits for the P_N delay before moving to the next state
            if (Timer(delayP_N, wTimer))
            {
                wTimer = bcm2835_millis();
                printf("Leaving Hose Off S3 \n");
                RNMOS_Set(NMOS_OFF);
                //RNMOS_Set(DEMUX_OFF);
                nextState = HOSE_IDLE;
                hoseSet = 1;
                printf("Leaving Hose Off S4 \n");
            }
            break;
    }
    waterState = nextState;
    return hoseSet;     //1 if set, 0 if still in S1-4

}

/**
   @Function LCD_PrintArrow(int state)
   @param int x, int y Used to determine x,y position of arrow
   @return None
   @brief This function prints Arrow on OLED at x,y coordinates
   @note
   @author Brian Naranjo, 1/25/20
   @editor   */

void OLED_PrintArrow(int x, int y)
{
    print_String(x, y, (const uint8_t *)
        "<", FONT_5X8);
}
/*@name: OLED_SM
   @param: Color of Page Text
   @return: void
*/

void OLED_SM(uint16_t color)
{
    int16_t temp_x, temp_y = 0;
    int i, j = 0;
    int element_Changed = FALSE;
    int arrowOptions = 0;

    Set_Color(color);

```

```
    if (Timer(sleepTime, oledSleepTimer))
    {
        //If Sleep Timer Expires, return to SLEEP
        arrowState = 0;
        nextPage = SLEEP;
        Clear_Screen();
        oledSleepTimer = bcm2835_millis();      //reset sleep timer after transition to Sleep
    }

    oledState = nextPage;       //Transition to next state
    prevArrowState = arrowState;        //Save arrow State

        //Toggle Arrow
    if (DOWN_PRESSED)
    {
        //if down, increment arrowstate.
        if (arrowState >= 3)
        {
            arrowState = 0;
        }
        else
        {
            arrowState++;
        }
        oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
    }
    else if (UP_PRESSED)
    {
        if (arrowState <= 0)
        {
                //otherwise, decrement arrowstate.
            arrowState = 3;
        }
        else
        {
            arrowState--;
        }
        oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
    }

    switch (oledState)
    {
        case WELCOME_PAGE:
            print_String(24, 25, (const uint8_t *)
                "Welcome To ", FONT_8X16);      //Print Home Page
            print_String(30, 55, (const uint8_t *)
                "Intuitive", FONT_8X16);
            print_String(8, 85, (const uint8_t *)
                "Auto Irrigation", FONT_8X16);

            if (Timer(EIGHT_SECONDS, oledSleepTimer) || ENTER_PRESSED)
            {
                //If Sleep Timer Expires, return to SLEEP
                arrowState = 0; //Reset Arrow State
                nextPage = HOME_PAGE;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after transition to Sleep
            }
            break;
```

```
        case SLEEP:
            print_String(35, 55, (const uint8_t *)
                "SLEEPING", FONT_8X16);
            print_String(35, 85, (const uint8_t *) timeBuffer, FONT_8X16);

            if (ENTER_PRESSED)
            {
                nextPage = HOME_PAGE;
                arrowState = 0; //Reset Arrow State
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            break;

        case HOME_PAGE:
            if (prevArrowState != arrowState)
            {
                //Update Screen if arrowState changes
                Clear_Screen();
            }

            print_String(0, 0, (const uint8_t *)
                "Home Page", FONT_8X16);
            print_String(0, 30, (const uint8_t *)
                "Sensor Data", FONT_5X8);
            print_String(0, 45, (const uint8_t *)
                "Hose Configuration", FONT_5X8);
            print_String(0, 60, (const uint8_t *)
                "Settings", FONT_5X8);
            print_String(35, 95, (const uint8_t *) timeBuffer, FONT_8X16);

                //Update Oled Printing
            if (arrowState == 0)
            {
                OLED_PrintArrow(70, 30);
            }
            else if (arrowState == 1)
            {
                OLED_PrintArrow(113, 45);
            }
            else
            {
                OLED_PrintArrow(55, 60);
            }

            if (ENTER_PRESSED)
            {
                //Enter Page Corresponding to Arrow State
                if (arrowState == 0)
                {
                    nextPage = SENSORS_HOME;
                }
                else if (arrowState == 1)
                {
                    nextPage = HOSES_HOME;
                }
                else
                {
```

```
                    nextPage = SETTINGS_HOME;
                }
                arrowState = 0; //Reset Arrow State
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press

            }
            else if (BACK_PRESSED)
            {
                nextPage = SLEEP;
                arrowState = 0; //Reset Arrow State
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            break;

        case SENSORS_HOME:
            if (prevArrowState != arrowState)
            {
                //Update Screen if arrowState changes
                Clear_Screen();
            }

            print_String(0, 0, (const uint8_t *)
                "Sensors Home", FONT_8X16);

            print_String(0, 30, (const uint8_t *)
                "Connected Sensors", FONT_5X8);
            print_String(0, 45, (const uint8_t *)
                "Current Readings", FONT_5X8);
            print_String(0, 60, (const uint8_t *)
                "Plot Sensor Data", FONT_5X8);
            print_String(35, 95, (const uint8_t *) timeBuffer, FONT_8X16);

            if (arrowState == 0)
            {
                OLED_PrintArrow(110, 30);
            }
            else if (arrowState == 1)
            {
                OLED_PrintArrow(110, 45);
            }
            else
            {
                OLED_PrintArrow(110, 60);
            }

            if (ENTER_PRESSED)
            {
                Clear_Screen();
                if (arrowState == 0)
                {
                    nextPage = SENSORS_LIST;
                }
                else if (arrowState == 1)
                {
                    nextPage = SENSORS_CURRENT;
                }
                else
```

```
                {
                    nextPage = SENSORS_PLOT_START;
                }
                arrowState = 0;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            else if (BACK_PRESSED)
            {
                nextPage = HOME_PAGE;
                arrowState = 0;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            break;

        case SENSORS_LIST:
            if (prevArrowState != arrowState)
            {
                //Update Screen if arrowState changes
                Clear_Screen();
            }

            print_String(0, 0, (const uint8_t *)
                "Sensors List", FONT_8X16);

            temp_x = 0;
            temp_y = 30;

            for (i = 0; i < mesh.addrListTop; i++)
            {
                //Prints all Connected Sensors
                // Add sensor nodes to the list of sensors mapped to the hose
                sprintf(sensorIDBuffer, "Sensor Node %d", mesh.addrList[i].nodeID);
                print_String(temp_x, temp_y, (const uint8_t *) sensorIDBuffer, FONT_5X8);
                temp_y += 15;
            }

            if (arrowState == 0)
            {
                //Update Arrow State
                OLED_PrintArrow(110, 30);
            }
            else
            {
                OLED_PrintArrow(110, 45);
            }

            if (ENTER_PRESSED)
            {
                nextPage = SENSORS_CURRENT;     //Go into Current Sensors Menu
                selected_Node = 2;      //Set selected Node for printing
                arrowState = 0;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press

            }
            else if (BACK_PRESSED)
            {
```

```
                nextPage = SENSORS_HOME;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            break;

        case SENSORS_CURRENT:
            if (prevArrowState != arrowState)
            {
                Clear_Screen();
            }

            if (new_Data)
            {
                Clear_Screen();
                new_Data = FALSE;
            }
            print_String(0, 0, (const uint8_t *)
                "Current Data", FONT_8X16);
            print_String(0, 30, (const uint8_t *)
                "Node ID:", FONT_5X8);
            print_String(0, 45, (const uint8_t *)
                "Moisture(%):", FONT_5X8);
            print_String(0, 60, (const uint8_t *)
                "Light(%):", FONT_5X8);
            print_String(0, 75, (const uint8_t *)
                "Temp(C):", FONT_5X8);

            if (selected_Node == 2)
            {
                //Store Struct Variables as Strings
                convertFloat_String(current_Dat_1.soilMoisture, currentBuffer1);
                convertFloat_String(current_Dat_1.lightLevel, currentBuffer2);
                sprintf(currentBuffer3, "%d", current_Dat_1.temp_C);
                sprintf(currentBuffer4, "%d", current_Dat_1.nodeID);
            }
            else if (selected_Node == 5)
            {
                convertFloat_String(current_Dat_2.soilMoisture, currentBuffer1);
                convertFloat_String(current_Dat_2.lightLevel, currentBuffer2);
                sprintf(currentBuffer3, "%d", current_Dat_2.temp_C);
                sprintf(currentBuffer4, "%d", current_Dat_2.nodeID);
            }
            else if (selected_Node == 4)
            {
                convertFloat_String(current_Dat_3.soilMoisture, currentBuffer1);
                convertFloat_String(current_Dat_3.lightLevel, currentBuffer2);
                sprintf(currentBuffer3, "%d", current_Dat_3.temp_C);
                sprintf(currentBuffer4, "%d", current_Dat_3.nodeID);
            }

                //Print Variable Strings
            print_String(55, 30, (const uint8_t *) currentBuffer4, FONT_5X8);
            print_String(72, 45, (const uint8_t *) currentBuffer1, FONT_5X8);
            print_String(60, 60, (const uint8_t *) currentBuffer2, FONT_5X8);
            print_String(55, 75, (const uint8_t *) currentBuffer3, FONT_5X8);

            if (arrowState == 0)
            {
```

```
            OLED_PrintArrow(65, 30);
        }
        else if (arrowState == 1)
        {
            OLED_PrintArrow(115, 45);
        }
        else if (arrowState == 2)
        {
            OLED_PrintArrow(110, 60);
        }
        else
        {
            OLED_PrintArrow(70, 75);
        }

        if (ENTER_PRESSED)
        {
            if (arrowState == 0)
            {
                if (selected_Node == 2)
                {
                    selected_Node = 5;
                }
                else if (selected_Node == 5)
                {
                    selected_Node = 4;
                }
                else
                {
                    selected_Node = 2;
                }
                new_Data = TRUE;
            }
            else if (arrowState == 1)
            {
                dataType = MOISTURE;
                nextPage = SENSORS_PLOT;
            }
            else if (arrowState == 2)
            {
                dataType = SUNLIGHT;
                nextPage = SENSORS_PLOT;
            }
            else
            {
                dataType = TEMP;
                nextPage = SENSORS_PLOT;
            }
            arrowState = 0;
            Clear_Screen();
            oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
        }
        else if (BACK_PRESSED)
        {
            nextPage = SENSORS_HOME;
            Clear_Screen();
            oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
        }
        break;
```

```
        case SENSORS_PLOT_START:
            if (prevArrowState != arrowState)
            {
                Clear_Screen();
            }
            print_String(0, 0, (const uint8_t *)
                "Plot Sensor Data", FONT_8X16);
            print_String(0, 30, (const uint8_t *)
                "Moisture", FONT_5X8);
            print_String(0, 45, (const uint8_t *)
                "Sunlight", FONT_5X8);
            print_String(0, 60, (const uint8_t *)
                "Temperature", FONT_5X8);

            if (arrowState == 0)
            {
                OLED_PrintArrow(110, 30);
            }
            else if (arrowState == 1)
            {
                OLED_PrintArrow(110, 45);
            }
            else
            {
                OLED_PrintArrow(110, 60);
            }

            if (ENTER_PRESSED)
            {
                Clear_Screen();
                if (arrowState == 2)
                {
                    dataType = TEMP;
                }
                else if (arrowState == 1)
                {
                    dataType = SUNLIGHT;
                }
                else
                {
                    dataType = MOISTURE;
                }
                nextPage = SENSORS_PLOT;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press

            }
            else if (BACK_PRESSED)
            {
                nextPage = SENSORS_HOME;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            break;

        case SENSORS_PLOT:

                //Set Up Grid for Printing
```

```
            printGrid(20, 120, 20, 120, 10, 10);
            printAxesLabels(0, 115);

                //Plot Corresponding to Sensor Node
            if (selected_Node == 2)
            {
                plotSampleData(sensor1_data, dataType, MAX_ELEMENTS);
            }
            else if (selected_Node == 5)
            {
                plotSampleData(sensor2_data, dataType, MAX_ELEMENTS);
            }
            else if (selected_Node == 4)
            {
                plotSampleData(sensor3_data, dataType, MAX_ELEMENTS);
            }
            else
            {
                plotSampleData(sensor_data, dataType, MAX_ELEMENTS);
            }

            if (ENTER_PRESSED)
            {
                //Update Sensor Struct for Plotting
                if (selected_Node == 2)
                {
                    selected_Node = 5;
                }
                else if (selected_Node == 5)
                {
                    selected_Node = 4;
                }
                else
                {
                    selected_Node = 2;
                }

                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            else if (BACK_PRESSED)
            {
                nextPage = SENSORS_PLOT_START;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }

            break;

        case HOSES_HOME:
            if (prevArrowState != arrowState)
            {
                Clear_Screen();
            }

            print_String(0, 0, (const uint8_t *)
                "Hoses", FONT_8X16);

            print_String(0, 30, (const uint8_t *)
```

```
                "Current Hose Status", FONT_5X8);
            print_String(0, 45, (const uint8_t *)
                "Hose Control", FONT_5X8);
            print_String(0, 60, (const uint8_t *)
                "Watering Log", FONT_5X8);
            print_String(0, 75, (const uint8_t *)
                "Map Sensors ", FONT_5X8);
            print_String(35, 95, (const uint8_t *) timeBuffer, FONT_8X16);

            if (arrowState == 0)
            {
                //Update Arrow
                OLED_PrintArrow(115, 30);
            }
            else if (arrowState == 1)
            {
                OLED_PrintArrow(80, 45);
            }
            else if (arrowState == 2)
            {
                OLED_PrintArrow(80, 60);
            }
            else
            {
                OLED_PrintArrow(85, 75);
            }

            if (ENTER_PRESSED)
            {
                //Menu traversal based on Arrowstate
                if (arrowState == 0)
                {
                    nextPage = HOSES_STATUS;
                }
                else if (arrowState == 1)
                {
                    nextPage = HOSES_CONTROL;
                }
                else if (arrowState == 2)
                {
                    nextPage = HOSES_WATER;
                }
                else
                {
                    nextPage = HOSES_MAP;
                }
                arrowState = 0;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            else if (BACK_PRESSED)
            {
                nextPage = HOME_PAGE;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            break;

        case HOSES_STATUS:
```

```
            if (prevArrowState != arrowState)
            {
                Clear_Screen();
            }

            if (hose_statuses != prev_hose_statuses)
            {
                Clear_Screen();
            }

            print_String(0, 0, (const uint8_t *)
                "Hoses Status", FONT_8X16);
            printHoseStatus(0, 40, hose_statuses);      //Print OFF/ON for each hose
            prev_hose_statuses = hose_statuses; //Store statuses for OLED updating

                // Print Connected Sensors
            temp_x = 90;
            temp_y = 40;

                //Update Hose 0 Nodes
            for (i = 0; i < hose0_elements; i++)
            {
                //Iterate and print connected node IDs
                sprintf(intBuffer, "%d", Hose[HOSE0].sensors[hose0_elements]);
                print_String(temp_x, temp_y, (const uint8_t *) intBuffer, FONT_5X8);
                temp_x += 10;
            }

            temp_x = 90;
            temp_y = 55;

                //Update Hose 1 Nodes
            for (i = 0; i < hose1_elements; i++)
            {
                //Iterate and print connected node IDs
                sprintf(intBuffer, "%d", Hose[HOSE1].sensors[hose1_elements]);
                print_String(temp_x, temp_y, (const uint8_t *) intBuffer, FONT_5X8);
                temp_x += 10;
            }

            temp_x = 90;
            temp_y = 70;

                //Update Hose 2 Nodes
            for (i = 0; i < hose2_elements; i++)
            {
                //Iterate and print connected node IDs
                sprintf(intBuffer, "%d", Hose[HOSE2].sensors[hose2_elements]);
                print_String(temp_x, temp_y, (const uint8_t *) intBuffer, FONT_5X8);
                temp_x += 10;
            }

            if (arrowState == 0)
            {
                OLED_PrintArrow(100, 40);
            }
            else if (arrowState == 1)
            {
                OLED_PrintArrow(100, 55);
```

```
            }
            else
            {
                OLED_PrintArrow(100, 70);
            }

            if (ENTER_PRESSED)
            {
                nextPage = HOSES_CONTROL;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            else if (BACK_PRESSED)
            {
                nextPage = HOSES_HOME;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            break;

        case HOSES_CONTROL:
            if (prevArrowState != arrowState)
            {
                Clear_Screen();
            }

            if (element_Changed == TRUE)
            {
                //Update screen only once change is made
                Clear_Screen();
                element_Changed = FALSE;
            }

            print_String(0, 0, (const uint8_t *)
                "Hose Control", FONT_8X16);

            temp_x = 0; //Initialize Starting Print Locations
            temp_y = 40;

            for (i = 0; i < NUM_HOSES; i++)
            {
                //Iterate and print hose control status

                if (Hose[i].control == AUTOMATIC)
                {
                    sprintf(hoseBuffer, "Hose %d: AUTO", (i + 1));
                    print_String(temp_x, temp_y, (const uint8_t *) hoseBuffer, FONT_5X8);
                }
                else if (Hose[i].control == ON)
                {
                    sprintf(hoseBuffer, "Hose %d: ON", (i + 1));
                    print_String(temp_x, temp_y, (const uint8_t *) hoseBuffer, FONT_5X8);
                }
                else
                {
                    sprintf(hoseBuffer, "Hose %d: OFF", (i + 1));
                    print_String(temp_x, temp_y, (const uint8_t *) hoseBuffer, FONT_5X8);
                }

```

```
                temp_y += 15;   //Increment 15 to move to next row

            }

            if (arrowState == 0)
            {
                OLED_PrintArrow(100, 40);
            }
            else if (arrowState == 1)
            {
                OLED_PrintArrow(100, 55);
            }
            else
            {
                OLED_PrintArrow(100, 70);
            }

            i = 0;      //initialize index variable
            if (ENTER_PRESSED)
            {
                if (arrowState <= 1)
                {
                    i = arrowState;
                }
                else
                {
                    i = 2;      //else store as last element
                }

                if (Hose[i].control == AUTOMATIC)
                {
                    Hose[i].control = OFF;
                }
                else if (Hose[i].control == OFF)
                {
                    Hose[i].control = ON;
                }
                else
                {
                    Hose[i].control = AUTOMATIC;
                }

                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            else if (BACK_PRESSED)
            {
                nextPage = HOSES_HOME;
                arrowState = 0;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            break;

        case HOSES_WATER:
            if (prevArrowState != arrowState)
            {
                Clear_Screen();
            }
```

```
            for (i = 0; i > 3; i++)
            {
                //Update Screen if new data received
                if (prev_Liters_fs[i] != water_liters_fs[i])
                {
                    Clear_Screen();
                }
            }

            print_String(0, 0, (const uint8_t *)
                "Watering Log", FONT_8X16);

                //Convert Floats containing amount of water in liters into strings
                //tempVal = Hose0.waterLevel;
            convertFloat_String(water_liters_fs[0], hoseBuffer);
            print_String(0, 40, (const uint8_t *)
                "Hose 1:", FONT_5X8);
            print_String(50, 40, (const uint8_t *) hoseBuffer, FONT_5X8);
            print_String(90, 40, (const uint8_t *)
                "L", FONT_5X8);

                //tempVal = Hose1.waterLevel;
                //tempVal = 3;
            convertFloat_String(water_liters_fs[1], hoseBuffer);
            print_String(0, 55, (const uint8_t *)
                "Hose 2:", FONT_5X8);
            print_String(50, 55, (const uint8_t *) hoseBuffer, FONT_5X8);
            print_String(90, 55, (const uint8_t *)
                "L", FONT_5X8);

                //tempVal = 5;  //used for testing
                // sprintf(hoseBuffer,"%d L", tempVal);

            convertFloat_String(water_liters_fs[2], hoseBuffer);
            print_String(0, 70, (const uint8_t *)
                "Hose 3:", FONT_5X8);
            print_String(50, 70, (const uint8_t *) hoseBuffer, FONT_5X8);
            print_String(90, 70, (const uint8_t *)
                "L", FONT_5X8);

                //tempVal = Hose2.waterLevel + Hose1.waterLevel + Hose0.waterLevel;
                //tempVal = 8;  //used for testing
                //sprintf(hoseBuffer,"%d L", tempVal);

            convertFloat_String((water_liters_fs[0] + water_liters_fs[1] + water_liters_fs[2]), hoseBuffer);
            print_String(0, 85, (const uint8_t *)
                "Total:", FONT_5X8);
            print_String(50, 85, (const uint8_t *) hoseBuffer, FONT_5X8);
            print_String(90, 85, (const uint8_t *)
                "L", FONT_5X8);

            for (i = 0; i < 3; i++)
            {
                prev_Liters_fs[i] = water_liters_fs[i]; //Save previous readings
            }

            if (arrowState == 0)
            {
```

```
                //Update Arrow State
                OLED_PrintArrow(100, 40);
            }
            else if (arrowState == 1)
            {
                OLED_PrintArrow(100, 55);
            }
            else
            {
                OLED_PrintArrow(100, 70);
            }

            if (ENTER_PRESSED)
            {
                nextPage = SLEEP;
                arrowState = 0;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            else if (BACK_PRESSED)
            {
                nextPage = HOSES_HOME;
                arrowState = 0;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            break;

        case HOSES_MAP:
            if (prevArrowState != arrowState)
            {
                Clear_Screen();
            }

            print_String(0, 0, (const uint8_t *)
                "Map Sensors", FONT_8X16);

            temp_x = 0;
            temp_y = 30;

            arrowOptions = 0;   //Initialize Arrow Options
            nodeUnmapped = FALSE;       //Initialize flag
            j = 0;      //Initialize Secondary Index

            for (i = 0; i < mesh.addrListTop; i++)
            {
                // Add sensor nodes to the list of sensors mapped to the hose
                if (mesh.addrList[i].nodeID != mappedSensors[i])
                {
                        //Check if mapped
                    sprintf(sensorIDBuffer, "Sensor Node %d", mesh.addrList[i].nodeID);
                    print_String(temp_x, temp_y, (const uint8_t *) sensorIDBuffer, FONT_5X8);
                    unmappedSensors[j] = i;     //Save Array Index for Selected Sensor
                    arrowOptions += 1;  // Increment amount of arrow options
                    nodeUnmapped = TRUE;        //Set Flag to true
                    temp_y += 15;       //Increment to new line
                    j += 1;     //Increment to next element
                }
            }
```

```
            if (!nodeUnmapped)
            {
                //if no Sensors left to map
                print_String(0, 45, (const uint8_t *)
                    "No Sensors to Map", FONT_5X8);
            }

            /*
            if (nodeMapState == 0){
              print_String(0,45, (const uint8_t*)"Sensor Node 2", FONT_5X8);
              print_String(0,60, (const uint8_t*)"Sensor Node 5", FONT_5X8);
              print_String(0,75, (const uint8_t*)"Sensor Node 4", FONT_5X8);
            } else if (nodeMapState == 1){
              print_String(0,45, (const uint8_t*)"Sensor Node 5", FONT_5X8);
              print_String(0,60, (const uint8_t*)"Sensor Node 4", FONT_5X8);
            } else if (nodeMapState == 2){
              print_String(0,45, (const uint8_t*)"Sensor Node 5", FONT_5X8);
            } else {
              print_String(0,60, (const uint8_t*)"Back", FONT_5X8);
              print_String(0,45, (const uint8_t*)"No Sensors to Map", FONT_5X8);
            }

            */

            if (arrowOptions == 4)
            {
                //Update Arrow State Corresponding to # of connected sensors
                if (arrowState == 0)
                {
                    OLED_PrintArrow(90, 45);
                }
                else if (arrowState == 1)
                {
                    OLED_PrintArrow(90, 60);
                }
                else if (arrowState == 2)
                {
                    OLED_PrintArrow(90, 75);
                }
                else
                {
                    OLED_PrintArrow(50, 90);
                }
            }
            else if (arrowOptions == 3)
            {
                if (arrowState == 0)
                {
                    OLED_PrintArrow(90, 45);
                }
                else if (arrowState == 1)
                {
                    OLED_PrintArrow(90, 60);
                }
                else
                {
                    OLED_PrintArrow(50, 75);
                }
```

```
            }
            else if (arrowOptions == 2)
            {
                if (arrowState == 0)
                {
                    OLED_PrintArrow(90, 45);
                }
                else if (arrowState == 1)
                {
                    OLED_PrintArrow(90, 60);
                }
                else
                {
                    OLED_PrintArrow(50, 75);
                }
            }
            else if (arrowOptions == 3)
            {
                if (arrowState == 0)
                {
                    OLED_PrintArrow(90, 45);
                }
                else if (arrowState == 1)
                {
                    OLED_PrintArrow(90, 60);
                }
                else
                {
                    OLED_PrintArrow(50, 75);
                }
            }
            else if (arrowOptions == 3)
            {
                if (arrowState == 0 || arrowState == 2)
                {
                    OLED_PrintArrow(90, 45);
                }
                else
                {
                    OLED_PrintArrow(50, 60);
                }
            }
            else if (arrowOptions == 1)
            {
                OLED_PrintArrow(90, 45);
            }

            if (ENTER_PRESSED)
            {
                sensorToMap = unmappedSensors[arrowState];      //Transition to corresponding arrowState
                nextPage = HOSES_MAP_SELECT;    //Transition to Map Select

                if (!nodeUnmapped)
                {
                        //If There are no more nodes to map
                    nextPage = HOSES_HOME;
                }

                arrowState = 0; //reset arrow State
```

```
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press

            }
            else if (BACK_PRESSED)
            {
                nextPage = SENSORS_HOME;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            break;

        case HOSES_MAP_SELECT:  //page allowing for selection of hoses to map sensor to
            if (prevArrowState != arrowState)
            {
                Clear_Screen();
            }

            print_String(0, 0, (const uint8_t *)
                "Select Hose", FONT_8X16);

            print_String(0, 45, (const uint8_t *)
                "Hose 1", FONT_5X8);
            print_String(0, 60, (const uint8_t *)
                "Hose 2", FONT_5X8);
            print_String(0, 75, (const uint8_t *)
                "Hose 3", FONT_5X8);

            if (mesh.addrList[sensorToMap].nodeID == 2)
            {
                print_String(45, 30, (const uint8_t *)
                    "(Node 2)", FONT_5X8);
            }
            else if (mesh.addrList[sensorToMap].nodeID == 2)
            {
                print_String(45, 30, (const uint8_t *)
                    "(Node 4)", FONT_5X8);
            }
            else
            {
                print_String(45, 30, (const uint8_t *)
                    "(Node 5)", FONT_5X8);
            }

            if (arrowState == 0)
            {
                OLED_PrintArrow(50, 45);
            }
            else if (arrowState == 1)
            {
                OLED_PrintArrow(50, 60);
            }
            else
            {
                OLED_PrintArrow(50, 75);
            }

            if (ENTER_PRESSED)
            {
```

```
                //If Enter Pressed, Save Hose Sensors within Struct
                if (arrowState == 0)
                {
                    Hose[HOSE0].sensors[hose0_elements] = mesh.addrList[sensorToMap].nodeID;
                    hose0_elements++;
                }
                else if (arrowState == 1)
                {
                    Hose[HOSE1].sensors[hose1_elements] = mesh.addrList[sensorToMap].nodeID;
                    hose1_elements++;
                }
                else
                {
                    Hose[HOSE2].sensors[hose2_elements] = mesh.addrList[sensorToMap].nodeID;
                    hose2_elements++;
                }

                //Store List of Mapped Sensor Node ID's corresponding to Mesh Array Index
                mappedSensors[sensorToMap] = mesh.addrList[sensorToMap].nodeID;

                arrowState = 0;
                nextPage = HOSES_MAP;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press

            }
            else if (BACK_PRESSED)
            {
                nextPage = HOSES_MAP;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }

            break;

        case SETTINGS_HOME:

            if (prevArrowState != arrowState)
            {
                Clear_Screen();
            }

                //Print Settings Menu
            print_String(0, 0, (const uint8_t *)
                "Settings", FONT_8X16);

            print_String(0, 30, (const uint8_t *)
                "Adjust Sleep Timer", FONT_5X8);
            print_String(0, 45, (const uint8_t *)
                "Calibrate Sensors", FONT_5X8);
            print_String(0, 60, (const uint8_t *)
                "Change Menu Color", FONT_5X8);
            print_String(0, 75, (const uint8_t *)
                "Reset System", FONT_5X8);
            print_String(35, 95, (const uint8_t *) timeBuffer, FONT_8X16);

                //Update Arrow Print statements depending on arrowState
            if (arrowState == 0)
            {
```

```
                OLED_PrintArrow(112, 30);
            }
            else if (arrowState == 1)
            {
                OLED_PrintArrow(105, 45);
            }
            else if (arrowState == 2)
            {
                OLED_PrintArrow(112, 60);
            }
            else
            {
                OLED_PrintArrow(85, 75);
            }

            if (ENTER_PRESSED)
            {
                //Traversal to Page
                if (arrowState == 0)
                {
                    nextPage = SETTINGS_SLEEP;
                }
                else if (arrowState == 1)
                {
                    nextPage = SETTINGS_CAL;
                }
                else if (arrowState == 2)
                {
                    nextPage = SETTINGS_COLOR;
                }
                else
                {
                    nextPage = SETTINGS_RESET;
                }
                arrowState = 0;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            else if (BACK_PRESSED)
            {
                // Return to Home
                nextPage = HOME_PAGE;
                arrowState = 0;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            break;

        case SETTINGS_SLEEP:
            if (prevArrowState != arrowState)
            {
                Clear_Screen();
            }

            print_String(0, 0, (const uint8_t *)
                "Sleep Settings", FONT_8X16);

                //Print out current Sleep TIme
            print_String(0, 30, (const uint8_t *)
```

```
        "Current:", FONT_5X8);
    if (sleepTime == 1)
    {
        sprintf(intBuffer, "%d Min", (sleepTime / MIN_1));
    }
    else
    {
        sprintf(intBuffer, "%d Mins", (sleepTime / MIN_1));
    }

        //Sleep Menu Options
    print_String(70, 30, (const uint8_t *) intBuffer, FONT_5X8);
    print_String(0, 50, (const uint8_t *)
        "1 Minute", FONT_5X8);
    print_String(0, 65, (const uint8_t *)
        "3 Minutes", FONT_5X8);
    print_String(0, 80, (const uint8_t *)
        "5 Minutes", FONT_5X8);
    print_String(0, 95, (const uint8_t *)
        "SLEEP", FONT_5X8);

        //Update Oled States
    if (arrowState == 0)
    {
        OLED_PrintArrow(65, 50);
    }
    else if (arrowState == 1)
    {
        OLED_PrintArrow(65, 65);
    }
    else if (arrowState == 2)
    {
        OLED_PrintArrow(65, 80);
    }
    else
    {
        OLED_PrintArrow(50, 95);
    }

    if (ENTER_PRESSED)
    {
        //Set Timers corresponding to arrowState
        if (arrowState == 0)
        {
            sleepTime = MIN_1;
        }
        else if (arrowState == 1)
        {
            sleepTime = MIN_3;
        }
        else if (arrowState == 2)
        {
            sleepTime = MIN_5;
        }
        else
        {
            nextPage = SLEEP;
            arrowState = 0;
        }
```

```
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            else if (BACK_PRESSED)
            {
                nextPage = SETTINGS_HOME;
                arrowState = 0;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            break;

        case SETTINGS_CAL:
            if (prevArrowState != arrowState)
            {
                Clear_Screen();
            }

            if (new_Data)
            {
                //Updates each time a message is received
                Clear_Screen();
                new_Data = FALSE;
            }

                //Menu Used for Calibration
            print_String(0, 0, (const uint8_t *)
                "Sensor Recal", FONT_8X16);
            print_String(0, 30, (const uint8_t *)
                "Node ID:", FONT_8X16);
            print_String(0, 50, (const uint8_t *)
                "Moisture(%):", FONT_5X8);
            print_String(0, 80, (const uint8_t *)
                "Prev Threshold(%):", FONT_5X8);
            print_String(0, 100, (const uint8_t *)
                "Set as ", FONT_5X8);
            print_String(0, 110, (const uint8_t *)
                "New Threshold ", FONT_5X8);

                //Check corresponding node Ids
                //Print current soil Moisture reading
            if (selected_Node == 2)
            {
                //Store Struct Variables as Strings
                convertFloat_String(current_Dat_1.soilMoisture, currentBuffer1);
                sprintf(currentBuffer4, "%d", current_Dat_1.nodeID);
                convertFloat_String(moisture_s_thresh[0], currentBuffer5);
            }
            else if (selected_Node == 5)
            {
                convertFloat_String(current_Dat_2.soilMoisture, currentBuffer1);
                sprintf(currentBuffer4, "%d", current_Dat_2.nodeID);
                convertFloat_String(moisture_s_thresh[1], currentBuffer5);
            }
            else if (selected_Node == 4)
            {
                convertFloat_String(current_Dat_3.soilMoisture, currentBuffer1);
                sprintf(currentBuffer4, "%d", current_Dat_3.nodeID);
                convertFloat_String(moisture_s_thresh[2], currentBuffer5);
```

```
            }

                //Print Variable Strings
            print_String(70, 30, (const uint8_t *) currentBuffer4, FONT_8X16);
            print_String(10, 60, (const uint8_t *) currentBuffer1, FONT_5X8);
            print_String(10, 90, (const uint8_t *) currentBuffer5, FONT_5X8);

            if (arrowState == 0 || arrowState == 2)
            {
                OLED_PrintArrow(95, 33);
            }
            else
            {
                OLED_PrintArrow(85, 110);
            }

                //Update Selected Node on Enter being pressed
            if (ENTER_PRESSED)
            {
                if (arrowState == 0 || arrowState == 2)
                {
                    if (selected_Node == 2)
                    {
                        selected_Node = 5;
                    }
                    else if (selected_Node == 5)
                    {
                        selected_Node = 4;
                    }
                    else
                    {
                        selected_Node = 2;
                    }
                    new_Data = TRUE;
                    arrowState = 0;
                }
                else
                {
                    if (selected_Node == 2)
                    {
                        moisture_s_thresh[0] = current_Dat_1.soilMoisture;
                    }
                    else if (selected_Node == 5)
                    {
                        moisture_s_thresh[1] = current_Dat_2.soilMoisture;
                    }
                    else
                    {
                        moisture_s_thresh[2] = current_Dat_3.soilMoisture;
                    }
                    new_Data = TRUE;
                }
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            else if (BACK_PRESSED)
            {
                nextPage = SETTINGS_HOME;
                Clear_Screen();
```

```
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            break;

        case SETTINGS_COLOR:

            if (prevArrowState != arrowState)
            {
                Clear_Screen();
            }

                //Present Color Settings with available options
            print_String(0, 0, (const uint8_t *)
                "Color Settings", FONT_8X16);

            print_String(0, 30, (const uint8_t *)
                "White", FONT_5X8);
            print_String(0, 45, (const uint8_t *)
                "Blue", FONT_5X8);
            print_String(0, 60, (const uint8_t *)
                "Green", FONT_5X8);
            print_String(0, 75, (const uint8_t *)
                "Red", FONT_5X8);

            if (arrowState == 0)
            {
                OLED_PrintArrow(40, 30);
            }
            else if (arrowState == 1)
            {
                OLED_PrintArrow(40, 45);
            }
            else if (arrowState == 2)
            {
                OLED_PrintArrow(40, 60);
            }
            else
            {
                OLED_PrintArrow(40, 75);
            }

                //Change color corresponding to arrowState on Enter Press
            if (ENTER_PRESSED)
            {
                if (arrowState == 0)
                {
                    oledColor = WHITE;
                }
                else if (arrowState == 1)
                {
                    oledColor = BLUE;
                }
                else if (arrowState == 2)
                {
                    oledColor = GREEN;
                }
                else
                {
                    oledColor = RED;
```

```
                }
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            else if (BACK_PRESSED)
            {
                nextPage = SETTINGS_HOME;       //Return to Settings Page
                arrowState = 0;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            break;

        case SETTINGS_RESET:
            if (prevArrowState != arrowState)
            {
                Clear_Screen();
            }

                //Print Reset Statement
            print_String(0, 0, (const uint8_t *)
                "System Reset", FONT_8X16);

            print_String(0, 30, (const uint8_t *)
                "This will reset all ", FONT_5X8);
            print_String(0, 40, (const uint8_t *)
                "system settings to ", FONT_5X8);
            print_String(0, 50, (const uint8_t *)
                "default", FONT_5X8);

            print_String(0, 70, (const uint8_t *)
                "Are you sure?", FONT_5X8);
            print_String(80, 70, (const uint8_t *)
                "No", FONT_5X8);
            print_String(80, 90, (const uint8_t *)
                "Yes", FONT_5X8);

            if (arrowState == 0 || arrowState == 2)
            {
                OLED_PrintArrow(95, 70);
            }
            else
            {
                OLED_PrintArrow(100, 90);
            }

            if (ENTER_PRESSED)
            {
                if (arrowState == 0 || arrowState == 2)
                {
                    nextPage = SETTINGS_HOME;
                }
                else
                {
                    Reset_System();     //Resets all global variables of system
                    nextPage = WELCOME_PAGE;
                }
                arrowState = 0;
                Clear_Screen();
```

```
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            else if (BACK_PRESSED)
            {
                nextPage = SETTINGS_HOME;
                arrowState = 0;
                Clear_Screen();
                oledSleepTimer = bcm2835_millis();      //reset sleep timer after each button press
            }
            break;
    }

        //prevArrowState = arrowState;
        //oledState = nextPage;

    return;
}

/**
   @Function checkButtons(void)
   @param None
   @return None
   @brief This function checksButtons and sets appropriate flag
   @note
   @author Brian Naranjo, 1/25/20
   @editor   */

void checkButtons(void)
{
    ENTER_PRESSED = FALSE;
    BACK_PRESSED = FALSE;
    DOWN_PRESSED = FALSE;
    UP_PRESSED = FALSE;

    enterButtonValue = DEV_Digital_Read(ENTER_Pin);
    downButtonValue = DEV_Digital_Read(DOWN_Pin);
    backButtonValue = DEV_Digital_Read(BACK_Pin);
    upButtonValue = DEV_Digital_Read(UP_Pin);

    DEV_Delay_ms(5);

    enterButtonValue2 = DEV_Digital_Read(ENTER_Pin);
    downButtonValue2 = DEV_Digital_Read(DOWN_Pin);
    backButtonValue2 = DEV_Digital_Read(BACK_Pin);
    upButtonValue2 = DEV_Digital_Read(UP_Pin);

    if (enterButtonValue == enterButtonValue2)
    {
        //Change in State
        if (enterButtonValue != lastEnterButtonState)
        {
                //Flipped, Low is pressed
            if (enterButtonValue == LOW)
            {
                ENTER_PRESSED = TRUE;   //set flag TRUE
                printf("Enter Pressed \r \n ");
            }
            else
            {
```

```
                ENTER_PRESSED = FALSE;  //set flag FALSE
            }
            lastEnterButtonState = enterButtonValue;
        }
    }
    if (downButtonValue == downButtonValue2)
    {
        //Change in State
        if (downButtonValue != lastDownButtonState)
        {
                //Flipped, Low is pressed
            if (downButtonValue == LOW)
            {
                DOWN_PRESSED = TRUE;    //set flag TRUE
                printf("Down Pressed \r \n ");
            }
            else
            {
                DOWN_PRESSED = FALSE;   //set flag FALSE
            }
            lastDownButtonState = downButtonValue;
        }
    }
    if (upButtonValue == upButtonValue2)
    {
        //Change in State
        if (upButtonValue != lastUpButtonState)
        {
                //Flipped, Low is pressed
            if (upButtonValue == LOW)
            {
                UP_PRESSED = TRUE;      //set flag TRUE
                printf("Up Pressed \r \n ");
            }
            else
            {
                UP_PRESSED = FALSE;     //set flag FALSE
            }
            lastUpButtonState = upButtonValue;
        }
    }
    if (upButtonValue == upButtonValue2)
    {
        //Change in State
        if (backButtonValue != lastBackButtonState)
        {
                //Flipped, Low is pressed
            if (backButtonValue == LOW)
            {
                BACK_PRESSED = TRUE;    //set flag TRUE
                printf("Back Pressed \r \n ");
            }
            else
            {
                BACK_PRESSED = FALSE;   //set flag FALSE
            }
            lastBackButtonState = backButtonValue;
        }
    }
```

```
}

/*@name: convertFloat_String
   @param: in - float to be converted to String
   @param: buffer - char variable used to store float string output
   @return: 0
*/

int convertFloat_String(float in, char buffer[100])
{
    temp_wholeVal = in ;        //Store float into temporary float variable

    if (in < 0)
    {
        temp_wholeVal = - in ;  //Store positive value if negative
    }   //Minus sign taken care of in print statement

    wholeVal = temp_wholeVal;   //Store Whole Integer Value

    temp_decimalVal = temp_wholeVal - wholeVal; //Obtain remainder as float

    decimalVal = trunc(temp_decimalVal *10000); //Store decimal value as whole int

        //Store Int Values into string to format as a float value
    if (in < 0)
    {
        sprintf(buffer, "-%d.%01d", wholeVal, decimalVal);
    }
    else
    {
        sprintf(buffer, "%d.%01d", wholeVal, decimalVal);
    }

    return 0;
}

/*@name: printGrid
   @param: x0 - initial x position for grid
   @param: x1 - final x position for grid
   @param: y0 - initial y position for grid
   @param: y1 - final y position for grid
   @param: xtics - # of lines on x line
   @param: ytics - # of lines on y line
   @return: TRUE/FALSE depending if grid was successfully printed
*/

int printGrid(int16_t x0, int16_t x1, int16_t y0, int16_t y1, int16_t xtics, int16_t ytics)
{
    int i = 0;
    int xTic = 0;
    int yTic = 0;
    int incrementX = (x1 - x0) / xtics;
    int incrementY = (y1 - y0) / ytics;

        //printf("Xspaces: %d", incrementX);    //Testing incrementX/Y
        //printf("Yspaces: %d", incrementY);

        //print x-axis
    Write_Line(x0, y1, x1, y1);
```

```

    xTic = x0 + incrementX;

        //Print Tic Marks on X Axis
    for (i = 0; i <= xtics - 1; i++)
    {
        Write_Line(xTic, y1 - 2, xTic, y1 + 2);
        xTic += incrementX;
    }

        //print y-axis
    Write_Line(x0, y0, x0, y1);

    yTic = y1 - incrementY;

        //Print Tic Marks on Y Axis
    for (i = 0; i <= ytics - 1; i++)
    {
        Write_Line(x0 - 2, yTic, x0 + 2, yTic);
        yTic -= incrementY;
    }

    return 0;

}

/*@name: printHoseStatus
   @param: x - initial x position for first Hose Print line
   @param: y - initial y position for first Hose Print line
   @param: status - current hose status
   @return: void
*/

void printHoseStatus(int16_t x, int16_t y, uint8_t status)
{
    if (status & 0x01)
    {
        print_String(x, y, (const uint8_t *)
            "Hose 1: ON", FONT_5X8);
    }
    else
    {
        print_String(x, y, (const uint8_t *)
            "Hose 1: OFF", FONT_5X8);
    }

    if (status & 0x02)
    {
        print_String(x, y + 15, (const uint8_t *)
            "Hose 2: ON", FONT_5X8);
    }
    else
    {
        print_String(x, y + 15, (const uint8_t *)
            "Hose 2: OFF", FONT_5X8);
    }

    if (status & 0x04)
    {
```

```
        print_String(x, y + 30, (const uint8_t *)
            "Hose 3: ON", FONT_5X8);
    }
    else
    {
        print_String(x, y + 30, (const uint8_t *)
            "Hose 3: OFF", FONT_5X8);
    }
}

/*@name: printAxelsLabels
   @param: x0 - initial x position for grid
   @param: x1 - final x position for grid
   @param: y0 - initial y position for grid
   @param: y1 - final y position for grid
   @param: xtics - # of lines on x line
   @param: ytics - # of lines on y line
   @return: TRUE/FALSE depending if grid was successfully printed
*/

int printAxesLabels(int16_t x0, int16_t y0)
{
    int x_Axis = 80;
    int y_Axis = 0;
    int temp_x = x0 + 5;        //Initialize Index variables
    int temp_y = y0;
    int i = 0;

        //Print X Value Axes Values
    for (i = 0; i < 6; i++)
    {
        sprintf(intBuffer, "%d", y_Axis);
        print_String(temp_x, temp_y, (const uint8_t *) intBuffer, FONT_5X8);
        y_Axis += 20;
        temp_y -= 20;
    }

    temp_y = y0;        //Initialize Index variables
    temp_x += x0 + 35;

        //Print Y Value Axes Values
    for (i = 0; i < 5; i++)
    {
        sprintf(intBuffer, "%d", x_Axis);
        print_String(temp_x, temp_y, (const uint8_t *) intBuffer, FONT_5X8);
        x_Axis -= 20;
        temp_x += 20;
    }

    return 0;
}

/*@name: plotSampleData
   @param: TestData - array of structs used for plotting
   @param: dataType - type of sensor data to display
   @param: size - # of elements in array

   @return: TRUE/FALSE depending if data was successfully printed
*/
```

```
int plotSampleData(D_Struct TestData[], uint8_t dataType, int16_t size)
{

    int gridPlotted = FALSE;
    int i = 0;
    int16_t x_Increment = 0;
    int16_t mapped_y_Value = 0;
    int16_t x_Value = 0;
    int16_t mapped_x_Value = x_Value + 20;      //start X at Left Side of Grid

        //Print Corresponding to selecetd DataType
    if (dataType == MOISTURE)
    {

        printf("Plotting Moisture \r\n ");

        Set_Color(RED);
        print_String(10, 0, (const uint8_t *)
            "Moisture Level", FONT_5X8);
        print_String(10, 10, (const uint8_t *)
            "(Node 2)", FONT_5X8);

        Set_Color(BLUE);
        print_String(0, 60, (const uint8_t *)
            "Water%", FONT_5X8);
        print_String(55, 120, (const uint8_t *)
            "Mins Ago", FONT_5X8);

        //Set number
        x_Increment = 100 / MAX_ELEMENTS;       //100 indicates the number of pixels
        //vertically on the OLED module

        for (i = 0; i <= (size - 1); i++)
        {
                //Plot Data Points

            mapped_y_Value = (int16_t)(110 - TestData[i].soilMoisture); //110 is bottom of OLED

                //printf("Element: %d \r\n", i);        //Testing Struct Elements

                //printf("Moisture Value: %f \r\n", TestData[i].soilMoisture);

            Draw_Pixel(mapped_x_Value, mapped_y_Value);
            mapped_x_Value += x_Increment;
        }

        gridPlotted = TRUE;
    }
    else if (dataType == SUNLIGHT)
    {
        printf("Plotting Sunlight \r\n ");

        Set_Color(RED);
        print_String(10, 0, (const uint8_t *)
            "Light Level", FONT_5X8);
        print_String(10, 10, (const uint8_t *)
            "(Node 2)", FONT_5X8);

```

```
            Set_Color(YELLOW);
            print_String(0, 60, (const uint8_t *)
                "Light%", FONT_5X8);
            print_String(55, 120, (const uint8_t *)
                "Mins Ago", FONT_5X8);

            x_Increment = 100 / MAX_ELEMENTS;       //Determine number points to increment in x-range

            for (i = 0; i <= (size - 1); i++)
            {
                    //Plot Data Points

                mapped_y_Value = (int16_t)(110 - TestData[i].lightLevel);   //110 is bottom of OLED

                    //printf("Element: %d \r\n", i);        //Testing Struct Elements

                    //printf("Light Level Value: %f \r\n", TestData[i].lightLevel);

                Draw_Pixel(mapped_x_Value, mapped_y_Value);
                mapped_x_Value += x_Increment;
            }

            gridPlotted = TRUE;
        }
        else if (dataType == TEMP)
        {
            printf("Plotting Temperature \r\n ");

            Set_Color(RED);
            print_String(20, 0, (const uint8_t *)
                "Temperature", FONT_5X8);
            print_String(20, 10, (const uint8_t *)
                "Node ", FONT_5X8);

            Set_Color(RED);
            print_String(0, 60, (const uint8_t *)
                "Deg(C)", FONT_5X8);
            print_String(55, 120, (const uint8_t *)
                "Mins Ago", FONT_5X8);

            x_Increment = 100 / MAX_ELEMENTS;       //Determine number points to increment in x-range

            for (i = 0; i <= size; i++)
            {
                    //Plot data points

                mapped_y_Value = (int16_t) TestData[i].temp_C;      //110 is bottom of OLED

                    //printf("Element: %d \r\n", i);        //Testing Struct Elements

                    //printf("Temp Value: %d \r\n", TestData[i].temp_C);

                Draw_Pixel(mapped_x_Value, 110 - mapped_y_Value);
                mapped_x_Value += x_Increment;
            }

            gridPlotted = TRUE;
        }
        else
```

```
    {
        printf(" No Plot Selected \r\n ");      //Print error message if invalid dataType

        Set_Color(RED);
        print_String(20, 0, (const uint8_t *)
            "No Plot", FONT_5X8);
        print_String(20, 10, (const uint8_t *)
            "Selected", FONT_5X8);

        gridPlotted = FALSE;
    }

    return gridPlotted;

}

/*@name: plotSampleData
   @param: TestData - array of structs used for plotting
   @param: dataType - type of sensor data to display
   @param: size - # of elements in array

   @return: TRUE/FALSE depending if data was successfully printed
*/
void Reset_System(void)
{
    int i;
        //Reset Hose Variables
    Hose[0] = Hose0;
    Hose[1] = Hose1;
    Hose[2] = Hose2;
    Hose[0].waterLevel = 1;
    Hose[1].waterLevel = 1;
    Hose[2].waterLevel = 1;
    Hose[0].control = AUTOMATIC;
    Hose[1].control = OFF;
    Hose[2].control = OFF;
    Hose[0].status = WATER_ON;
    Hose[1].status = WATER_OFF;
    Hose[2].status = WATER_OFF;

        //Reset Flow Sensor Variables
    for (i = 0; i < 3; i++)
    {
        pulseCount_fs[i] = 0;
        moisture_s_thresh[i] = 45.0;
    }

    oledColor = WHITE;  //Set Oled Color to default
    sleepTime = MIN_3;  //Reset Sleep Timer
    selected_Node = 2;  //Reset Selected Node for Testing

    return;

}

void Set_Select(uint8_t hose_selected)
{
    if (hose_selected == HOSE0)
    {
```

```
        //Set Select to 0x00
        DEV_Digital_Write(SEL_0_Pin, LOW);
        DEV_Digital_Write(SEL_1_Pin, LOW);
    }
    else if (hose_selected == HOSE1)
    {
        //Set Select to 0x01
        DEV_Digital_Write(SEL_0_Pin, HIGH);
        DEV_Digital_Write(SEL_1_Pin, LOW);
    }
    else
    {
        //Set Select to 0x10
        DEV_Digital_Write(SEL_0_Pin, LOW);
        DEV_Digital_Write(SEL_1_Pin, HIGH);
    }
    return;
}
```

6.5 Appendix: Sensor Node Code

```
// ********** INCLUDES **********
#include <SPI.h>
#include <EEPROM.h>
#include <Wire.h>
#include "RF24.h"
#include "nRF24L01.h"
#include "RF24Network.h"
#include "RF24Mesh.h"
#include "Adafruit_BMP085.h"
#include <printf.h>
#include <avr/sleep.h>
#include <avr/power.h>

/**** Configure the Radio ****/
RF24 radio(7, 8);
RF24Network network(radio);
RF24Mesh mesh(radio, network);

/**** #Defines ****/
#define time_Thresh Timer(C_Thresh.time_thresh, 10)//D_Struct.timeStamp)

/**** GLOBALS ****/
#define nodeID 4 // Set this to a different number for each node in the mesh network
#define MOISTURE_PIN A1
#define LIGHT_PIN A2
#define BATTERY A3
#define LIQUID_SENSE 10000
#define INTERRUPT_MASK 0b01000000
#define VOLTAGE_DIVIDER 10

#define MINS_10 600000

// C_Struct stores relevant thresholds
typedef struct {
  float sM_thresh;
  float sM_thresh_00;
  float lL_thresh;
  uint16_t tC_thresh;
  uint16_t time_thresh;
} C_Struct;

// D_Struct stores the relevant sensor data
typedef struct {
  float soilMoisture;
  float lightLevel;
  uint16_t temp_C;
  uint8_t digitalOut;
  uint8_t node_ID;
  uint8_t battLevel;
} D_Struct;

// Timers
uint32_t sleepTimer = 0;
uint32_t messageTimer = 0;
uint32_t witchTimer = 60000;
uint32_t batteryTimer = 0;

```

```
// Timer Support
uint8_t timerFlag = 0;
uint8_t message_Flag = 0;

// Sensor Vars
Adafruit_BMP085 bmp;
uint8_t bmpFlag = 0;

// RF24 Vars
uint8_t sleepFlag = 0;

// Use these vars to store the header data
uint8_t M_Dat = 0;

// C and D type structs
C_Struct Thresholds;
D_Struct Data_Struct;

/**** Function Prototypes ****/
void D_Struct_Serial_print(D_Struct);
void C_Struct_Serial_print(C_Struct);
void initC_Struct(C_Struct*);
float pullMoistureSensor(void);
float getMoistureReading(void);
float pullLightSensor(void);
float getLightReading(void);
uint8_t pullBatteryLevel(void);
int Timer(uint32_t, uint32_t);
int run_DeepOcean(D_Struct, C_Struct);


void setup() {
  Serial.begin(115200);
  printf_begin();

  // Set the IO
  pinMode(MOISTURE_PIN, INPUT);
  pinMode(LIGHT_PIN, INPUT);
  pinMode(BATTERY, INPUT);

  // Begin the Barometric Pressure Sensor
  // Pin out: Vin->5V, SCL->A5, SDA->A4
  //BMP not working on sensor nodes
  if (bmp.begin()) {
    bmpFlag = 1;
  } else {
    Serial.println(F("BMP Failed to init"));
  }

  // Set this node as the master node
  mesh.setNodeID(nodeID);

  // Connect to the mesh
  Serial.println(F("Connecting to the mesh..."));
  mesh.begin();

  // Print out the mesh addr
  Serial.print(F("Mesh Network ID: "));
  Serial.println(mesh.getNodeID());
```

```
  Serial.print(F("Mesh Address: ")); Serial.println(mesh.getAddress(nodeID));
  radio.setPALevel(RF24_PA_MAX);

  //  radio.printDetails();
  Serial.println(F("**********************************\r\n"));

  // initialize the thresholds
  initC_Struct(&Thresholds);
  C_Struct_Serial_print(Thresholds);
  Serial.print(F("\n"));

  // Setting the watchdog timer
  set_sleep_mode(SLEEP_MODE_IDLE);
  network.setup_watchdog(9);
  sleepFlag = 1;
}

void loop() {

  // Keep the network updated
  mesh.update();



  /**** Network Data Loop ****/
  // Check for incoming data from other nodes
  if (network.available()) {

    // Create a header var to store incoming network header
    RF24NetworkHeader header;
    // Get the data from the current header
    network.peek(header);

    // Switch on the header type, we only want the data if addressed to the master
    switch (header.type) {

      // 'S' Type messages ask the sensor to read and send sensor data after evals
      case 'S':
        network.read(header, &M_Dat, sizeof(M_Dat));
        Serial.print(F("\r\n"));
        Serial.print(F("Received 'S' Type Message: ")); Serial.println(M_Dat);
        break;

      // 'C' Type messages tell the sensor to calibrate or change its thresholds
      case 'C':
        network.read(header, &M_Dat, sizeof(M_Dat));
        Serial.print(F("\r\n"));
        Serial.print(F("Received 'C' Type Message: ")); Serial.println(M_Dat);
    }
  }


  /**** Battery Level Check ****/
  if (Timer(MINS_10, batteryTimer) && bmpFlag) {
    batteryTimer = millis();
    uint8_t batteryVoltage = pullBatteryLevel();
    if (batteryVoltage <= 35) {
      batteryVoltage = pullBatteryLevel();
      if (batteryVoltage <= 35) {
```

```
      printf("Battery Level Low: %d\n\n------- Reset Device To Continue -------",\
        batteryVoltage);
      set_sleep_mode(SLEEP_MODE_PWR_DOWN);
      while (1) {
        sleep_enable();
        sleep_cpu();
      }
    }
  }
}




/**** Read Sensors ****/

if (sleepFlag) {
  sleepFlag = 0; // Ensures that we only read and send a message once after waking up

  // Read all sensors
  Data_Struct.soilMoisture = pullMoistureSensor();
  Data_Struct.lightLevel = pullLightSensor();
  if (bmpFlag) {
    Data_Struct.temp_C = bmp.readTemperature();
  }
  Data_Struct.battLevel = pullBatteryLevel();
  Data_Struct.digitalOut = run_DeepOcean(Data_Struct, Thresholds);
  Data_Struct.node_ID = nodeID;


  /**** Data Transmission ****/

  if (mesh.checkConnection()) {
    // Send the D_Struct to Master
    // Sends the data up through the mesh to the master node to be evaluated
    if (!mesh.write(&Data_Struct, 'D', sizeof(Data_Struct), 0)) {
      Serial.println(F("Send failed; checking network connection."));
      // Check if still connected
      if (!mesh.checkConnection()) {
        // Reconnect to the network if disconnected and no send
        Serial.println(F("Re-initializing the Network Address..."));
        mesh.renewAddress();
        Serial.print(F("New Network Addr: ")); Serial.println(mesh.getAddress(nodeID));
      } else {
        Serial.println(F("Network connection good."));
        Serial.println(F("**********************************\r\n"));
        sleepFlag = 1;
      }
    } else {
      Serial.println(F("**********************************"));
      Serial.println(F("Sending Data to Master")); D_Struct_Serial_print(Data_Struct);
      // Set the flag to check for a failed message response
      message_Flag = 1; messageTimer = millis();
    }
  } else {
    // Reconnect to the mesh if disconnected
    Serial.println(F("Re-initializing the Network Address..."));
    mesh.renewAddress();
    Serial.print(F("New Network Addr: ")); Serial.println(mesh.getAddress(nodeID));
```

```
    }
  }


  /**** No Message Response ****/

  // Reset the mesh connection
  if (message_Flag && Timer(1000, messageTimer)) {
    message_Flag = 0;
    // Reconnect to the network
    if (!mesh.checkConnection()) {
      Serial.println(F("Re-initializing the network ID..."));
      mesh.renewAddress();
      Serial.print(F("New Network Address: ")); Serial.println(mesh.getAddress(nodeID));
    }
    network.sleepNode(8, 255); // Node goes to sleep here
    sleepFlag = 1; // Tell the node it's time to read sensors and send a message
  }

  /**** 'C' Type Data Evaluation ****/

  // Based on the 'C' type data, re-configure the thresholds


  /**** 'D' Type Data Evaluation ****/

  // Responding to the S or C type message from the master
  if (M_Dat && message_Flag) {
    // Turn off the message response flag
    message_Flag = 0;
    // If M_Dat == 2, reconfig the thresholds
    // Reset the data variables
    M_Dat = 0;
    // Go to sleep
    Serial.println(F("Received Sleep Instructions From Master"));
    network.sleepNode(8, 255); // Node goes to sleep here
    sleepFlag = 1; // Tell the node it's time to read sensors and send a message
  }

  /**** Config Options ****/

} // Loop


/**** Helper Functions ****/

void C_Struct_Serial_print(C_Struct sct) {
  Serial.print(F("Soil Moisture Threshold: ")); Serial.println(sct.sM_thresh);
  Serial.print(F("Soil Moisture Danger Threshold: ")); Serial.println(sct.sM_thresh_00);
  //Serial.print(F("Barometric Pressure Threshold: "); Serial.println(F(sct.bP_thresh);
  Serial.print(F("Ambient Light Level Threshold: ")); Serial.println(sct.lL_thresh);
  Serial.print(F("Ambient Temperature Threshold: ")); Serial.println(sct.tC_thresh);
  Serial.print(F("Maximum TimeStamp Threshold: ")); Serial.println(sct.time_thresh);
  return;
}

void D_Struct_Serial_print(D_Struct sct) {
  Serial.print(F("Soil Moisture Cont. (g%): ")); Serial.println(sct.soilMoisture);
  Serial.print(F("Ambient Lux Level   (lx): ")); Serial.println(sct.lightLevel);
```

```
  Serial.print(F("Ambient Temperature (C ): ")); Serial.println(sct.temp_C);
  Serial.print(F("Calucated Digital Output: ")); Serial.println(sct.digitalOut);
  Serial.print(F("Power Supply Battery(dV): ")); Serial.println(sct.battLevel);
  Serial.print(F("Node ID: ")); Serial.println(sct.node_ID);
  return;
}

void initC_Struct(C_Struct* sct) {
  sct->sM_thresh = 65;
  sct->sM_thresh_00 = 45;
  sct->lL_thresh = 30;
  sct->tC_thresh = 5;
  sct->time_thresh = 30000;
  return;
}


/* @name: getMoistureReading
   @param: none
   @return: value of the mapped sensor value
*/
float getMoistureReading(void) {
  // First map the voltage reading into a resistance
  float soilV = map(analogRead(MOISTURE_PIN), 0, 1023, 0, 500);
  // convert to soil resistance in kohms
  float R_probes = (500 / soilV);
  R_probes -= 1;
  R_probes *= 10;
  // convert to percentage of gravimetric water content (gwc)
  R_probes = pow((R_probes / 2.81), -1 / 2.774) * 100;
  // Returns the mapped analog value
  // A voltage of 2.5V should return a gwc of 60-70%
  return R_probes;
}


/* @name: pullMoistureSensor
   @param: none
   @return: value of the mapped sensor value
*/
float pullMoistureSensor(void) {
  float read1 = getMoistureReading();
  delayMicroseconds(10);
  float read2 = getMoistureReading();
  delayMicroseconds(10);
  float read3 = getMoistureReading();
  delayMicroseconds(10);
  float read4 = getMoistureReading();
  delayMicroseconds(10);
  float read5 = getMoistureReading();
  return ((read1 + read2 + read3 + read4 + read5) / 5);
}


/* @name: getLightReading
   @param: none
   @return: value of the mapped sensor value
*/
float getLightReading(void) {
```

```
  float b = -0.94;
  float c = 38.9;
  float a = 0.014;
  // First map the voltage reading
  float lightV = map(analogRead(LIGHT_PIN), 0, 1023, 0, 500);
  float mr_Lumen = lightV - b * c * 100;
  mr_Lumen /= c * 100;
  mr_Lumen = pow(mr_Lumen, 1 / a);
  // Returns the mapped analog value
  return (mr_Lumen);
}


/* @name: pullLightSensor
   @param: none
   @return: averaged value of the mapped sensor value
*/
float pullLightSensor(void) {
  float read1 = getLightReading();
  delayMicroseconds(10);
  float read2 = getLightReading();
  delayMicroseconds(10);
  float read3 = getLightReading();
  delayMicroseconds(10);
  float read4 = getLightReading();
  delayMicroseconds(10);
  float read5 = getLightReading();
  return ((read1 + read2 + read3 + read4 + read5) / 5);
}

/* @name: getBatteryReading
   @param: none
   @return: mapped battery voltage in dV
*/
uint8_t getBatteryReading(void) {
  float rawVoltageDivider1 = ((float)analogRead(BATTERY) * 50.5) / 1023.0;
  delayMicroseconds(10);
  float rawVoltageDivider2 = ((float)analogRead(BATTERY) * 50.5) / 1023.0;
  delayMicroseconds(10);
  float rawVoltageDivider3 = ((float)analogRead(BATTERY) * 50.5) / 1023.0;
  delayMicroseconds(10);
  float rawVoltageDivider4 = ((float)analogRead(BATTERY) * 50.5) / 1023.0;
  delayMicroseconds(10);
  float rawVoltageDivider5 = ((float)analogRead(BATTERY) * 50.5) / 1023.0;
  delayMicroseconds(10);
  float rawVoltageDivider6 = ((float)analogRead(BATTERY) * 50.5) / 1023.0;
  delayMicroseconds(10);
  float rawVoltageDivider7 = ((float)analogRead(BATTERY) * 50.5) / 1023.0;
  delayMicroseconds(10);
  float rawVoltageDivider8 = ((float)analogRead(BATTERY) * 50.5) / 1023.0;
  delayMicroseconds(10);
  float rawVoltageDivider9 = ((float)analogRead(BATTERY) * 50.5) / 1023.0;
  delayMicroseconds(10);
  float rawVoltageDivider10 = ((float)analogRead(BATTERY) * 50.5) / 1023.0;
  delayMicroseconds(10);
  float rawVoltageDivider11 = ((float)analogRead(BATTERY) * 50.5) / 1023.0;
  float battAvg = rawVoltageDivider2 + rawVoltageDivider3 + rawVoltageDivider4 +\
    rawVoltageDivider5 + rawVoltageDivider6;
  battAvg =  battAvg + rawVoltageDivider7 + rawVoltageDivider8 + rawVoltageDivider9 +\
```

```
    rawVoltageDivider10 + rawVoltageDivider11;
  uint8_t bat_soup = (uint8_t)battAvg;
  return bat_soup;
}


/* @name: getBatteryReading
   @param: none
   @return: mapped battery voltage in dV
*/
uint8_t pullBatteryLevel(void) {
  uint8_t mr_avg = getBatteryReading() + getBatteryReading() + getBatteryReading() +\
    getBatteryReading() + getBatteryReading();
  return (mr_avg / 5);
}


/* @name: Timer
   @param: delayThresh - timer duration
   @param: prevDelay - time in millis() when the timer started
   @return: digital high/low depending if timer elapsed or not
   This is a non-blocking timer that handles uint32_t overflow,
   it works off the internal function millis() as reference
*/
int Timer(uint32_t delayThresh, uint32_t prevDelay) {
  // Checks if the current time is at or beyond the set timer
  if ((millis() - prevDelay) >= delayThresh) {
    return 1;
  } else if (millis() < prevDelay) {
    //Checks and responds to overflow of the millis() timer
    if (((4294967296 - prevDelay) + millis()) >= delayThresh) {
      return 1;
    }
  }
  return 0;
}


/* @name: run_DeepOcean
   @param: D_Struct - struct that holds sensor data
   @param: C_Struct - struct that holds thresholds
   @return: digital high/low telling the system to
                      turn on or off the water
*/
int run_DeepOcean(D_Struct D_Struct, C_Struct C_Thresh) {
  int HydroHomie = 0;
  // Check for the time threshold

  // Chcek the soil moisture against the first threshold
  // If its light, then don't water unless it has been a long time
  if ((D_Struct.soilMoisture < C_Thresh.sM_thresh) && \
    (D_Struct.lightLevel <= C_Thresh.lL_thresh)) { //
    HydroHomie = 1;
  }

  // Check temperature to prevent freezing
  // Also make sure you only water once in a while so water is not
  // always on when its cold
  else if ((D_Struct.temp_C <= C_Thresh.tC_thresh) && bmpFlag) {
```

```
    HydroHomie = 1;
  }

  // Water immediately if soilMoisture goes below a certain level
  if (D_Struct.soilMoisture < C_Thresh.sM_thresh_00) {
    return 2;
  }

  // In main, make sure you update the timestamp if the output is >0
  return HydroHomie;
}
```

6.6 Appendix: Dark Sky API Code

```
import sys

sys.path.append('/home/pi/.local/lib/python2.7/site-packages')

import forecastio

# Place the unique Dark Sky API key here
api_key = "2ef3d37cae4747a0cdc3c75cb4c5b3ad"

# Location: Santa Cruz (lat = 36.9741, lng = -122.0308)
lat = 36.9741
lng = -122.0308

# Egypt
#lat = 26.8206
#lng = 30.8025

# this returns the load forecast object with the given parametersL key, latitiude, and longitude
forecast = forecastio.load_forecast(api_key, lat, lng)

# provides a vague hourly forecast
byHour = forecast.hourly()
daily = forecast.daily()
currentForecast = forecast.currently()
#print ("Daily Forecast: %s" % daily.summary)
#print ("Hourly Forecast: %s" % byHour.summary)
#print ("Current Forecast: %s" % currentForecast.summary)
#print (byHour.icon)

# this returns a map object, which contains all the weather data
result = forecast.json
curr = result.get('currently')
dail = result.get('daily')

# acquires current time, outputs in seconds
seconds = curr.get('time')

# conversion from seconds to clock time
seconds = seconds % (24 * 3600)
hour = seconds // 3600
hour = hour
seconds %= 3600
minutes = seconds // 60
seconds %= 60
```

```

# convert humidity to percent
#humidity = curr.get('humidity') * 100

# prints out the desired forecast information to the shell, where it is read
# by the main central hub program
print (dail.get('data')[0].get('precipProbability')) # daily rain probability
print (curr.get('temperature')) # current temperature
print (curr.get('humidity')) # current humidity
print (curr.get('pressure')) # current pressure
print (curr.get('windSpeed')) # current wind speed
print (curr.get('windBearing')) # current wind direction
```