C.H.A.N.O.S.

Controlled Hydration Automated Nutrient Optimization System

Senior Design 1 Final Report

Department of Electrical and Computer Engineering
College of Engineering and Computer Science
University of Central Florida

Reviewers:
Dr. George Atia (ECE)
Dr. Jonguok Choi (CS)
Mark Maddox (ECE)

Mentor: Dr. Chung Yong Chan

Group 10
Authors:

Elliott D'Amato	Brian Ericson	Brandon James	Adrian Vergara
Computer Engineering	*Computer Engineering*	*Electrical Engineering*	*Electrical Engineering*

Table of Content

Table of Content ..... **I**
List of Tables ..... **VIII**
List of Figures ..... **X**
Acknowledgment ..... **XI**
Chapter 1 - Executive Summary **1**
Chapter 2 - Project Description **2**
2.1 Motivation and Background 2
2.2 Existing Products 2
2.3 Goals, Objectives, and Features 4
2.4 Required Specifications 6
2.4.1 House of Quality Table 7
2.5 Block Diagrams 7
2.5.1 Hardware 8
2.5.2 Software 9
2.6 Prototype illustration/blueprint 10
Chapter 3 - Research and Investigation **11**
3.1 Processing and Control 11
3.1.1 MCU, MPU, & FPGA 11
3.1.2 Microcontroller Products 12
3.1.2.1 ESP32 (Espressif) 12
3.1.2.2 STM32 (STMicroelectronics) 14
3.1.2.3 nRF (Nordic Semiconductor) 14
3.1.2.4 CC32xx (Texas Instruments) 15
3.1.2.5 PIC32/WFI32 & ATSAMW (Microchip Technology Inc.) 15
3.1.2.6 Selection of Microcontroller 16
3.1.3 Microcontroller Clock Source 16
3.2 Sensors and peripherals 18
3.2.1 Pumps 18
3.2.1.1 Centrifugal Pumps 19
3.2.1.2 Positive Displacement Pumps 19
3.2.1.3 Submersible Pumps 20
3.2.1.4 Peristaltic Pump 20
3.2.2 Pump Selection 21
3.2.2.1 Submersible Pumps 21
3.2.2.2 Peristaltic Pumps 22
3.2.3 Light Sensor 24
3.2.3.1 Light Dependent Resistors 24

3.2.3.2 Photodiodes....24
3.2.3.3 Phototransistors....25
3.2.3.4 Light Sensor Selection....25
3.2.3.5 Conclusion To Light Sensor Selection....26
3.2.4 Water Level Sensor....27
3.2.4.1 Float/Switch Sensor....28
3.2.4.2 Ultrasonic Sensor....29
3.2.4.3 Pressure Sensors....29
3.2.4.4 Capacitive Sensors....30
3.2.5 Water Level Sensor Selection....30
3.2.5.1 Water Level Sensor Selection Conclusion....31
3.2.6 Soil Moisture Sensor Technology....31
3.2.7 Soil Moisture Sensor Product....33
3.3 PSU....34
3.3.1 Batteries....34
3.3.1.1 Lithium Ion....34
3.3.1.2 Lithium Polymer....35
3.3.1.3 Nickel Metal Hydride....35
3.3.1.4 Lead Acid....36
3.3.1.5 Battery Technology Conclusion....36
3.3.1.6 Battery Selection....36
3.3.2 Solar Panels....37
3.3.2.1 Monocrystalline Silicon....38
3.3.2.2 Polycrystalline Silicon....39
3.3.2.3 Passivated Emitter and Rear Cell....39
3.3.2.4 Thin-Film....40
3.3.2.5 Solar Technology Conclusion....41
3.3.2.6 Solar Panel Selection....41
3.3.3 Voltage Regulators....42
3.3.3.1 Linear Regulators....42
3.3.3.2 Switching Regulators....42
3.3.3.3 Regulator Technology Conclusion....43
3.3.3.4 Buck Converter Selection....43
3.3.3.5 Boost Converter Selection....44
3.3.4 Charge Controllers....45
3.3.4.1 PWM Pulse Width Modulation Controllers....46
3.3.4.2 MPPT Maximum Power Point Tracking Controllers....47
3.3.4.3 Other Controller Types....47

3.3.4.4 Conclusion for Charge controllers..... 48
3.3.4.5 Charge controller Selection..... 48
3.4 Communication..... 49
3.4.1 Internet Communication..... 49
3.4.2 Serial Communication..... 50
3.4.2.1 USB-UART Conversion..... 50
3.5 Software..... 51
3.5.1 Web Development Stacks..... 51
3.5.1.1 LAMP..... 51
3.5.1.2 MERN..... 53
3.5.1.3 PERN..... 54
3.5.2 Containerization..... 54
3.5.2.1 Docker..... 54
3.5.2.2 Podman..... 54
3.5.3 Embedded (ESP32) Programming Language..... 54
3.5.4 ESP32 Integrated Development Environment (IDE)..... 55
Chapter 4 - Standards and Design Constraints..... 55
4.1 Standards..... 55
4.1.1 IEEE 802.11..... 56
4.1.2 HTTP..... 56
4.1.3 PCB..... 59
4.1.4 ESP Hardware Guidelines..... 60
4.1.5 TCP..... 60
4.1.6 Wiring Standards..... 62
4.2 Constraints..... 62
4.2.1 Economic Constraints..... 62
4.2.2 Environmental Constraints..... 63
4.2.3 Sustainability Constraints..... 64
4.2.4 Health and Safety Constraints..... 64
4.2.5 Security Constraints..... 65
4.2.6 Time Constraint..... 66
Chapter 5 - Applications of ChatGPT and other Similar Platforms..... 67
Chapter 6 - Hardware Design..... 72
6.1 Hardware..... 72
6.1.1 Materials and Overall Design..... 72
6.1.2 MCU..... 73
6.1.3 Pump..... 74
6.1.4 Sensors..... 75

6.1.4.1 Light Sensor..... 75
6.1.4.2 Soil Moisture Sensor..... 77
6.1.4.3 Water level Sensor..... 78
6.1.5 Electrical Power System..... 79
6.1.5.1 Charging and Power Storage..... 80
6.1.5.2 Voltage Regulation..... 82
6.1.6 Final Overall psu design..... 85
6.1.7 USB-UART Conversion Circuit..... 89
Chapter 7 - Software Design..... 91
7.1 System Initialization Process..... 91
7.1.1 Initial Setup..... 91
7.1.2 Restoring State After a Power Loss Event..... 92
7.2 Microcontroller Software Architecture..... 93
7.3 Definition of important data classes and structs..... 100
7.3.1 Plants..... 100
7.3.2 Pots..... 102
7.3.3 Users..... 105
7.4 Reading data from sensors..... 107
7.5 Memory Management..... 109
7.6 Communications..... 110
7.6.1 Transferring Data..... 110
7.6.2 HTTP Methods..... 111
7.6.2.1 Microprocessor..... 111
7.6.2.2 Server..... 111
7.6.2.3 Client..... 112
7.7 Web App Architecture..... 112
7.7.1 Client..... 112
7.7.2 Server..... 115
Chapter 8 - System Fabrication / Prototype Construction..... 116
8.1 Housing Enclosure..... 116
8.2 PCB Fabrication..... 116
8.2.1 Light Sensor..... 116
8.2.2 Horizontal Float Sensor..... 117
8.2.3 Soil Moisture Sensor..... 119
8.2.4 MCU and UART-USB PCB..... 120
8.2.4 PSU..... 120
Chapter 9 - System Testing and Evaluation..... 121
9.1 Breadboard testing..... 122

9.1.1 Sensor testing.....123
9.1.1.1 Light Sensor Testing.....123
9.1.1.2 Soil Moisture Sensor Testing.....124
9.1.1.3 Horizontal Float Sensor Testing.....126
9.1.2 Pump Testing.....126
9.2 Software Testing.....127
9.2.1 Embedded Software Testing.....127
9.3 overall integration.....129
9.3.1 Web Server / Microcontroller Integration.....129
9.4 Plan for SD2.....132
Chapter 10 - Administrative Content.....134
10.1 Budget.....134
10.2 Dates and Milestones.....136
10.3 Distribution of Work.....137
Chapter 11- Conclusion.....138
Appendix A - references.....
Appendix B - copyright permission.....
Appendix C - Data Sheet.....
Appendix D - Software code.....
Primary Arduino File.....
Appendix E - ChatGPT prompts and outcomes.....
How to Use This Code:.....
How the Code Works:.....
Operational Flow.....
Block Diagram.....
Block Diagram.....
Explanation of Components and Their Interaction:.....
1. Sensors:.....
2. ESP32 (Microcontroller):.....
3. Solar Power System:.....
4. Water Pump / Solenoid Valve (Actuator):.....
5. Cloud Server / MQTT Broker:.....
6. Web Dashboard (User Interface):.....
Overall Workflow:.....
Peristaltic Pumps: The Ideal Choice.....
Submersible Pumps: A Viable, Cost-Effective Alternative.....
Diaphragm Pumps & Solenoid Valves: Less Suitable for this Design.....
Conclusion.....

Analysis of the Proposed Circuit.....
How it Addresses the Submersible Pump Limitation.....
Minor Considerations / Enhancements (Optional):.....
Conclusion.....

List of Tables

Table 2.4.1 Specifications for System 6
Table 2.4.2 Specification for Components 6
Table 2.4.1.1 House of Quality 7
Table 3.1.1.1 Microprocessor, Microcontroller, FPGA Comparison 12
Table 3.1.2.1 ESP32 Microcontroller Feature Comparison 13
Table 3.1.2.2 ESP32 Microcontroller Power Consumption Comparison 14
Table 3.1.2.6 Select Microcontroller Cost Comparison 16
Table 3.1.3.1 Clock Source Technology Comparison 17
Table 3.2.1.1 Pump Comparison 18
Table 3.2.2.3.1 Pump Brand Comparison 23
Table 3.2.3.5.1 Light Sensor Comparison 26
Table 3.2.4.1 Water Level Sensor Comparison 27
Table 3.2.5.1.1 Comparison Between Pump Brands 31
Table 3.2.6.1 Soil Moisture Technology Comparison 33
Table 3.2.7.1 Capacitive Soil Moisture Sensor Component Comparison 34
Table 3.3.1.6.1 Comparison of Selected 18650 Li-Ion Batteries 36
Table 3.3.2.1 Solar Panel Technology Comparison 37
Table 3.3.2.6.1 Solar Panel Component Comparison 41
Table 3.3.3.4.1 Buck Converter Comparison 44
Table 3.3.3.5.1 Boost Converter Comparison 45
Table 3.3.4.1 Charge Controller Technology Comparison 46
Table 3.3.4.5.1 Charge Controller Comparison 49
Table 3.4.1.1 Wireless vs. Wired Communication Comparison 50
Table 3.4.2.1.1 USB-UART Converter IC Comparison 51
Table 3.5.3.1 Embedded Programming Languages for ESP32 Comparison 55
Table 3.5.4.1 ESP32 Integrated Development Environment Comparison 55
Table 5.1. Language Learning Model (LLM) Comparison 68
Table 6.1.5.1 Electrical Component Comparison 80
Table 6.1.5.1.1 Charging and MPPT Component Values 81
Table 6.1.5.1.2 Charging and Storage Electrical Characteristics 82
Table 6.1.5.2.1 Pump Electrical Characteristics 83
Table 6.1.5.2.2 Microcontroller + Sensor Rail Characteristics 84
Table 6.1.5.2.3 Regulator Summary Comparison 84
Table 6.1.7.1 Auto-reset Signals and States 91
Table 7.3.1.1 Plant Class Fields 100
Table 7.3.2.1 Pot Class Fields 102
Table 7.3.3.1 User Class Fields 105
Table 7.4.1 SensorReader Methods 108
Table 9.1.1.1 Light Sensor Readings 123
Table 9.1.1.2.1 Soil Moisture Sensor Data Test Results 125
Table 9.1.1.2.2 Soil Moisture Sensor Electrical Measurements 125
Table 10.1.1 Budget Estimate 134
Table 10.1.2 Bill of Materials 134
Table 10.2.1 Senior Design 1 Project Documentation Milestones 136
Table 10.2.2 Senior Design 1 Project Design 136

Table 10.2.3 Senior Design 1I Project Documentation Milestones 136
Table 10.3.1 Distribution of roles 137

List of Figures

Figure 2.5.1.1. Hardware Block Diagram…… 8
Figure 2.5.2.2. Microcontroller Software Block Diagram…….. 9
Figure 2.5.2.1. Web Application Software Block Diagram…… 9
Figure 2.6, top left, front view (a), top right, side view (b), bottom left, top view (c), and bottom right, internal view (d), of C.H.A.N.O.S……
10
Figure 6.1.2.1 Microcontroller Schematic…… 73
Figure 6.1.3.1 Pump Driver Schematic…… 75
Figure 6.1.4.1.1 Light Sensor Schematic……...76
Figure 6.1.4.2.1. Soil Moisture Sensor Schematic……77
Figure 6.1.4.3.1 Water Level Sensor Schematic…… 79
Figure 6.1.5.2.1 Boost Converter……...83
Figure 6.1.5.2.2 Buck Converter……84
Figure 6.1.6.1 12V Boost Converter Schematic.……
86
Figure 6.1.6.2 3.3V Buck Converter Schematic.……
86
Figure 6.1.6.3 Charge Controller Schematic.…….
87
Figure 6.1.7.1 USB-UART Conversion Schematic……
89
Figure 7.1.1.1 Initialization Setup Flowchart…… 92
Figure 7.1.2.1 Power-Loss Restoration Flowchart…… 93
Figure 7.2.1 Overall Software State Diagram……95
Figure 7.2.2 Activity Diagram for ESP32-AP State……...96
Figure 7.2.3 Activity Diagram for ESP32-STA State……97
Figure 7.2.4 Activity Diagram for Sensor Init State……..98
Figure 7.2.5 Activity Diagram for Sensor Check State……99
Figure 7.3.1.2 Plant Data Structure Diagram……...102
Figure 7.3.2.1 Pot Data Structure Diagram……..105
Figure 7.3.3.2 User Data Structure Diagram……
107
Figure 7.4.1 SensorReader Class Diagram…….. 108
Figure 7.7.1.1 Use Case Diagram for Web Server Client…… 113
Figure 7.7.1.2 Plant View Mock-Up…… 115
Figure 7.7.2.1 Server API Message Flow…… 116
Figure 8.2.1.1 Light Sensor PCB…… 117
Figure 8.2.1.2 2D Light Sensor Overview……
117 Figure 8.2.2.1 Water Level Sensor
PCB…….. 118
Figure 8.2.2.2 2D Horizontal Water Sensor……118
Figure 8.2.3.1 Soil Moisture Sensor PCB Design……119
Figure 8.2.4.1 MCU and UART-USB PCB Design……120
Figure 8.2.5.1 MCU PSU……121
Figure 9.1.1 Breadboard Testing……..123

Figure 9.1.1.1 Light Sensor with ESP32……. 124
Figure 9.1.1.3.1 Horizontal Float Sensor Test……. 126
Figure 9.2.2.1 Web Page Test Results……..129
Figure 9.3.1.1 Web Page with received data……130
Figure 9.3.1.2 Successful update of microcontroller data through server……
131
Figure 9.3.1.3 Updated received data seen by the server…….132

Acknowledgments

We would like to express our sincere gratitude to our professors Dr. Chung Yong Chan, Dr. Lei Wei, and Dr. Aravinda Kar for their invaluable guidance, feedback, and encouragement throughout the development and writing of this paper. Their support was instrumental to the success of the C.H.A.N.O.S. project.

We also extend our appreciation to our reviewers Dr. George Atia, Dr. Jonguok Choi, and Mr. Mark Maddox for generously taking the time from their busy schedules to review our project and provide thoughtful insights and assistance to make this project a success.

I would also like to Thank my wife for the support throughout this project, balancing a full time job and finishing this degree has been quite a journey. -Brandon James

Chapter 1 - Executive Summary

C.H.A.N.O.S. — the Controlled Hydration Automated Nutrient Optimization System — is a smart, solar-powered irrigation and sensing platform built to support plant health without requiring constant human attention. Designed with off-grid sustainability and intelligent automation in mind, C.H.A.N.O.S. monitors soil moisture, light levels, and water availability, and autonomously waters the plant using a controlled peristaltic pump system. It connects to a web dashboard that allows users to remotely monitor status and update plant-specific parameters.

The system is powered by a 2S Li-ion battery pack charged via solar energy. From that stored power, the system generates three regulated rails: 12 V for the water pump, 5 V for internal logic and voltage conversion, and 3.3 V for the ESP32 microcontroller and sensors. This modular power approach ensures the pump only activates when needed, while the logic and sensors remain always-on at minimal draw. A TPS61088-based boost converter supplies the 12 V rail with ample current headroom. For the 5 V and 3.3 V rails, we selected the LM2596 buck converter and AMS1117 linear regulator, respectively balancing efficiency with ease of prototyping and PCB integration. The system runs on a custom-configured ESP32 that polls soil and water level sensors and controls the pump based on user-defined thresholds. Every 15–20 minutes, it samples data and checks for watering events. The device also supports wireless communication, sending updates to the web dashboard via HTTP and providing a basic interface for users to track soil moisture and manage target moisture ranges. Plant configurations can be selected from preset templates or entered manually.

Hardware is split into subsystems to simplify design and prototyping: power delivery, water system, and sensing are all housed on separate PCBs. The components were selected not just for electrical compatibility, but also for long-term durability, solar efficiency, and ease of sourcing. The enclosure and internal layout were 3D-modeled to support modularity and serviceability. A 3D-printed casing houses the pot, tubing, sensors, and electronics while protecting against splashes and minimizing environmental exposure.This system addresses a modern, widespread problem: people want to care for plants, but life often gets in the way. Whether due to work schedules, travel, or simple forgetfulness, plants are frequently under- or over-watered. Our goal with C.H.A.N.O.S. was to eliminate that burden without losing the joy of plant ownership. What makes our solution stand out is its balance of autonomy, power-conscious design, and future adaptability including compatibility with smart home platforms via ESPHome and Home Assistant.

C.H.A.N.O.S. is a working prototype of a real-world product that meets both technical and practical needs. We've selected components with that in mind parts that are proven during testing but also reliable and compact enough for long-term embedded use.

With a core design grounded in modular hardware, efficient power regulation, and responsive sensing, C.H.A.N.O.S. delivers on its promise: intelligent, low-maintenance plant care powered entirely by the sun.

Chapter 2 - Project Description

2.1 Motivation and Background

Since transitioning to agriculture thousands of years ago, gardening has been a central part of human society. Even today, when most people do not perform agricultural labor, gardening can provide joy, relieve stress, and build confidence. Particularly, the care of indoor and outdoor plants is ubiquitous and can even develop a deep human-plant relationship.

However, the busy schedule of modern life and long work hours thwart many people's abilities to garden and effectively tend to plants. Sadly, this loss is only one of many disruptions between people and nature, a trend which is correlated with reduced cognitive function and increased anxiety and depression.

Meanwhile, many technological advancements have occurred within electrical and computer engineering. Robotics has helped automate menial tasks which previously required human intervention. New sensors provide these robots with information imperceptible to humans. Microelectronics and internet-of-things have permitted these tools to be affordable and convenient to use within the home.

It should come as no surprise, then, that these technological advancements have been used to implement products that assist in gardening without direct human intervention. However, many of these products have a narrow scope or do not exploit the full potential of these technologies, which lead to oversight of the nuances of gardening and plant biology. Furthermore, many commercial applications of new technologies have recklessly increased the power consumption within a person's life and home.

Thus, there is a clear need for a product which maximizes the potential of modern electronics to intricately care for plants commensurate with the compassion and experience of dedicated horticulturists. Despite additional complexity, this product should attempt to mitigate the environmental disadvantages of common electronic commodities.

2.2 Existing Products

An automatic irrigation system is not a novel idea. These systems already exist for horticultural (household) and agricultural (industrial) purposes. The distinction between these systems include the processes used to control the system and the sensors used to collect data about the plant, soil, or environment.

An article written by Moss et al. from Oklahoma State University describes two types of controllers, evapotranspiration and soil moisture sensor based, and three types of sensors–soil moisture, rain, and wind [2]. Evapotranspiration, or climate-based, controllers use data about the soil-to-air or plant-to-air movement of water to regulate the irrigation of an area of soil. The costs of these controllers were reported to be in the

hundreds or thousands of dollars. Controllers based on soil moisture sensors use data collected about water content in the soil to regulate the irrigation of the area. The location of these controllers must be carefully chosen. The efficacy of these controllers were reported as possibly higher than ET controllers, and the cost was similar to non-professional ET controllers.

The latter half of the article described three types of sensors which could be integrated into existing irrigation systems to increase efficiency. Many existing irrigation systems are timer-based, so the integration of soil moisture sensors can help classify whether a plot of soil must be watered at the specified time or if the soil is already sufficiently irrigated. These sensors are significantly cheaper than the aforementioned soil moisture sensor controllers (~ $100-$200). Rain/freeze sensors detect whether it is raining or frozen in the environment. In either case, further irrigation of the soil risks overwatering the plant or crop or freezing and damaging the roots/soil. Wind sensors measure the wind speed in the environment; high winds can hinder the efficacy of irrigation by spreading the water horizontally across the soil rather than vertically into the soil. Rain, freeze, and wind sensors are similar in cost to a soil moisture sensor.

Commercial automatic irrigation systems for indoor gardening were researched via a search on Google Shopping. An initial search was performed using the keyword "automatic plant watering system". The cost of these systems widely varied from about $30 to more than $100. All of the systems observed were primarily controlled by a timer, irrigating the plant/soil at time intervals and for watering durations set by the user. Additionally, these systems were powered by either power adapter cord or a battery. Most of these systems did not collect any data about the soil contents or environment surrounding the plant. However, one lower-cost system did have a separate irrigation setting based on the humidity of the surrounding environment. The lack of sensors or data collection is likely due to a tradeoff between number of plants that can be watered and the precision of care. In order to control irrigation based on the contents of soil, a sensor is required to place in the soil, so in order to irrigate multiple plants based on soil content, a large number of sensors would be required, increasing the cost and complexity of the system's electronics.

In fair consideration of all options for data-based automatic irrigation controllers, further research was conducted regarding evapotranspiration controllers. One of the most common ET controllers is a lysimeter–a device which calculates the evapotranspiration of water in a plot of soil by measuring the amount of water collected then lost in a small sample of the plot. Weighing lysimeters specifically measure the weight of a sample to calculate the amount of water gained and lost. A specific study of low-cost weighing lysimeters was performed by Dong and Hansen at Michigan State University and published in *Smart Agricultural Technology* [3]. The paper found that four accurate weighing lysimeters could be custom designed and manufactured for an average cost of $327 per lysimeter. Although this cost is drastically lower than professional-grade lysimeters, it is still high relative to the cost of a soil moisture sensor.

After developing many of our goals and objectives, multiple YouTube videos were found which described similar automatic gardening or irrigation systems. The first YouTube video was 10 years old and received about 3.2 million views. It described a “Arduino Garden Controller”, which used soil moisture and temperature sensors and a photoresistor to collect data about and automatically irrigate the creator’s garden [4]. The second YouTube video was 3 years old and received 50 thousand views. It showcased Flaura, a custom 3D-printed pot which held a pot above a water supply [5]. Using water level and soil moisture sensors, Flaura automatically waters the soil and uses an ESP32 microcontroller to communicate with a mobile app where the user can change and track certain parameters of the system. Flaura also used a rechargeable battery, and the creator mentioned adding solar power supply.

2.3 Goals, Objectives, and Features

Basic Goals
The first basic goal of this project is to design and create a system which automatically waters and nourishes a plant. A second basic goal is the automatic drainage of the water runoff. A third basic goal is the system should be adaptable to a plant and pot of the user’s choosing. There must be a way for the user to define the parameters relevant to the plant being watered, such as the required minimum and maximum values for soil moisture, temperature, electrical conductivity, or other characteristics.

Advanced Goals
An advanced goal is to reduce the environmental footprint of the design. This entails both low energy consumption and use of sustainable energy sources. This advanced goal will be met in the final product design.

Stretch Goals
A stretch goal is the integration of the automatic watering system into the users’ smart home system as an internet-of-things (IoT) device.

Objectives
To achieve the first basic goal, the first objective is collection of information about the plant that indicates when it must be watered. The cheapest and easiest form of this objective is measurement of the soil moisture using sensor technology. If the soil moisture is below a certain threshold, a pump is activated which transports water from a supply container to the soil. Other characteristics of the soil, such as temperature, may also help indicate whether the plant must be watered.

The objective for the second basic goal is the transport of the runoff from the pot to a separate container. An objective for the third basic goal is the design of a web or mobile app where the user can either select preset information for some common plants or define specific requirements for each characteristic. This would require the microcontroller to have features such as bluetooth or WiFi capabilities in order to communicate with the mobile/web app.

Multiple objectives are required for the advanced goal. A low energy consumption can be achieved by periodically sampling measurements every hour using timer based interrupts. Additionally, selection of lower voltage components, such as certain microcontrollers, pumps, and motors, can also reduce energy consumption. The use of sustainable energy can be achieved by using a solar-powered battery, because the plant and system will be positioned in the sunlight.

An objective which helps achieve the basic and advanced goals is the reuse of water runoff as a source of irrigation/nourishment, which can be accomplished via a filter. The runoff from the soil/pot would be filtered and transferred back to the water supply tank.

An objective for the stretch goal is the communication between C.H.A.N.O.S. and other smart home devices via bluetooth or other wireless communication protocols. For example, the system may communicate with and close smart home curtains when the plant has received a sufficient amount of sunlight. The system may also communicate with a smart home thermostat in order to measure or control humidity or temperature.

Features & Functionality

Based on the basic goals and objectives, a custom pot system/design was determined to be most convenient for the user, instead of a system which would be attached to a pot chosen by the user. This would limit the interconnections that the user is required to make and increase stability and reliability of the product. This custom pot system is planned as a 3D-printed design.

A soil moisture sensor is one of the key components of the product. The data from this sensor will determine if the plant must be watered. In addition to a soil moisture sensor, a solar sensor will also be connected to the system. This sensor will measure how much sunlight the plant has received and determine when the plant has received sufficient sunlight or when it must be rotated to receive sunlight on its other sides. A horizontal float switch sensor will be placed in the water supply tank and determine if the tank is low on water and must be refilled by the owner.

These sensors will be connected to a microcontroller. Based on sensor data, the microcontroller will determine when a pump must be activated to water the plant. The water pump will be connected to the water supply tank. Additionally, the microcontroller will determine when the pot must be rotated and when curtains must be moved to shade the plant. These tasks will be performed by motors connected to the microcontroller.

The microcontroller also communicates with a web app via Wi-Fi. The microcontroller sends data regarding the soil moisture levels, the water level/volume in the supply tank, and the solar sensor. The user can view this information, but can also enter data on the web app for the particular plant that is in the pot. This data includes the minimum and maximum soil moisture values for the plant, which the user can determine by manually entering or selecting a preset from a database of common plants.

In addition to the web app, the stretch goal described above requires additional communication features. A feature which helps meet these objectives would include a framework that streamlines the integration of C.H.A.N.O.S. with an existing smart home infrastructure. ESPHome, developed by Open Home Foundation, is an example of such a framework, which focuses on ESP microcontroller as well as providing desktop support. ESPHome cooperates with Home Assistant, a free, open-source software, which would detect and manage existing smart home devices.

The final feature of the system is the power supply, consisting of a solar panel, solar charge controller, and battery. The charge controller sends power to a voltage regulator and data to the microcontroller. The regulator distributes this power to the electronic components of the system, such as the microcontroller and water pump.

2.4 Required Specifications

Table 2.4.1 Specifications for System.

Parameter	Specification
Sampling of sensor data frequency	15 - 20 min
Time to observe sensor data in web app after wakeup	< 5 min
Accuracy of water level	<.5L
Water Flow Rate	0.1 - 5 mL / s
Supply Water Level Accuracy	>= 90%
Cost of final design	< $75 USD
Battery life	21-27 hours
Time to activate water pump after low soil moisture detection	< 1 min
Temperature	21 C < temperature < 50 C

Table 2.4.2 Specification for Components.

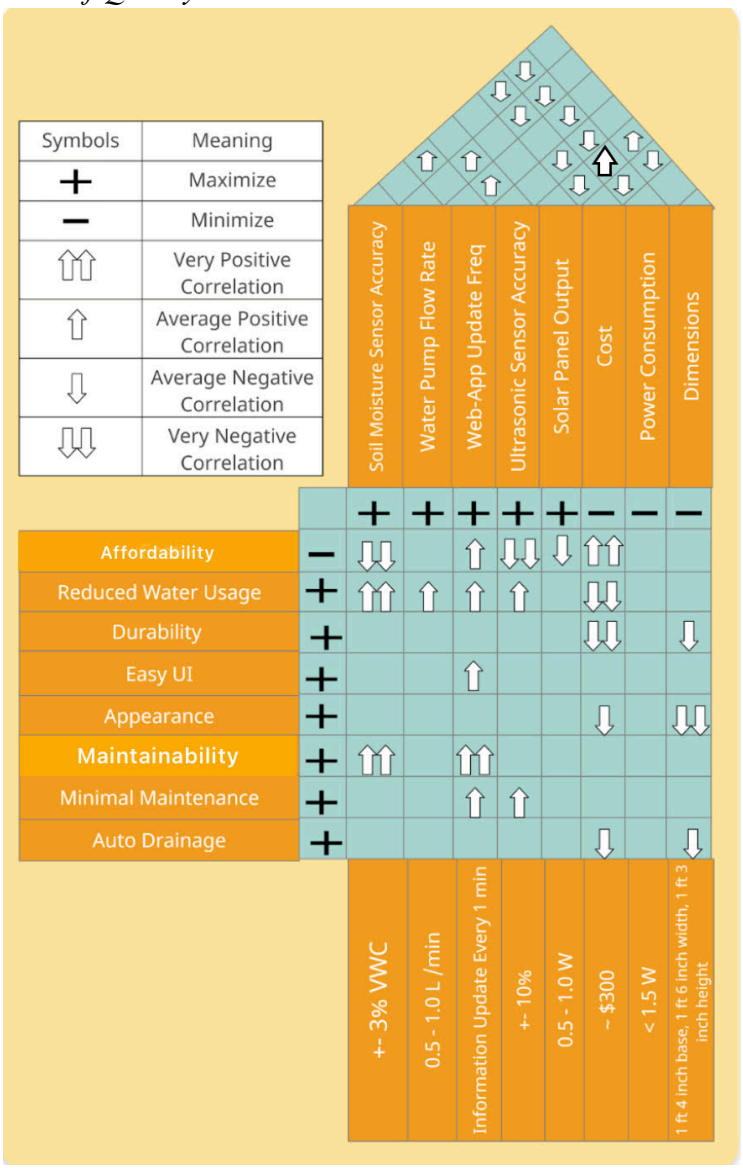
Component	Parameter	Specification
Soil Moisture Sensor	Measurement accuracy	± 3% volumetric water content (VWC)
Soil Moisture Sensor	Standard deviation	± 5%
Soil Moisture Sensor	Measurement time	< 1 s
Solar Panel	Max power output	> 3.0 W
Float Switch	Closing height	< 10 cm

Light Sensor	Resolution	± 10 lux
Light Sensor	Measurement Time	< 1 s

2.4.1 House of Quality Table

The house of quality table allows us to extrapolate between meeting our customers expectations with the design specifications needed to accomplish it. Comparing the tradeoffs between these allows us to view the potential impact of changing certain parts or aspects of the project. Below is a table with what we identified as the most important marketable aspects and design requirements for C.H.A.N.O.S.

Table 2.4.1.1 House of Quality.



2.5 Block Diagrams

2.5.1 Hardware

This block diagram represents the architecture of a solar powered plant life support system. The goal of this system is to maintain soil moisture using sensors and a water pump. As well as provide wireless updates to the user about the plant's life status.

We will be using an ESP32 microcontroller to monitor and control the various sensors and life sustaining systems. The sensors we plan to use are the following - light sensor, ultrasonic water level sensor and capacitive soil moisture sensor. These sensors will both provide information to the microcontroller (MCU) to make automated systems activate and provide data to the user.

Each of the blocks - life sustain, power and pcb will all be separate boards. The soil moisture sensor and solar sensor will be custom designed boards along with the water level sensor. The power block will also be its own board all feeding into the main pcb board label PCB on the diagram.

The hardware design aspects will primarily be done by the two EE - Brandon and Adrian. While the software components will be handled by the two CPE Elliott and Brian.

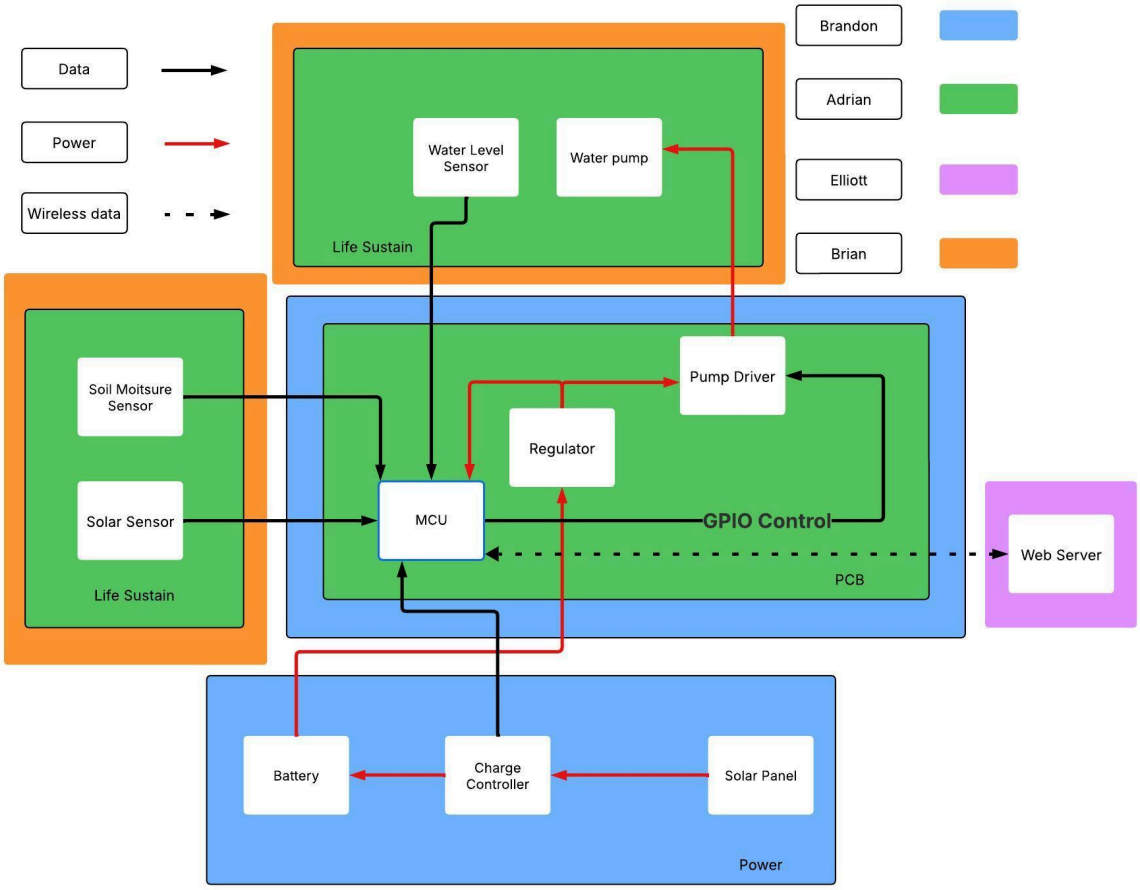


Figure 2.5.1.1. Hardware Block Diagram.

2.5.2 Software

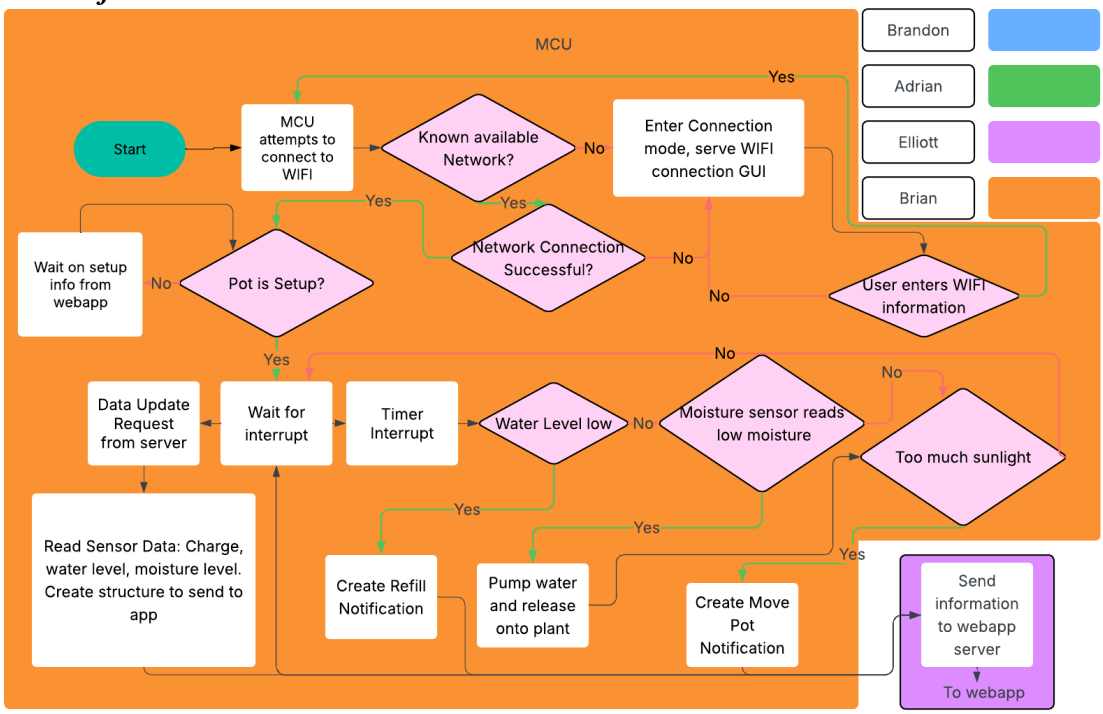


Figure 2.5.2.2. Microcontroller Software Block Diagram.

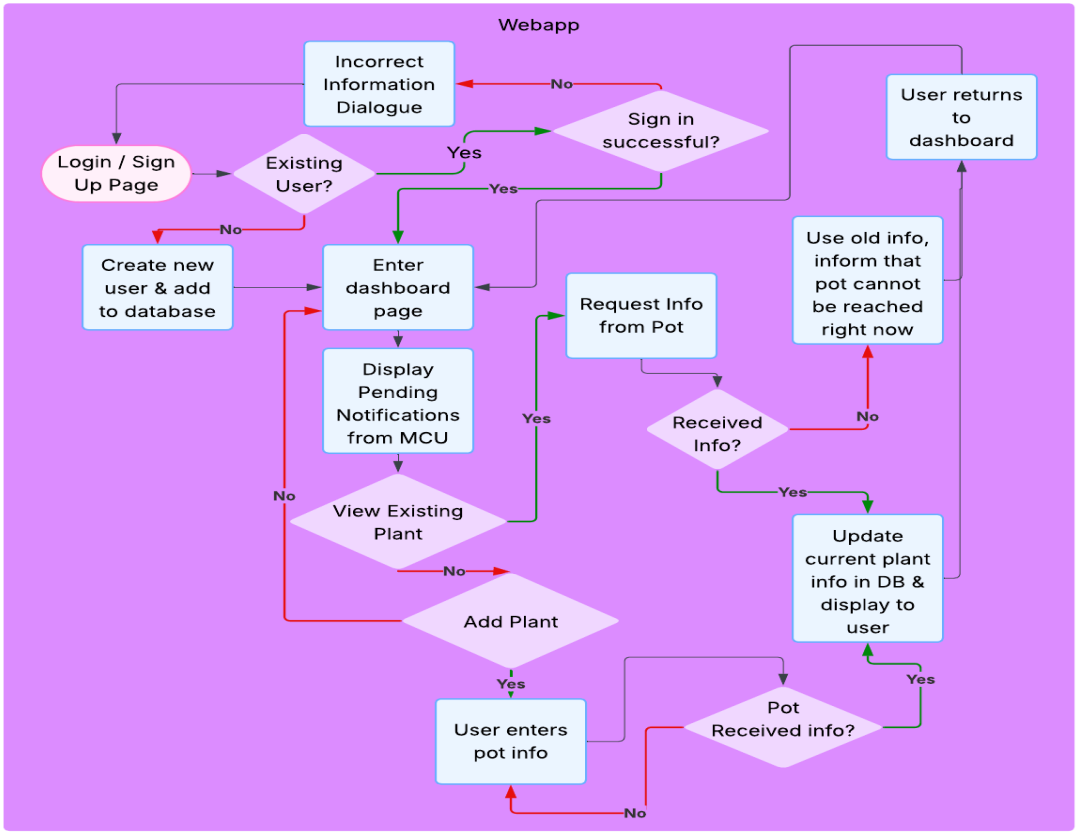


Figure 2.5.2.1. Web Application Software Block Diagram.

2.6 Prototype illustration/blueprint

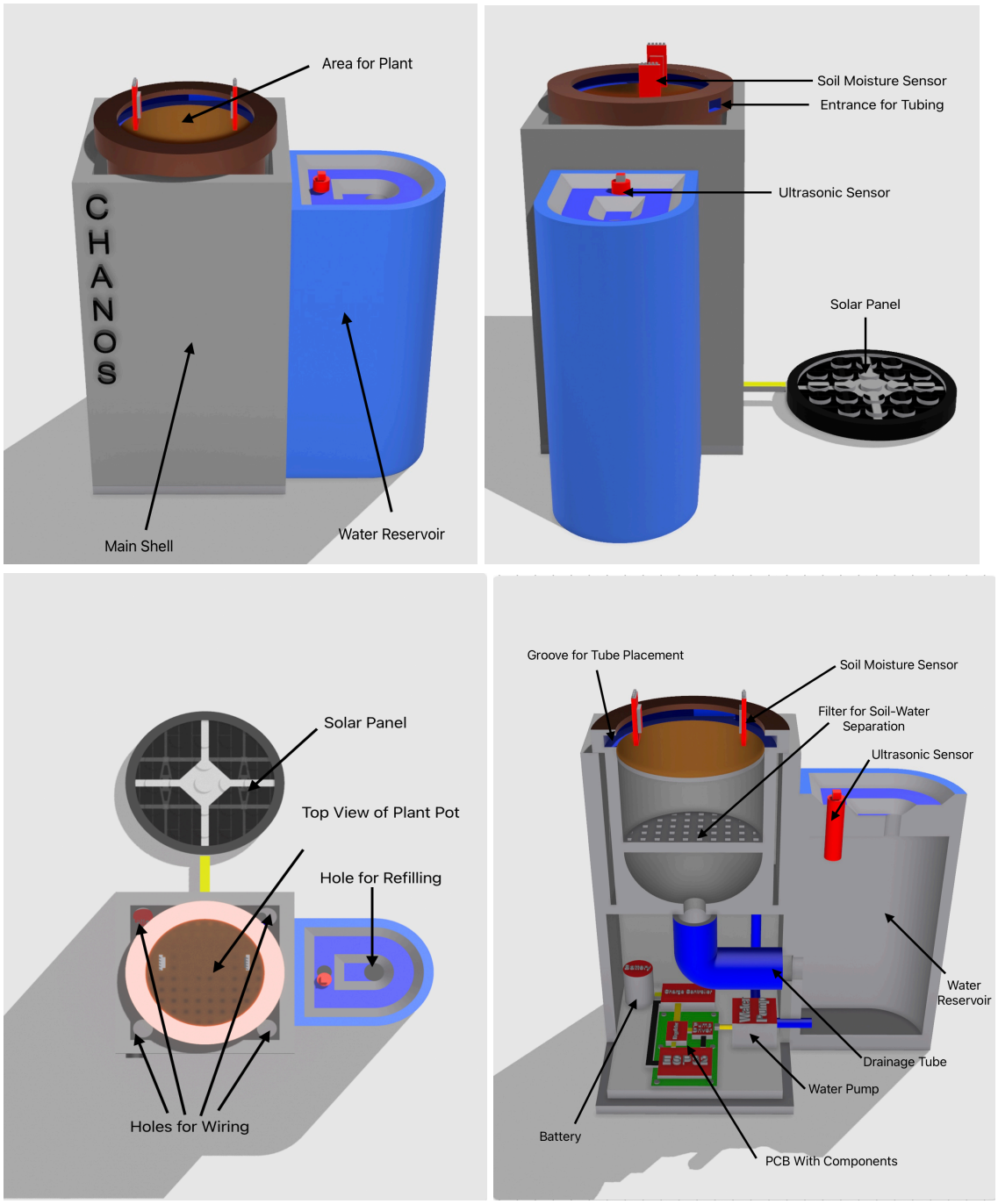


Figure 2.6, top left, front view (a), top right, side view (b), bottom left, top view (c), and bottom right, internal view (d), of C.H.A.N.O.S.

The above visuals show several views of the C.H.A.N.O.S. prototype. Breaking the color scheme down, we can loosely identify electrical components as anything highlighted in red, blue for anything related to water, green for the PCB, brown for the pot the plant

would be put in, and grey as the overall shell. We will start with explaining the area around where the pot would be. In this area we have our soil moisture sensor (red) and a groove (blue) that extends around the inner perimeter of the pot. This groove is meant for the end section of the tubing where water would be shooting out in a 360 degree fashion, making sure it reaches all parts of the soil in a semi-even way. In side view (b) you will see a small rectangular cut out on the outer perimeter of the pot where the tubing would enter. The remaining part of the tube, which is connected to the pump, can be seen in (d) behind the drainage tube also highlighted in blue.

Continuing slightly under where the pot is positioned you will see a mesh where a potential filter would be placed, seen above the drainage tube in (d). This is meant to separate the soil so that clean water can return to the reserve shown on the right in figure (a) in blue. For the reserve, you can see on most visuals a hole where new water would enter to refill it. In (d) we can more easily see the potential placement of where the ultrasonic sensor could be in order to accurately measure water levels. Staying with visual (d) in mind, we can see a small tube, towards the bottom, in blue going from the reserve straight to the pump located in the central part of the shell.

The lower part of the shell is located straight below where the plant would be and houses most of the electrical components needed for the project to function. In green we have our PCB with the MCU, voltage regulator, and pump driver. Outside the PCB we have the actual pump, charge controller, and battery. Finally, behind the shell, seen more clearly in visual (b) and (c), we have our solar panel which will be our main energy source for the project.

Chapter 3 - Research and Investigation

3.1 Processing and Control

3.1.1 MCU, MPU, & FPGA

An integrated circuit is required to process the inputs from the sensors and output data to the web app and control the motors. Several integrated circuits were initially considered: microprocessor, microcontroller, and programmable logic device. Microprocessors provide better performance than microcontrollers. Microprocessors excel at performing a specific set of instructions and pipeline via a unique architecture. Microprocessors often have a larger size which allows more logic components for data processing and pipeline stages. Microprocessors also usually have higher frequency clocks than microcontrollers. However, in exchange for better performance, microprocessors consume more energy than microcontrollers and are more expensive [53].

Field programmable gate arrays (FPGAs), the most well known types of programmable logic devices (PLDs), contain large copies of silicon cells and wiring which require engineers to define the hardware logic. This may improve performance by allowing the engineer to tailor the hardware to a specific project/application and also allows multiple operations to occur in parallel. FPGAs generally have better performance than both microprocessors and microcontrollers [54]. However, like microprocessors, FPGAs

consume more energy, largely due to switching activity. Although FPGAs usually have lower energy consumption than microprocessors, both consume higher power than microcontrollers [55]. Although power consumption can be reduced via clock and voltage gating, this increases an already complex and relatively stochastic design. In addition to defining hardware logic, the software must still be written. FPGAs are also much more expensive and larger than microcontrollers [54].

The most important feature of microcontrollers is the population of peripheral components on the chip or module at the time of manufacturing. Microcontrollers generally consist of a processor chip along with peripherals such as ADCs, DACs, clocks/oscillators, memory units, and even RF subsystems. In contrast, these components might need to be purchased and interfaced with a microprocessor. FPGAs with on-board peripheral components may be purchased, but these are extremely expensive. A microcontroller was selected for this system as the integrated circuit for this system due to the goals of power efficiency, low cost, support for sensor peripherals, and convenient wireless communication.

Table 3.1.1.1 Microprocessor, Microcontroller, FPGA Comparison.

	Microprocessor	Microcontroller	FPGA
Energy Consumption	Highest	Lowest	High
Cost	High	Lowest	Highest
Processing Speed	High	Lowest	Highest
Design Complexity	High	Lowest	Highest
Built-in Peripheral Support	Lowest	Highest	Low
Size	Large	Smallest	Largest

3.1.2 Microcontroller Products

As a central component of the system, the specifications of the chosen microcontroller must be in accordance with all the objectives described in Section 2.3. The microcontroller should have capability for wireless communication with the web app. The microcontroller should also have at least one low-power mode in order to reduce power consumption, but more modes are more desirable.

3.1.2.1 ESP32 (Espressif)

An ESP DEVKIT V1 was available for immediate use, so the ESP32 family of microcontrollers was first researched. This development kit uses the ESP32-WROOM-32 module which contains the ESP32-D0WDQ6 chip. The purpose of section 3.2.3 shall be to compare various SoC modules to determine if the performance of other

microcontrollers outweighs the cost and convenience benefit provided by using the ESP32-WROOM-32.

The ESP32-D0WDQ6 supports the 802.11 b/g/n standard and Bluetooth for on-chip wireless communication. The chip contains a 31.25 kHz internal RC oscillator, but has pins to support an external 32 kHz crystal oscillator. Like most ESP32 modules, four low-power modes are provided: modem-sleep, light-sleep, deep-sleep, and hibernation. In modem- and light-sleep modes, a WiFi connection may be maintained between the module and an access point. The module will wake up before the access point periodically broadcasts a delivery traffic indication message, which would allow devices on the network to establish a connection with the module [71]. In light-sleep mode, the ultra-low-power processor remains active, whereas the main processor and other components are clock-gated [72].

The respective Espressif datasheets were used to acquire information regarding the ESP32 modules [73] [74] [75] [76]. Generally, ESP32 modules with on-chip wireless capabilities will support similar standards and utilize the same power saving methods. The presence of peripherals and interfaces relevant to this project also remain the same. The main differences between ESP32 modules would include cost, memory, processor speed, and power consumption specifications. The following table compares various ESP32 modules with on-chip WiFi capability. All microcontrollers had a multi-channel 12-bit SAR ADC. The specifications for the various ESP32 microcontrollers were approximately the same. Notably the ESP32-C6 was the most power-efficient, but also had the slowest processor. It should also be noted that the ESP32-D0WD-Q6 (the chip on the ESP32-WROOM-32) does not have above 4GB of flash, like the other microcontrollers. However, the ESP32-D0WD-V3 does reach 8-16GB of flash. It would be trivial to replace the SoC module from the development board with a different module if more memory is required within the system. Ultimately, the ESP32-WROOM-32/D0WD-Q6 was selected from the possible ESP32 microcontrollers due to the lower cost of the development boards and similarity in features and power specifications.

Finally, the ESP32-WROOM-32 was selected over the ESP32-D0WD due to its built-in (presoldered) components despite a slightly higher cost.

Table 3.1.2.1 ESP32 Microcontroller Feature Comparison.

	ESP32 - D0WD	ESP32-C6	ESP32-PICO	ESP32-S3
Dev Kit Cost	$0	$9-14 [77]	$10 [78]	$9.39-13.30 [79]
SoC (PCB) Cost	$1.50-$3 [80]	$1.85-3.15 [77]	$2.50-4.50 [81]	$1.85-2.49 [82]
Processor(s)	240MHz Xtensa	160MHz RISC-V	240MHz Xtensa	240MHz Xtensa dual-core LX7

	dual-core LX6 [73]	20MHz RISC-V [74]	dual-core LX6 8 MHz ULP [75]	ULP RISC-V & FSM coprocessors [76]
WiFi Speed	150 Mbps [74-77]			
Memory Flash: ROM: SRAM (HP/LP):	2/4 MB* [80] 448 KB 520/16 KB [73]	4/8 MB [77] 320 KB 512/16 KB [74]	4/8 MB [81] 448 KB 520 KB [75]	2/4/8/16 MB [82] 384 KB 512 KB [76]
ULP - ultra-low power FSM - finite state machine * ESP32-D0WD-V3 has 8-16GB flash capabilities				

Table 3.1.2.2 ESP32 Microcontroller Power Consumption Comparison. [73-76]

	ESP32 - D0WD-Q6	ESP32-C6	ESP32-PICO	ESP32-S3
VDD (Power Supply; nominal)	3.3 V	3.3 V	3.3 V	3.3 V
I_VDD (Power Supply; min.)	0.5 A	0.5 A	0.5 A	0.5 A
WiFi TX/RX (Active Mode)	240/95 mA	354/82 mA	370/120 mA	340/91 mA
Modem-Sleep	20-68 mA	14-38 mA	20-68 mA	13.2-107.9 mA
Light-Sleep	0.8 mA	35-180 uA	0.8 mA	0.24 mA

Table 3.1.2.3 ESP32 Module v. Chip Comparison

	ESP32-WROOM-32	ESP32-D0WD
Built-in RF Antenna	Yes	No
Built-in Oscillator	Yes	No
Built-in Flash Module	Yes	No
SoC (PCB) Cost	$4.84-$5.71 [121]	$3

3.1.2.2 STM32 (STMicroelectronics)
STMicroelectronics offers the STM32 family of microcontrollers, consisting of four lines of MCUs: high performance, mainstream, low-power, and wireless. The wireless STM32 microcontrollers are the only products that support wireless communication. However, likely due to the compactness and emphasis on low power consumption, the wireless STM32 microcontrollers do not support IEEE 802.11 WiFi standards, instead supporting only a combination of 802.15 task groups. It would still be possible to transfer data between the STM32 and a web server using IEEE 802.15 or to add a Wi-Fi connectivity module to the microcontroller. However, the lack of built-in or on-chip compatibility with IEEE 802.11 increases the research, design, and debugging time and cost of materials compared to the ESP32. The lower power consumption in the wireless STM32 microcontrollers do not offset the increase in time and cost.

3.1.2.3 nRF (Nordic Semiconductor)
Nordic Semiconductor offers many microcontroller products with wireless communication capabilities, such as the nRF52840 and nRF5340, but these are limited to IEEE 802.15 standards (e.g. Bluetooth, WPAN) [83]. A companion integrated circuit which allows a SoC to communicate via WiFi, such as the nRF91 series, would be necessary to add to the system [84]. Thus, the nRF family of microcontrollers was not researched further.

3.1.2.4 CC32xx (Texas Instruments)
The TI SimpleLink microcontroller series provides on-chip WiFi capabilities in a network subsystem containing a dedicated ARM processor [88-90]. The subsystem supports a TCP/IP protocol stack and is compliant with IEEE 802.11b/g/n standard [88-90]. The microcontrollers also contain on-chip 32KHz crystal and RC oscillators [88-90]. Three SimpleLink microcontrollers were initially considered: CC3200, CC3220, and the CC3235. However, the price of the lowest-cost CC3200 and CC3235 modules more than twice the ESP32 modules discussed earlier–$11.42 and $9.09, respectively [85] [86]. The price of the lowest-cost, in-stock CC3220 module was $5.37 for the CC3220RM2ARGKR [87]. From these choices, only the CC3200 was researched further.

The CC3220 has four low power modes associated with the main application processor: sleep, low-power deep sleep (LPDS), hibernate, and shutdown [89]. As a separate subsystem, the network processor has a “connected idle” mode, in which the processor is in active listen mode during Wi-Fi beacon transmission instances and is in LPDS mode between these instances (every ~104 ms) [91]. The network processor’s sleep and “idle connected” mode (active/LPDS) is compatible with the active, sleep, and LPDS modes of the application processor [91]. If the application processor is in hibernate mode, the low-power modes of the network processor subsystem are not available [91].

3.1.2.5 PIC32/WFI32 & ATSAMW (Microchip Technology Inc.)
The PIC32MZ and WFI32 microcontroller families support on-chip WiFi communication compliant with the IEEE 802.11 b/g/n standard [95-96]. WFI32 products consist of a PIC32MZ MCU with a front-end module [95]. The ATSAMW25 module also supports

on-chip WiFi communication compliant with the IEEE 802.11 b/g/n standard [96]. Microchip also provides the MPLAB Harmony software framework, which includes TCP/IP Stack libraries available for the PIC32MZ family [95]. The prices of the lowest-cost, in-stock modules were $4.31, $5.80, and $15.30 for the PIC32MZ1025W, WFI32E02UE-I, and ATSAMW25H18, respectively [92-94]. Thus, only the PIC32/WFI32 families were researched further.

The PIC32MZ family supports four low-power modes: dream, sleep, deep sleep, and extreme deep sleep. The family also supports low-power Wi-Fi modes: wireless sleep mode (WSM) and wireless deep sleep (WDS) [95]. The modules also provide an on-chip 32.768kHz low-power RC oscillator as a clock source [95]. In the WSM, the RF is placed in a Sleep mode where the primary oscillator remains on, but RF components are clock-gated [95]. In the WDS, the primary oscillator is turned off if no other peripherals request it and the RF components are shut down [95]. In both modes, the wake-up source includes the arrival of WiFi data from the access point.

Unlike the CC3220, the network subsystem of the PIC32MZ shares the same processor with other subsystems of the MCU [95]. Due to this, the WiFi sleep mode controller (SMC) will enter Sleep mode if the MCU enters Sleep mode and the MCU will exit Sleep mode if the WiFi SMC exits Sleep mode [95].

3.1.2.6 Selection of Microcontroller

The ESP32-WROOM-32 module will be selected as the microcontroller. The features of this module include support for timer- and Wi-Fi- based interrupts, native TCP/IP support, low power consumption, and, most importantly, low cost of development products.

Table 3.1.2.6 Select Microcontroller Cost Comparison.

	ESP32-WROOM-32	CC3220	PIC32MZ
Dev Kit Cost	$0	$47.99 [97]	$73.04 [98]
SoC (PCB) Cost	$4.84-$5.71 [121]	$5.37	$4.31

After comparing the microcontrollers with on-chip WiFi and low-power modes, it was clear that the ESP32-WROOM-32 is the least cost prohibitive choice. Even if the development kits for the ESP32 were not freely available to use, the development boards for the TI CC3220 and Microchip PIC32MZ are several times more expensive than the price of a ESP32 microcontroller development kit [99]. The ESP32-WROOM-32 microcontroller was selected as the microcontroller for this project because of the capabilities for on-chip WiFi protocol, low-power modes, wireless and timer interrupts, and low cost.

3.1.3 Microcontroller Clock Source

Microcontrollers exit low-power modes due to interrupts. Interrupts are core features of microcontroller architecture. They allow for the usage of multiple peripherals with the microcontroller. The capability for timer-based interrupt is required in the desired microcontroller in order to check the soil moisture and water the plant in specific intervals. The uptake of water by plants is a long process, often taking multiple hours, so it would be redundant and power-inefficient to constantly measure the soil moisture.

In order to generate timer-based interrupts, an internal or external clock source is necessary. The table below compares multiple types of clock sources: crystal, crystal oscillator, ceramic resonator, integrated silicon oscillator, and RC oscillator. EMI and humidity sensitivity are the most important factors. The clock will be adjacent or within a pot containing a watered plant and soil, making any exposed componentry susceptible to damage from humidity or wetness. The microcontroller to which the clock is connected will also be transmitting ~2.4GHz electromagnetic waves in order to communicate wirelessly. Corruption of the clock due to these factors would ruin the component and prevent the system from operating. High accuracy and low power consumption are also important features. While high accuracy is necessary to water the plants at the correct time, the accuracy of the clock can be improved via software methods, such as real-time clock correction/calibration or time synchronization. On the other hand, power consumption of the clock is an innate feature of the physical component.

Table 3.1.3.1 Clock Source Technology Comparison. [61]

Clock Source	**Accuracy**	**Power**	**Cost**	**External Comp.**	**EMI/Humidity Susceptibility**
Crystal (XTAL)	~±10–50 ppm [56]	Low	Low–Medium	Yes (XTAL + caps)	Yes/Highly
Crystal Oscillator Module	~±5–20 ppm [57]	Medium	Medium–High	No (all in one module)	No/Moderately Insensitive
Ceramic Resonator	~±0.5–1% [58]	Low	Low	Sometimes internal caps	Yes/Moderately Sensitive
Integrated Silicon Oscillator	~±1–3% [59]	Very low	None	No	No/Very Insensitive
RC Oscillator (Internal)	~±1–10% [60]	Very low	None	No	Often Sensitive

Usually two RC clocks (one low- and one high-frequency clock) are included within microcontroller chips/modules that contain at least one low-power mode. Although RC oscillators have notable disadvantages with low accuracy and sensitivity to interference, these can be mitigated or circumvented. Since wireless communication is already being utilized, the current time from a more accurate source, such as a web page/server, can be recorded within the server for the web application. When the microcontroller wakes up, the current time can be given to the MCU, which performs a simple calculation to measure the clock drift of the RC oscillator and recalibrate the oscillator or modify the length of time that the MCU is placed in a low-power mode. In terms of sensitivity to EMI, the clock used for low-power mode will not be active while the microcontroller is transmitting data to the network. Although wireless communication may be active during the low-power mode in order to provide Wi-Fi wakeup source, only reception will occur and the period of activity will likely be too short to drastically affect the oscillator. Finally, humidity is a factor that may affect all components, including vital microcontroller circuitry, in the system; humidity must be kept out of the system anyways.

A crystal oscillator module could help balance the loss of accuracy and high sensitivity of the RC oscillator. If the RC oscillator becomes too inaccurate or unstable, then the monitoring and irrigation of the soil will not work properly. At this point, crystal oscillator modules would have to be used in order to maintain system functionality, despite the fact that crystal oscillator modules consume more power than RC oscillators. While crystal oscillator modules will increase cost and area usage of the PCB, these values are negligible. However, the main difficulty in using crystal oscillator modules is the input voltage/current specification ranges. These modules usually require a narrow range or small maximum for the input current (e.g. 50 uA max current). Microcontroller general-purpose output pins usually are around 10s of mA. This necessitates the careful design of a current divider circuit which must output a consistent, very small amperage, while not causing significant voltage loss. If permitted by time, dimension, and budget constraints, a crystal oscillator module may be placed on the circuit board.

In conclusion, an RC oscillator will be selected as the primary clock source due to built-in support provided by Espressif on the ESP32-WROOM-32 module. Software modifications will be made in an attempt to correct the accuracy of the timer.

3.2 Sensors and peripherals

3.2.1 Pumps

Water pumps play a critical role in the operation of C.H.A.N.O.S., serving as the primary mechanism for the transportation of water from the storage source to the plant. The pump ensures that the water is delivered at the correct flow rate and pressure to maintain the correct soil moisture levels based on feedback from the moisture sensor. Selecting an appropriate pump is crucial for maintaining a low energy consumption, cost, and overall reliability. In the context of C.H.A.N.O.S., various types of pumps may be suitable for our needs depending on the scale of our system, design, and the environment it will be

operating in. This includes the depth of the water source, required flow rate, power availability, and control complexity. In this case we will be looking at 4 pumps, the centrifugal, positive displacement, submersible, and peristaltic pumps.

Table 3.2.1.1 Pump Comparison.

Pump Type	How It Works	Average Flow Rate (LPM)	Advantages	Disadvantages	Cost ($)
Centrifugal	Uses an impeller to create centrifugal force	1 - 10	- Simple design - Continuous smooth flow - Low cost - Easy maintenance	- Loses efficiency at low or high flow - Needs priming	20 - 80
Positive Displacement	Moves fixed amount of liquids per cycle using gears, pistons, and screws	.5 - 5	- Precise flow rate - Works at low flow rates - Handles high pressure	- More complex mechanical structure - High maintenance - Higher cost	30 - 100
Submersible	Fully submerged pushing water upwards	1 - 10	- No priming needed - Quiet - Efficient - Compact	- Harder to access for maintenance - Higher risk of seal failure	20 - 100
Peristaltic	Squeezes flexible tubing to move water	.05 - 2	- Very precise - Self primes	- Low flow rate - Tubing wear	10 - 100

3.2.1.1 Centrifugal Pumps

Centrifugal pumps are amongst the most popular types of pumps in the industry. At its core a centrifugal pump is a mechanical device designed to pump fluid by converting rotational kinetic energy into hydraulic energy. They have a wide variety of applications in industrial, domestic, and agricultural areas due to their versatility and efficiency. They work by using a rotating impeller to make a centrifugal force that moves water through the pump and into the discharge pipe. Impellers act as the heart of the pump and are characterized by their wheel-like shape with curved blades at the end, as it spins it throws water outwards using centrifugal force.

There are three main mechanisms to a centrifugal pump, this includes the impeller, a casing, and a shaft. The casing is a stationary component that surrounds the impeller, guiding the water to its discharge point. The shaft is what connects the impeller to the motor that rotates it. One of the main advantages of centrifugal pumps is its ability to provide a continuous flow of water with little pulsation. Pulsation refers to the fluctuations or variations in water flow. Centrifugal pumps are better suited for low-viscosity fluids, in this case water, and perform best when operating near their design flow rate, which for small scales is between 5- 50 gallons per minute. Another advantage of the pump is its compact shape, needing only the few parts mentioned to operate. It is also relatively cost effective, cheaper than other pumps, and comes in a wide variety of sizes. Some of the disadvantages associated with it though is that it does not do well with high viscosity fluids, which in this case should not be a problem. It also has reduced efficiency at low flow rates and performs best when at the designed flow rate. [31]

3.2.1.2 Positive Displacement Pumps

Positive Displacement pumps are made to ensure that the pumping action is moving forward, thus the name 'positive'. It operates by physically trapping a fixed amount of fluid then forcing it through a pump then into a system. Unlike other pumps, which may rely on high speed impeller, positive displacement pumps move water by mechanically displacing it with pistons, gears, and diaphragms. These components allow the pumps to deliver a precise volume of fluid with each cycle.

One of the big advantages of displacement pumps is their ability to handle both low and high flow rates while keeping a consistent output. This means that you can either pump a small or large amount of water precisely, like using a needle to inject specific amounts of fluids. Another advantage is its ability to self prime, which just means flooding the system to make it devoid of air. Some of the disadvantages stem from the amount of parts used, making the pump more mechanically complex, the higher cost, and occasions of pulsating flow. [32]

3.2.1.3 Submersible Pumps

Submersible pumps are made to operate while fully submerged in water. Unlike a surface mounted pump, which gets its water through suction, submersible pumps are placed directly into the water source, like tanks and wells, pushing water upwards. This allows it to avoid many of the suction related limits found in other pump designs like priming and air bubbles. These pumps are normally used when a water source is located below ground where a compact and quiet design is required. [33]

One of the primary advantages of submersible pumps is their ability to operate efficiently without the need for priming, since they are always submerged and filled with water. This makes them particularly reliable for continuous watering systems. Additionally, submersible pumps are quieter than surface-mounted pumps, as the water surrounding the pump dampens operating noise. They are also effective at handling medium to high flow rates and can be used in both small and large-scale systems depending on the model. However, because submersible pumps operate underwater, they can be more difficult to

access for repairs, and potential water intrusion can damage electrical components if seals fail. [33]

For small scale projects, submersible pumps suitable for moving 1 to 10 liters per minute are widely available. Basic costs typically range from $20 to $100 USD, depending on the flow rate, head height, and build quality. More advanced or larger submersible pumps used for deeper wells or larger irrigation systems can cost several hundred dollars or more. In general, submersible pumps offer a compact, reliable, and simple solution for water delivery in systems where the water source can accommodate submerged equipment.

3.2.1.4 Peristaltic Pump

Peristaltic pumps use a simple yet highly effective mechanism that involves compressing and releasing flexible tubing to move fluid. Inside the pump, a set of rollers, rotate along the flexible tube, squeezing the liquid forward in controlled, isolated pockets. As each roller moves away, the tubing expands back to its original shape, drawing in more fluid to continue the pumping. They provide a simple solution for moving many types of fluids, and can be incorporated into a variety of industry applications.

One of the primary advantages of peristaltic pumps is their ability to deliver precise and consistent flow rates, even at very low volumes. This makes them particularly well suited for applications requiring accurate dosing, where small, controlled amounts of water are delivered directly to plant roots. Additionally, peristaltic pumps are self priming and can safely run dry without damage. Running dry refers to little or no water within the tubes. They are also capable of handling more dense fluids, which can be beneficial in certain agricultural setups.

Peristaltic pumps do have some limitations though. Their flow rates are generally lower compared to other pump types, making them unsuitable for large scale water delivery. Over time, the flexible tubing inside the pump can wear out and require replacement, adding to maintenance costs. Additionally, while the initial purchase price for small scale units is relatively affordable, typically within $50 to $300, the cost per volume of water moved is higher than for most centrifugal or submersible pumps. Despite these drawbacks, peristaltic pumps remain a valuable option for projects that prioritize precision, control, and ease of integration.[34]

3.2.2 Pump Selection

In Section 3.1.5, we introduced and analyzed several types of pumps commonly used in fluid handling systems, including centrifugal, positive displacement, submersible, and peristaltic pumps. Each pump type offers distinct advantages and drawbacks in terms of flow rate, installation complexity, maintenance requirements, and overall cost. By examining these factors, we gained a clearer understanding of the trade-offs involved in selecting an appropriate pump for a given application.

In this section, we will apply that knowledge to the specific needs of the C.H.A.N.O.S. project. Our system requires reliable water delivery, energy efficiency, and

cost-effectiveness, making pump selection a critical design decision. Based on our earlier evaluations, we have chosen to focus our part selection on submersible and peristaltic pumps. These options fit best with the functional and spatial limitations of the project. Other pump types, such as centrifugal pumps, were excluded due to key disadvantages. Specifically, for centrifugal pumps, their large size and excessive noise, which make them impractical for our intended use. The following subsections will explore the selected pump products in greater detail and justify their relevance to our final design.

3.2.2.1 Submersible Pumps

The first pump that we will be reviewing is the Whale GP1002, manufactured by Munster Simms Engineering Ltd. in the United Kingdom. Priced around $36 on Amazon [45], the GP1002 was evaluated for its compatibility with the requirements of the C.H.A.N.O.S. project. The system requires a consistent and efficient water delivery from an external container. The Whale GP1002 submersible pump provides this by meeting several technical criteria. It operates at around 12V DC, making it compatible with our small scale battery system. It operates best when the tubing is at level with the pump (i.e., level with the water source at 0 meters), delivering 10.3 liters per minute with a current draw of 2.4 amps. When the delivery height increases to 1 meter, the flow remains practical at 8.75 liters per minute with a current draw of 2.2 amps. This allows the project to maintain functional water delivery even at modest heights, which is especially useful for the project where the verticality is not super high.

In addition to meeting electrical and flow requirements, the Whale is lightweight in design at around .15 kg with a 10 mm bore hose, fitting the physical constraints of what we intend for the system. The dimensions of the pump are important, as we discussed in the 3.2.4 introduction, something that is too big is undesirable as we have a limited amount of space to work with. The dimensions for this pump are small at 12.7 mm x 31.5 mm x 49.5 mm, an aspect which is especially appealing for our project. Finally the pump’s construction materials, includes ABS plastic, nitrile seals, and a stainless-steel impeller, providing durability and a reasonable level of resistance to corrosion and wear, both of which are beneficial for a long-term, low-maintenance system. [44]

The second submersible pump we will talk about is the SEAFLO inline pump which offers a compact solution for water transfer in low head applications. Priced at $44 it is designed to operate at 12V DC, this pump delivers a flow rate of up to 12 liters per minute at level with the pump, and about 9.5 liters per minute 1 meter above the pump. This aligns well with the project's watering requirements. It functions as both a fully submersible or inline pump, which provides flexibility in installation, especially when the space or tank shape becomes a constraint. With a current draw of 3.1 amps 1 meter from the pump, it remains within the specs of our battery system.

The SEAFLO pump also has a lightweight design, quiet operation, and minimal maintenance requirements. It's made of durable materials such as thermoplastic housing and stainless steel shaft, which resist corrosion from long term water exposure. The pump can tolerate brief dry run conditions, an important feature in case of sensor failure or unanticipated dry tanks. Additionally, the manufacturer provides clear performance

curves and installation documentation, which is useful for power budgeting and system planning. Given these factors, the SEAFLO pump stands out as a strong candidate for compact, battery powered plant watering systems where moderate flow and vertical lift are needed. [46]

3.2.2.2 Peristaltic Pumps

The first Peristaltic pump we'll talk about is the Adafruit peristaltic liquid pump. This pump is a compact product that is suitable for low volume, precision water delivery, a benefit for the project if we wanted to do targeted irrigation or dosing. It operates at 12 volts and draws about 200 - 300 mA, making it very compatible with our existing power framework. The low current draw also helps reduce heat buildup and power supply stress during extended operation. The flowrate of the pump reaches 100 mL per minute, slower than the submersible pumps but advantageous if we seek to avoid overwatering.

It uses a 4 mm in diameter tube making it easier to route it in compact designs. The full length of the tubing is about half a meter, which may need to be expanded upon depending on our specs. The weight of the pump is .2 kg, making it a very lightweight option. While it's more niche compared to higher capacity pumps, it serves as a good option where moderate, specified water delivery is needed. [47]

The final pump we will talk about is the CONQUERALL peristaltic pump. Priced at $12 for 2 pumps, it is the most affordable pump that we will talk about. It operates at 5 volts and draws approximately 0.4 amps during regular operation, with a noted startup surge of up to 3 amps. With a maximum flow rate of 150 mL per minute, it is designed for small scale liquid delivery, making it a potential fit for the project's goal of controlled plant watering. The pump is self priming and can draw up water half a meter below the main pump, giving it flexibility in its potential placement.

Its small size and relatively low power consumption makes it compatible with our power system. There are some downsides though such as a low maximum flow rate which may not be suitable for handling larger watering cycles. It also has a high inrush current of 3 amps, which we need to consider when paired with other components. Overall the CONQUERALL pumps strongest quality is both its cost and suitability for low flow irrigation systems [48]

3.2.2.3 Conclusion For Pump Selection

Table 3.2.2.3.1 Pump Brand Comparison.

Pump	Voltage (V)	Current (A)	Flow Rate (L/min)	Suitability	Cost ($)
Whale GP1002 (Submersible)	12	2.2	8.75	High flow	36
SEAFLO	12	3.1	9.5	High flow	44

SFSP1 (Submersible)					
Adafruit (Peristaltic)	12	.2 - .3	.1	Low flow precise delivery	24.95
CONQUERALL (Peristaltic)	5	.4 (3A startup surge)	<= .15	Very low flow, low voltage	12 (2 pack)

Based on all the information we gathered, we have decided to choose the Adafruit pump as it is better suited towards what we think the project needs in terms of flow rate and power specifications and cost.

3.2.3 Light Sensor

Light sensors are electronic components that detect and respond to different levels of light in the environment. They play a critical role in a wide range of modern technologies from smartphone brightness adjustment, automatic street lights, to advanced agricultural systems. In the context of C.H.A.N.O.S., light sensors are particularly useful for monitoring sunlight exposure, helping the system understand when and how long a plant has been in direct light. This information can be used to optimize watering schedules and keep the plant safe during peak solar hours.

The core function of a light sensor is to convert photons into an electrical signal that an MCU can interpret. Some light sensors do this by changing resistance, while others produce a current and/or voltage when exposed to direct light. Depending on the application different sensor types offer differing levels of sensitivity, speed, and precision. Understanding the pros and cons of these sensors is important when trying to select the right one for our project. The following sections will explain some different types of solar sensors such as light dependent resistors, photodiodes, and phototransistors.

3.2.3.1 Light Dependent Resistors

A Light Dependent Resistor, also known as a photoresistor, is one of the simplest and most widely used types of light sensors. It is made from a semiconductive material whose resistance decreases as light intensity increases. In darkness, the sensor resistance is high, often in the megaohm range. As light hits the surface, the material absorbs photons, freeing electrons and significantly lowering the resistance, sometimes to just a few hundred ohms.

Light dependent resistors are ideal for basic light detection in systems where precise light measurement is not required. They output an analog signal that can be read by a microcontroller, such as an ESP32, to detect changes in ambient light. This makes them especially good for applications like turning on lights at dusk, tracking day/night cycles, or estimating how much sunlight a plant receives.

They are great for many and low cost projects due to their simplicity, low cost (under $2), and ease of integration. However, they do have some downsides. Their response time is relatively slow compared to photodiodes or phototransistors, and they are less accurate in changing or low light conditions. In addition, their performance can go down over time due to exposure to strong heat. Despite these trade-offs, these light sensors are an effective choice for plant care systems that need general light monitoring rather than high-precision measurements. [35]

3.2.3.2 Photodiodes

A photodiode is a type of light sensor that converts light into electrical current using the photoelectric effect. It operates like a semiconductor device and is made of silicon that generates a small electrical current when photons strike its surface. The greater the light, the greater the current produced. Unlike an LDR, which varies resistance, a photodiode is designed for speed and precision, making a great choice for applications that require fast and accurate light detection. In a plant watering system, a photodiode can be used to detect rapid increases in sunlight, such as peak sun hours, and signal the system to temporarily stop watering or inform the user to move the plant. Their fast response time makes them suitable for systems that need to quickly adapt to changing lighting conditions.

Despite their benefits, photodiodes produce a very small current, typically in the microamp range, so they often need amplification circuits to make the signal usable for microcontrollers. This adds some complexity compared to simpler sensors like LDRs. Photodiodes are also directional, meaning they respond best when facing directly toward the sun. In spite of these downsides, photodiodes are reliable, long-lasting, and highly sensitive, making them a good choice for projects that demand more accurate light sensing. [36]

3.2.3.3 Phototransistors

A phototransistor is a type of light sensor that functions like a photodiode, but with the difference being it has a built in amplification. This means that when light hits the phototransistor's base, it allows current to flow from the collector to the emitter, producing a much stronger output signal than a photodiode. As a result, phototransistors are more sensitive to light and are better for applications where a larger signal is needed without adding external amplifiers.

Phototransistors are ideal for detecting changes in light intensity and can be used when faster response time is needed. It is also great at detecting light exposure during peak sunlight hours, helping to assess whether a plant needs to be moved. Compared to LDRs, phototransistors respond more quickly and are less affected by temperature and aging, making them more reliable in the long term. [37]

Despite their advantages, phototransistors still have some limits. They are directional, meaning they must be aimed toward the light source for good performance. Additionally, while they are more accurate than LDRs, they do not typically provide light readings in physical units like lux without additional circuitry. Overall, phototransistors are a

versatile middle ground between basic LDRs and more complex ambient light sensors. They offer a good balance of speed, sensitivity, and ease of use, making them a strong choice for systems that need reliable light detection.

3.2.3.4 Light Sensor Selection

In section 3.1.5 we discussed what light sensors were and the different types that existed. From these we talked about light dependent resistors, photodiodes, phototransistors, and digital sensors. Each has their own pros and cons, and by using that data we can now decide on the best component for our project. Despite the resistor, diode, and transistor being much simpler, from the perspective of trying to comply with PCB requirements, only digital light sensors will be considered in this section. [49]

The first digital light sensor we will talk about is the BH1750. It is a digital ambient light sensor that measures illuminance in lux and communicates data using an I2C interface. It is widely used in embedded systems due to its accuracy, ease of integration, and low power consumption. Unlike analog sensors like photoresistors, the BH1750 provides direct lux readings, eliminating the need for calibration or analog to digital conversion circuits. The sensor operates over a wide range (1 to 65,535 lux), making it a good choice for environments like dim indoor lighting to bright outdoor sunlight. This makes it a strong candidate for our C.H.A.N.O.S. project, where it's critical to detect and respond to intense light levels that could stress the plant.

The BH1750 is compact, reliable, and communicates using only four pins (VCC, GND, SDA, and SCL), simplifying PCB routing. Its low operating voltage (2.4 - 3.6 V) and minimal current draw (120 μA) also make it suitable for low power, battery operated systems. Libraries for the BH1750 are available for platforms like Arduino and ESP32, allowing flexible software integration. Due to its balance of accuracy, size, and low complexity, the BH1750 is a great sensor choice for our light monitoring system. [49]

The second light sensor we will talk about is the TSL2561 light sensor that measures both infrared and visible light to provide precise light readings in lux (.1 to 40,000). Unlike simpler sensors like LDRs or basic photodiodes, the TSL2561 features dual photodiodes, one sensitive to full spectrum light and one to infrared alone. By comparing these two readings internally, it can filter out the IR component and provide a result that closely matches the human eye’s perception of brightness. This makes it a great choice for applications where precise ambient light monitoring is needed.

The sensor communicates via the I2C protocol, making it easy to integrate with microcontrollers and is suitable for PCB layouts. It supports both manual and automatic gain control, allowing it to adapt to very dark or very bright environments without losing accuracy. However, the TSL2561 does require a bit more setup in software, and its output is not base lux, you'll need to apply the manufacturer’s formulas or use a library that handles it. Despite this, it remains a strong alternative for projects that require greater control and filtering precision. For the C.H.A.N.O.S. project, it could be considered if the environment has mixed lighting. [50]

3.2.3.5 Conclusion To Light Sensor Selection

Table 3.2.3.5.1 Light Sensor Comparison.

Features	**BH1750**	**TSL2561**
Interface	I2C	I2C
Output	Direct lux reading	Infrared and full spectrum
Lux Range	1-65,535 lux	.1-40,000 lux
Power Consumption	120 uA	200 - 300 uA
Operating Voltage	2.4 - 3.6	2.7 - 3.6
Suitability	Good for general light detection needs	Better for mixed or filtered light environments

After evaluating these light sensor options, we selected the BH1750 for our project due to its accurate lux readings, low power consumption, and ease of integration with our PCB design. Its simple I2C interface and strong support in development environments made it more practical and easier to understand versus the tsl2561, which in short terms was considered as overkill for our project needs.

3.2.4 Water Level Sensor

A water level sensor is an important component to C.H.A.N.O.S. because it will measure the amount of liquid in our water reservoir. Choosing the right type of sensor for this task is critical as the sensor should provide real time information about the amount of water in our tank. The data it transmits allows the system to make informed decisions about when to activate, conserving resources, or alerting the users when refilling is necessary. By incorporating this water level sensing, the system can act more independently, preventing dry running of pumps, and ensure no interruptions in irrigation.

There are several types of water level sensors, each with their own pro's and con's that could work for C.H.A.N.O.S.. Choosing one depends on factors such as size, weight, accuracy, maintenance, power consumption, and overall cost. Below we will go over three types of sensors that could go well for the project. This includes the float, ultrasonic, pressure, and capacitive sensors.

Table 3.2.4.1 Water Level Sensor Comparison.

Sensor Type	Measurement Type	Contact With Water	Accuracy	Cost ($)	Pros	Cons
Float Vertical	fixed	yes	moderate	5-15	Compact, simple, low	Needs vertical

					cost	space, fixed trigger point
Float Horizontal	fixed	yes	moderate	5-15	Side mounted, easy install, low cost	Single level detection
Cable Float	continuous	yes	low moderate	10-30	Adjustable, durability	Needs space to swing, low precision
Ultrasonic	continuous	no	high	10-50	Non contact, accurate, easy to integrate.	Affected by mist and splashing
Pressure	continuous	yes	high	20-100	Accurate, works with a variety of geometries	Must be submerged, needs waterproofing + calibration
Capacitive	Fixed + Continuous	yes +no	high	30-150	No moving parts, very accurate	Higher cost, sensitive to interference

3.2.4.1 Float/Switch Sensor

A float sensor can be described as a liquid contact sensor which uses a float to operate a switch. They can be a good option for many projects because they are cost effective, reliable, and easy to use. There are several different types of float sensors including the vertical, horizontal, and a cable float switch just to name a few.

- Vertical Float Switch: consists of a hollow ball that moves up and down along a vertical rod. Inside the stem there is a magnetically activated reed switch. As the float rises or falls, with the water level, a magnet inside the float triggers, making either an open or closed electrical circuit. They are ideal for compact systems, have a low cost, and are accurate in their sensing. Some of the downsides though is that the trigger points for the switch is predetermined, meaning that it is not adjustable, and it is susceptible to build up, needing an occasional cleaning. Overall though it is a great choice for precise water level detection. [39]

- Horizontal Float Switch: a switch that detects water levels at a fixed point in the tank. Typically it is mounted on the side of a wall, with a floating arm extending into the tank. When the water level falls, the float, which is on a hinge, will position itself perfectly horizontal, sending a signal that the water level is below a

fixed point on the tank. These pumps are especially useful for tasks like pump cutoff, overflow prevention, or low level alerts in small scale watering systems. While they are not ideal for continuous level measurement they provide a cost effective, and reliable way to measure single level detection, which for the C.H.A.N.O.S. project would work great. [40]

- Cable Float Switch: a flexible and durable sensor used to detect water levels in a certain range rather than at a fixed point. It consists of a sealed buoyant float that is attached to a cable or rod, which rises and falls with water level. When the float reaches a predetermined point, it triggers or deactivates a pump. They are mostly used in cases where precise water level control is critical like water treatment facilities, fuel tanks, and other industrial environments. One of the key advantages of the sensor is that the tether can be easily adjusted, changing the fixed mounting point of the switch. A longer tether results in a wider range of movement, setting a higher or lower trigger point. Shorter tethers result in more precise control. Cable switches are known for being durable, low cost, and suitable in harsh or dirty environments. They, however, are not ideal for small tanks where they do not have enough space to swing around. [40]

3.2.4.2 Ultrasonic Sensor

Ultrasonic sensors are non-contact sensors that measure the distance from the sensor to the water's surface using sound waves. They emit high frequency ultrasonic pulses towards the water which bounce off the surface, return to the sensor. The sensor then measures the time it takes for the wave to return. Using that data the system calculates the distance to the water level and determines how full the tank is. One of the main advantages of ultrasonic sensors is that they provide continuous water level measurements. This makes them ideal for sealed, clean, and hazardous environments, compared to horizontal switches which can degrade and get dirty over time.

These sensors also avoid corrosion, and floating debris on top of never actually touching the water. They are used most commonly in industrial tanks, rainwater collection systems, and various smart water projects. Despite its benefits though, ultrasonic water sensors do have some downsides. They are affected in situations where there is steam, heavy mist, splashing water, or an irregularly sized tank, which all distort the echo signal and overall accuracy. Positioning for the sensor is critical to get the accuracy needed to measure water levels. They should be mounted on top of the tank, and the surface below has to be leveled to maintain stable sensor readings. While they are slightly more expensive than other water level sensors at around $10 - $50, ultrasonic sensors are a good candidate for smart plant watering systems like C.H.A.N.O.S.. [41]

3.2.4.3 Pressure Sensors

Pressure sensors, also known as hydrostatic level sensors, measure the water level of a reservoir by detecting the pressure exerted by the height of the water above the sensor. Typically installed near the bottom of a tank, these sensors convert the pressure readings into an electrical signal that corresponds to the water level. The deeper the water the greater the pressure will be, regardless of how wide or narrow the water volume is. This

linear relationship, the deeper you go the greater the pressure, allows us to extrapolate an accurate reading of the water level.

Pressure sensors are particularly good for sealed tanks, deep reservoirs, and applications where, ultrasonic sensors for example, might have lots of mist, foam, or debris. The sensor offers continuous level measurement, works well in a variety of liquid types, and can be used in irregular shaped tanks. In systems with a known geometry, pressure sensors provide highly accurate readings that are unaffected by surface movement, where other sensors, like float sensors, would struggle. Take note that they must be fixed at the bottom of the tank to get an accurate fix on rising or lowering water levels.

Despite the benefits there are some downsides that we must consider. The first thing to take note of is that the sensor will be placed in the water. This exposes it to corrosion and a buildup of mineral deposits over time. It also must be calibrated based on the height and density of the liquid, which could pose a problem when changes in temperature or water composition occur. Finally they can be more expensive than other sensor types at around $20 on the lowest end to more than $100 for a higher end one. Regardless of this, pressure sensors offer a compact, reliable, and scalable solution for water level monitoring in both small and large scale plant watering systems. [42]

3.2.4.4 Capacitive Sensors

Capacitive water level sensors detect changes in water levels by measuring the changes in capacitance caused by the presence or absence of water near the sensor's probe. These sensors work by generating an electric field around a pair of electrodes and when water comes into proximity, the dielectric constant of the surrounding water changes, altering the capacitance value. This change is measured and used to determine the water level, either discretely or continuously, depending on the sensor's design.

Unlike float switches or pressure sensors, capacitive sensors have no moving parts, which makes them durable and pretty much maintenance free. They can be designed for non-contact applications, where the sensor is placed outside of a tank, making them ideal for clean or sealed environments. They also work in electrically noisy settings, provided proper shielding and calibration are used. Capacitive sensors offer high accuracy, are responsive to small level changes, and can detect a wide range of liquids, including conductive and non-conductive fluids.

However, these sensors are more expensive than others, with prices ranging from $30 - $150 or more. They can also be sensitive to temperature changes, tank wall thickness, and liquid composition, which may require calibration or different circuitry. Despite these downsides, capacitive sensors are an excellent choice for smart, high-precision plant watering systems, especially in applications where space is limited or where physical contact with water should be avoided. [43]

3.2.5 Water Level Sensor Selection

In section 3.2.4 we discussed different types of water sensors. This included the float, ultrasonic, pressure, and capacitive water level sensors. Like all the others before, these

sensors have their pros and cons, and by using that data we will be able to choose the most effective product for our project needs. From the options discussed in 3.2.4 we will reduce the choices of an ultrasonic water level sensor and a horizontal float switch.

The JSN-SR04T is a waterproof ultrasonic distance sensor designed for environments where moisture, mist, or outdoor exposure may damage sensors. It operates by emitting ultrasonic waves and measuring the time it takes for the echo to return, in this case, the water's surface, allowing it to accurately measure distance from the sensor to the water level.

What sets the JSN-SR04T apart from other ultrasonic sensors like the HC-SR04 is its single waterproof transducer, which makes it ideal for projects where the sensor must be mounted near or above a fluid. It has a detection range of 25 cm to 450 cm, with reasonable accuracy, making it a good choice for monitoring water levels in tanks or open containers.

The sensor communicates using standard digital signals which includes a trigger and echo pin, making it compatible not just with Arduino but also with our ESP32 boards. However, since its echo pin outputs a 5V signal, a voltage divider is recommended when connecting with 3.3V microcontrollers like the ESP series. The JSN-SR04T priced at $14 for 2, is a strong candidate for our project as it allows for noncontact measurement, which reduces the risk of contamination, corrosion, or wear over time. [51]

The Ximimark Horizontal Float Switch is a simple, side-mounted water level sensor that uses a magnetic reed switch triggered by a float. It detects whether water has reached a set level, offering a basic HIGH/LOW digital signal for microcontroller input. The switch supports both normally open and normally closed configurations, depending on float orientation. It's compact, low cost (around $9 for 3), and easy to integrate, making it ideal for basic liquid detection in our plant watering system. Its reliability and low power needs make it a practical choice for projects with straightforward on/off water detection needs. [52]

3.2.5.1 Water Level Sensor Selection Conclusion

Table 3.2.5.1.1 Comparison Between Pump Brands.

Brand	Ximimark	JSN-SR04T
Sensor Type	Float	Ultrasonic
Output	On/Off	Trigger/Echo
Waterproof	Yes	Yes
Supply Voltage	N/A	3 - 5.5 volts
Measurement Type	Fixed Point	Continuous

Cost	$9 for 3	$14 for 2

For our project needs we chose the Ximimark Horizontal float switch due to its simplicity in integration, low cost, and function for our project. The JSN-SR04T was not chosen due to its functions being much more than what we needed.

3.2.6 Soil Moisture Sensor Technology

Soil moisture is defined as "the water that is held in the spaces between soil particles" [62]. Various sensors can be used to measure soil moisture by taking advantage of the conductivity property of water compared to the lower conductivity of air and soil particles. The most common types of soil moisture sensors include reflectometer, conductive, and resistive sensors.

Reflectometers are primarily used to measure impedance in electrical circuits to find faults in a cable or verify signal integrity [62]. The ability to measure impedance allows reflectometers to measure soil moisture. In time domain reflectometry (TDR), a rectangular pulse is transmitted down a waveguide of known length, the time for the pulse to return is measured, and the speed of the signal is used to characterize the impedance near the waveguide [64]. The speed of the signal is inversely proportional to the dielectric permittivity; a lower speed indicates a higher dielectric constant which indicates the soil is conductive (has water/is moist) [65]. While time domain reflectometers provide significant accuracy compared to capacitance and resistance sensors, the high cost of off-the-shelf equipment, high complexity, and large number of components/circuit area indicates that TDR is an impractical technology for an embedded smart-home device.

Frequency domain reflectometry calculates the distance to a fault (high impedance) by transmitting sine waves and recording reflected sine waves then measuring the phase difference between the waves [66]. Greater differences in phase values/frequency indicate a larger amount of water/moisture in the soil [67]. Due to a lack of resources/equipment, consistent maintenance and calibration, and high complexity, frequency domain reflectometry was not selected to implement a soil moisture sensor.

Resistive soil moisture sensors also utilize the relatively high conductivity of water to determine the soil moisture content. However, the voltage across a resistor is used to determine the impedance between two probes/pads at the end of the sensor. According to the schematics of a couple of open-source resistive soil moisture sensors, the probes are generally connected from a resistor connected to Vcc and the base of a transistor [68] [69]. The measured voltage is across a resistor connected between the emitter of the transistor and ground [68] [69]. Thus, when there is conductivity within the soil, the transistor turns on and a voltage can be recorded across the resistor. If the conductivity of the soil increases (likely due to increase in water/moisture), then the resistance between the probes decreases, and the voltage across the resistor (at emitter terminal) increases. Resistive sensors require a direct connection between the probes and the soil, which drastically decreases the durability and lifespan of the probes and possibly even product without proper cleaning and maintenance. Furthermore, this direct connectivity also

means that the resistive sensor is highly sensitive to the type of soil, soil composition or topology, and salinity concentration. However, the benefit of a resistive soil moisture sensor is low cost, often being half as expensive as capacitive sensors [70].

Capacitive sensors are different from resistive sensors because they utilize the soil and presence of water as a dielectric instead of a conductor. Capacitive sensors ultimately measure the voltage across a capacitor in order to determine the impedance between the probes. The high dielectric constant of water (~80) increases the capacitance of the capacitor, allowing faster (dis)charging and measurements. Furthermore, since the soil is being used as a dielectric, the probes within the capacitive sensor do not need to be exposed to the soil. This allows protective material to encase or encoat the probes, increasing the durability and lifespan of the product. Additionally, since the soil serves as a dielectric, inconsistencies in the composition of the soil and types of soils are more tolerated than with resistive sensors.

Table 3.2.6.1 summarizes the differences between the four types of soil moisture sensor technology. This table was created using the information described above as well as information collected from the DFRobot blog article "How Soil Moisture Sensors Work in Home Gardening and Agricultural Irrigation" [67]. Ultimately, the capacitive soil moisture sensor was chosen for this project because it is much cheaper and less complex than TDR or FDR sensors but also more accurate and durable than resistive sensors.

Table 3.2.6.1 Soil Moisture Technology Comparison. [67]

	Capacitive	Resistive	TDR	FDR
Cost	$1-$10	$1-$10	$100+	$100+
Accuracy	Moderate	Low	Very High	Very High
Speed	Slow	Slowest	Fastest	Fast
Susceptible to Corrosion	No	Yes	No	No
Calibration	Least	Low	High	High
Design Complexity	Low	Least	High	High
Custom Integration	Moderate	Easy	Difficult	Difficult

3.2.7 Soil Moisture Sensor Product

Due to the printed circuit board (PCB) design requirement for the project, it is preferable that the desired soil moisture sensor is open-source or with available documentation. If the schematic is available, then the sensor used in the prototype design can be easily placed within the PCB. While it is possible to reverse engineer and trace the schematic

and design of other commercial-off-the-shelf sensors using a digital multimeter, this is not as reliable. Furthermore, an open-source design would likely be accompanied with more documentation or technical support.

The first soil moisture sensor with an available schematic is the STEMMA sensor offered by Adafruit. This sensor is sold for $7.50 [100]. The sensor also has an on-board ATSAM microcontroller and a 4-pin I2C interface [100]. However, this sensor was not used because of higher cost and area usage considerations with respect to PCB design. Additionally, other sensors did not require an I2C connection.

A second soil moisture sensor with an available schematic is offered by SparkFun. This sensor featured only three components, but was priced at $6.44 [101]. Due to the high costs and reduced complexity compared to the sensors discussed below, the SparkFun sensor was not selected.

Two other soil moistures with available schematics are offered by SunFounder and DFRobot. These sensors were priced at $6.39 and $5.90, respectively [102-103]. Both sensors had relatively fewer components; neither used an on-board microcontroller and only had 1-2 integrated circuits [102-103]. Furthermore, both sensors used a 3-pin interface with an analog output signal [102-103]. This allows the sensor value to be read with the AnalogRead arduino command. The sensor offered by DFRobot was selected due to slightly lower cost and more technical documentation/support.

Table 3.2.7.1 Capacitive Soil Moisture Sensor Component Comparison. [99-103]

	STEMMA Soil Sensor	SparkFun	SunFounder	DFRobot
Cost	$7.50	$6.44	$6.39	$5.90
Interface	4-pin I2C	3-pin	3-pin	3-pin
Input Power	3-5 VDC	3.3-5 VDC	3.3-5 VDC	3.3-5.5 VDC
Coating of Probes	Metal probe not exposed	Electroless nickel immersion gold	Metal probe not exposed	Metal probe not exposed
Output Range*	200-2000	0-880 (5Vin)	N/A	0-520+
* Output range based primarily on example code and output provided by the manufacturer.				

3.3 PSU

3.3.1 Batteries

The two major classifications of battery technologies are primary and secondary. Primary being non-rechargeable and disposable. They consist of a chemical inside that is consumed upon use and therefore are one time use only. Some examples of this type of battery would be alkaline or aluminum-air batteries. The second type of battery would be rechargeable batteries. Some examples of a secondary battery are Li-ion, Li-po, NI-MH and lead-acid batteries[1]. For the C.H.A.N.O.S project we will need to be relying on rechargeable battery technology due to the design goal of the system being fully powered by solar.

3.3.1.1 Lithium Ion

Lithium-ion (Li-ion) batteries are one of the most widely used types of secondary battery technologies due to their high energy density, lightweight properties, and long cycle life. These batteries operate by moving lithium ions between a graphite anode and a metal oxide cathode during charge and discharge cycles. Li-ion batteries are commonly found in portable electronics, electric vehicles, and energy storage systems because of their ability to deliver significant power in compact form factors. They typically operate within a voltage range of 3.0V to 4.2V per cell and offer energy densities in the range of 150–250 Wh/kg. However, their sensitivity to overcharging and overheating requires the use of protective circuitry to prevent thermal runaway or cell degradation. Due to the system's need for efficient solar charging and 24-hour energy availability, Lithium-ion batteries provide a strong balance of performance, size, and maturity of technology, making them a suitable candidate for solar-based systems. [4]. Our team considered this technology a leading candidate early on due to its wide adoption and proven integration in solar-powered IoT systems, which aligns well with the energy profile of C.H.A.N.O.S.

3.3.1.2 Lithium Polymer

Lithium Polymer (LiPo) batteries are a variation of lithium-ion chemistry that utilize a solid or gel-like polymer electrolyte instead of the traditional liquid electrolyte. This change allows for flexible packaging, reduced weight, and improved safety due to the absence of flammable solvents. LiPo batteries are commonly used in drones, RC systems, wearable technology, and some smartphones because of their ability to be shaped to fit non-standard enclosures. While they generally have slightly lower energy density than cylindrical Li-ion cells, LiPo batteries offer faster discharge rates and more stable structural integrity, especially under stress or damage. Their flat-pack construction and wide surface area also allow for better thermal management in space-constrained designs. This initially made LiPo batteries an attractive option during early prototyping discussions, especially since our project enclosure size may be limited. Despite these advantages, LiPo batteries are more prone to swelling and degradation if mishandled, and require balanced charging to maintain longevity. For applications like C.H.A.N.O.S., LiPo may offer mechanical design flexibility but may pose greater management complexity compared to traditional Li-ion cells [5].

3.3.1.3 Nickel Metal Hydride

Nickel Metal Hydride (NiMH) batteries are a widely adopted rechargeable technology known for their moderate energy density and environmental friendliness. These batteries

use a hydrogen-absorbing alloy as the anode and nickel hydroxide as the cathode. Compared to earlier Nickel-Cadmium (NiCd) batteries, NiMH cells provide higher capacity, reduced memory effect, and fewer toxic materials, making them a more sustainable option. Typical NiMH batteries deliver 1.2V per cell and can achieve 500–1000 charge cycles depending on usage patterns and depth of discharge. They are often found in cordless tools, digital cameras, and some hybrid vehicles. Although they have a lower energy density than lithium-based batteries (60–120 Wh/kg), NiMH batteries are more tolerant of overcharging and less susceptible to thermal events. However, they tend to exhibit higher self-discharge rates, especially in warm environments, which could be a drawback for energy-harvesting systems. NiMH batteries might be considered for their safety and recyclability, but would likely require a larger storage volume and more frequent charging [6]. However, the higher self-discharge rate was not ideal, as maintaining a stable energy reserve overnight is crucial to our design's off-grid performance.

3.3.1.4 Lead Acid

Lead-acid batteries are among the oldest and most proven forms of rechargeable battery technology. Invented in the mid-1800s, they continue to be widely used today in automotive, backup power, and uninterruptible power supply (UPS) systems due to their low cost and high surge current capabilities. The design includes a lead dioxide cathode, sponge lead anode, and a sulfuric acid electrolyte. Lead-acid batteries typically offer 30–50 Wh/kg of energy density and operate at 2.0V per cell. Their robustness and tolerance to overcharging make them reliable in standby or backup systems, but their heavy weight, limited cycle life (200–300 cycles), and poor energy efficiency (~70–80%) pose significant disadvantages for portable or solar-powered applications. Environmental concerns regarding lead toxicity have prompted strict recycling programs and material handling standards. While sealed variants like AGM (Absorbent Glass Mat) and Gel cells offer improved safety and reduced maintenance, their performance still falls short compared to newer chemistries. During our comparison, we acknowledged that while the cost efficiency of lead-acid batteries is compelling, the size and weight constraints would challenge our goal for a compact and efficient housing. In the context of the C.H.A.N.O.S. system, the bulk and limited lifespan of lead-acid batteries make them a less favorable option for solar-reliant energy storage [7].

3.3.1.5 Battery Technology Conclusion

After comparing the major rechargeable battery chemistries, lithium-ion emerged as the most well-rounded option for our application. While LiPo offered appealing mechanical flexibility and NiMH provided safety benefits, neither matched the energy density, efficiency, and system compatibility that Li-ion brings to a solar-powered design. Lead-acid, though cost-effective, was quickly ruled out due to size and cycle life limitations. Given our need for compact storage, consistent energy delivery, and straightforward integration with solar charging hardware, we ultimately determined that Li-ion batteries best aligned with the system requirements of C.H.A.N.O.S. This decision guided our evaluation of specific cell options, which is detailed in the following section.

3.3.1.6 Battery Selection

Selecting a suitable lithium-ion battery was a critical step in optimizing our system for long-term, off-grid operation. The battery had to meet several key criteria, including sufficient capacity to support overnight and low-light runtime, protection circuitry for safety, adequate discharge current to support peak load demands, and physical compatibility with our enclosure and charging circuitry.

We evaluated several commercially available 18650-format Li-ion batteries, as shown in Table 12. These batteries were chosen based on their reputation for reliability, availability from trusted sources, and suitability for solar-powered embedded systems.

Table 3.3.1.6.1 Comparison of Selected 18650 Li-Ion Batteries.

Model	Capacity (mAh)	Max Discharge (A)	Protection Circuit	Voltage Range	Notes
Samsung 30Q [18]	3000	15	No	2.5–4.2 V	High current, unprotected
LG MJ1 [19]	3500	10	No	2.5–4.2 V	High capacity, low cost
Panasonic NCR18650B [20]	3400	4.9	Optional	2.5–4.2 V	Lower discharge rate
Panasonic/Sanyo NCR18650GA [21]	3500	10	Yes (Protected)	2.5–4.2 V	Selected: Best balance of capacity and protection
Shockli Protected 18650 (GA)	3500	10	Yes	2.5–4.2 V	Similar spec, less tested source

Unprotected cells like the Samsung 30Q and LG MJ1 offer high current capabilities but require external battery management systems to ensure safe operation. This adds unnecessary complexity and potential risk for a system designed to run unattended in variable environmental conditions.

The Panasonic/Sanyo NCR18650GA (Protected) cells are the most suitable choice for our application. It delivers a high capacity of 3500 mAh and supports a continuous discharge current of 5A, making it capable of handling both base loads and short inrush events from the pump. The integrated protection circuit ensures safe operation by preventing overcharge, over-discharge, and excessive current draw, key features for lithium-ion safety in solar applications.

Panasonic and Sanyo cells are widely adopted in commercial-grade battery packs and electric mobility applications, offering proven performance and long-term reliability compared to lesser-known or rewrapped brands.

3.3.2 Solar Panels

Solar panels are used to convert energy from the sun into electricity. C.H.A.N.O.S will be using solar technology to have a fully renewable power source and fit with the theme of the plant based project. There are four major types of solar panels commercially available and their names correlate to the type of material and structure of the photovoltaic cells. Those types are the following: monocrystalline silicon, polycrystalline silicon, passivated emitter and rear cell, and thin-film solar cells [2].

Table 3.3.2.1 Solar Panel Technology Comparison.

Feature	**Mono**	**Poly**	**Thin-film**	**PERC**
Efficiency	17%-22%	13%-17%	10%-18%	19%-23%
Cost Per Watt	$3.00-$3.50	$2.80-$3.00	$2.00-$3.00	$3.10-$3.60
Color	Black	Blue	Dark Blue	Black
Temperature coefficient per °C [3]	-.36% to -.40%	-.39% to -.43%	-.20% to -.25%	-.30% to -.35%
Lifespan (years)	25-30	20-25	10-20	25-30
pros	Long life high efficiency	Cost effective	Lightweight and better in high temp	Better low light performance
cons	High cost, poor heat performance	Low efficiency	Short lifespace low efficiency	High cost

3.3.2.1 Monocrystalline Silicon

Monocrystalline silicon solar panels are composed of solar cells made from a single, continuous, silicon crystal. The manufacturing approach results in a highly ordered crystal lattice, allowing electrons to move more freely through the material and reducing recombination events. The production process starts off with growing a silicon ingot from a seed crystal, where molten silicon is slowly pulled and rotated, creating a uniform crystal structure. Once the ingot has been formed it is then sliced into thin wafers, which are further processed into solar cells [27].

The main advantage monocrystalline silicon has over other solar materials is its efficiency, sleek black appearance, and lack of grain boundaries which are found in polycrystalline materials. Without grain boundaries or impurities within the solar cells, electrons encounter few obstacles as they travel to the leads, resulting in quicker and

higher efficiency rates scaling from 18% to 23%. Because of this, fewer panels are needed to generate the same amount of power compared to other materials, making it great for situations where space is limited

Monocrystalline is more expensive, at around $3.00 to $3.50 per watt to make, due to the complexity of the manufacturing process. But due to this they have a longer lifetime and a lower cost per watt in the long run. On top of that, monocrystalline panels exhibit better temperature tolerance compared to other materials. Heat has a negative impact on solar cells because they cause the internal charge carriers to be more excited resulting in more recombination events, reducing overall power. Since monocrystalline has a uniform makeup it is less susceptible to performance degradation from higher temperatures. Overall, even though it has a higher upfront cost, monocrystalline is better for the long run. [28]

3.3.2.2 Polycrystalline Silicon

Polycrystalline Silicon solar cells are characterized by their blue flake-like appearance. This unique look is the result of the manufacturing process, where multiple individual silicon crystals are fused together, unlike monocrystalline panels which are composed of a single continuous crystal structure. This polycrystalline structure is formed through the extraction of silicon dust from sand, which is then combined with other materials, such as coal, and melted into a crucible. After it has been cooled for several hours, it solidifies into a block which is further processed into wafers and finally solar cells. [27]

The orientation of polycrystalline solar cells consists of atoms arranged in different orientations. The borders at which they meet is known as a grain boundary. These grain boundaries introduce structural defects into the material and become the collection point for impurities. Because of this, they cause a reduction in the materials electrical efficiency. When electrons are generated by photons, they want to go to the leads of the solar panel, but sometimes when they try crossing a grain boundary they are slowed down or in the worst case recombined with a hole.

While polycrystalline solar panels exhibit a lower efficiency, 13% to 17%, they remain a popular choice due to their lower production cost of $2.80 - $3.00 per watt. The simpler manufacturing process and reduced material waste make it a more economical option, especially for large scale instances where budget exceeds the need for slightly more efficiency. Some additional downsides though include its lower lifespan and cost in the long run. [28]

3.3.2.3 Passivated Emitter and Rear Cell

Passivated Emitter and Rear Cell, otherwise known as PERC, is an improved variation of both the monocrystalline and polycrystalline solar panels. It was developed to improve energy and efficiency by addressing several limitations to both the technologies. Traditional crystalline silicon solar cells experience what we call solar leakage. This solar leakage can occur as a result of several factors such as electrons recombining before reaching the front contact, imperfections and defects associated with semiconductor materials, and grain boundaries, which are especially found in polycrystalline material.

PERC technology introduces an additional dielectric passivation layer located on the rear side of a solar cell. This layer is a nonconductive material that covers imperfections where the crystal lattice is incomplete, minimizing losses of energy due to recombination. In addition it also serves as a highly reflective layer that redirects unabsorbed photons back into the active region of the solar cell to hopefully produce more electrical energy. PERC cells not only enhance the light absorption ability of the material but also enhance its abilities to perform under varying environmental conditions. This includes performing better in lowlight conditions, higher energy yields at elevated temperatures, and an improved overall performance throughout the day compared to other silicon based technologies. It is priced around $3.10 to $3.60 per watt, making it a slightly higher alternative to other solar panel materials. [29]

3.3.2.4 Thin-Film

Thin-film solar cells are manufactured by depositing very thin layers of P.V. material onto substrates such as glass, plastic or metal. Since the layers in thin film tech are so thin, sometimes only just a few micrometers, significantly less semiconductor material is needed compared to other traditional forms of crystalline silicon solar cells. This lowers the overall manufacturing cost and offers the advantage of a flexible panel design. Unlike monocrystalline and polycrystalline solar panels, thin film uses a different base material for their solar cells.

The most common thin film technologies include Cadmium Telluride (CdTe), Copper Indium Gallium Selenide (CIGS), Amorphous Silicon (a-Si), and Gallium Arsenide (GaAs). Each of which offers a unique advantage in terms of efficiency, manufacturing complexity, cost, and their overall practical applications. Below we go into more detail about all of them.

- CdTe (Cadmium Telluride): CdTe solar cells are made through absorber layers making up a p-n heterojunction. A p-n heterojunction is very similar to a normal p-n junction, the key difference being that it is made from two different semiconductor materials. In this case we have a p-doped Cadmium Telluride with another n-doped material such as Cadmium Sulfide (CdS) or Magnesium Zinc Oxide (MZO). This thin film technology has an efficiency between 15% - 19% and costs around $1.70 - $2.50 per watt. It deals well with high temperatures and performs better in low-light conditions than other materials. Aside from the advantages, there are several disadvantages associated with it. This includes Cadmium's toxicity in relation to environmental factors, Tellurium being a rare element, and a lower efficiency than some silicon based solar panels. [30]

- CIGS (Copper Indium Gallium Selenide): Like CdTe, CIGS is a heterojunction type thin film with a p-doped Copper Indium Gallium Selenide and an n-doped Cadmium Sulfide. The first CIGS thin film was reported by the National Renewable Energy Laboratory (NREL) to have an efficiency of about 13-16% with the most efficient being reported at 23.4% in lab controlled environments. It costs around $2.00 - $300 per watt to make. Besides its relatively good efficiency, it works well in low light environments and high temperatures. Some of the disadvantages include complex manufacturing, scarcity of indium and cadmium, and a higher price than CdTe. [30]

- Amorphous Silicon (a-Si): Unlike the other thin film technologies explained so far, Amorphous Silicon is built using a p-i-n or n-i-p configuration. The manufacturing process is relatively inexpensive, and the thin film performs well in higher temperatures, but has an overall lower efficiency at around 6% - 9%. It costs around $2.00 - $300 per watt to make. Some other advantages include its very light weight and increased flexibility. Some downsides to it though include its lower efficiency and the space needed to produce the same output power compared to other materials. [30]

- GaAs (Gallium Arsenide): GaAs has a more complex manufacturing process than other thin film types. Despite this though it has a very high efficiency rate above 25% with a lab recorded high of 39.2%. In addition, GaAs gives the thin film special properties such as an excellent temperature stability, lightweight on flexible substrates, and a high radiation resistance making it a great fit for satellites and spacecraft. Due to these overwhelming benefits, the cost of GaAs thin film technology is around $70/W, exceedingly more expensive than the other technologies mentioned. [30]

3.3.2.5 Solar Technology Conclusion

After comparing various thin-film solar technologies, it became clear that while options like GaAs and CIGS provide impressive efficiency and performance in extreme environments, their high cost and complex manufacturing processes make them more suitable for aerospace or other specialized applications. Other options like a-Si and CdTe were more affordable and flexible but didn't meet our efficiency needs or compact sizing constraints. Since our system needs to reliably operate off-grid with limited mounting space, thin-film solutions ultimately weren't a good fit. We instead chose to go with a monocrystalline panel due to its higher efficiency, durability, and proven compatibility with MPPT charge controllers like the BQ24650. The selected 18 V panel gives us the voltage headroom needed for efficient charging while holding up well in heat, sunlight, and outdoor conditions—key requirements for our long-term deployment goals.

3.3.2.6 Solar Panel Selection

Selecting the right solar panel was critical to ensuring that C.H.A.N.O.S. could reliably operate off-grid while maintaining charge through a wide range of sunlight conditions. We needed a panel with a high enough voltage to meet the input requirements of our BQ24650 MPPT charge controller, rugged construction for outdoor environments, and a form factor that could integrate cleanly with our enclosure.

We focused on panels with an output voltage in the 17–18 V range to allow for consistent MPPT operation with a 2S lithium-ion battery pack. We also prioritized durability, ease of mounting, and a trusted manufacturer. Below is a summary of the options we considered

Table 3.3.2.6.1 Solar Panel Component Comparison.

Panel Model	**Rated Power (W)**	**Open-Circuit Voltage (Voc)**	**Max Power Voltage (Vmp)**	**Size (mm)**	**Notes**

Voltaic Systems P108	3.5 W	22.5 V	18.0 V	178 × 114 × 3	Compact, rugged ETFE coating, selected panel
Tycon Systems TPS-12-15W	15 W	22.1 V	18.0 V	340 × 230 × 17	Larger capacity, sealed frame
ACOPOWER HY-12P-35W	35 W	21.6 V	17.9 V	470 × 340 × 25	High output, bulkier footprint

After weighing size, output, cost, and reliability, we selected the Voltaic Systems P108 as the final panel for the system. Its maximum power voltage of 18 V is well-matched for the BQ24650 to operate in an efficient MPPT mode, and the 22.5 V open-circuit voltage gives enough overhead for proper tracking and startup in lower light conditions. While its power rating is lower than some of the larger options, the compact footprint made it easier to mount to our enclosure without compromising portability. The ETFE coating provides strong protection against UV exposure, heat, and humidity, which is critical for long-term deployment outdoors. Compared to cheaper plastic-laminated panels, the P108 offers better resilience in harsh conditions and holds up well over time.

The P108's ETFE-coated monocrystalline surface ensures strong protection against UV exposure, heat, and moisture, offering longer-term durability compared to plastic-laminated panels. Additionally, the built-in 2.1 mm barrel connector and standard mounting points streamlined installation and wiring. In the end, the panel offered the best overall balance of electrical performance, environmental durability, and mechanical simplicity.

3.3.3 Voltage Regulators

Overview

Voltage controllers are essential components in power management, particularly in systems where the input voltage may fluctuate due to variable sources like solar panels or batteries. Their primary function is to ensure that critical components such as microcontrollers, sensors, and actuators receive a regulated, consistent voltage supply. Poor voltage regulation can lead to instability, data corruption, or even hardware failure. Therefore, selecting the appropriate voltage controller is vital to system reliability and energy efficiency. Voltage controllers are generally divided into two major categories: linear regulators and switching regulators. Each has distinct tradeoffs in terms of efficiency, complexity, noise, and thermal performance.

3.3.3.1 Linear Regulators

Linear voltage regulators operate by adjusting resistance internally to maintain a fixed output voltage. They are known for their simplicity, low cost, and minimal electromagnetic interference (EMI). A common example is the 7805 series regulator, which outputs 5V from a higher input voltage.

The major drawback of linear regulators is their inefficiency, especially in scenarios where the difference between input and output voltage is large. The excess energy is dissipated as heat, which not only wastes power but also creates thermal management challenges in compact or battery-powered systems. For instance, regulating 7.4V down to 3.3V at 500 mA results in nearly 2 W of power lost as heat, a significant drawback in energy-limited designs [16].

Despite their inefficiency, linear regulators are sometimes used in noise-sensitive analog circuits due to their low ripple output. However, for systems relying on solar and battery power linear regulators are often unsuitable.

3.3.3.2 Switching Regulators

Switching regulators (also known as switch-mode power supplies, or SMPS) use high-frequency switching elements to transfer energy through inductors or capacitors. These regulators can be configured as:

- Buck converters, which step down voltage
- Boost converters, which step up voltage
- Buck-boost converters, which do both depending on input/output conditions

Unlike linear regulators, switching regulators can achieve efficiencies over 85–90%, making them ideal for portable, solar-powered, or embedded systems where every milli watt counts [17]. Their main disadvantages include increased circuit complexity, potential EMI, and the need for external components like inductors and diodes.

Proper design and layout are critical to mitigate switching noise and to ensure stable performance across a range of loads. Additionally, switching regulators must be selected to handle peak current demands without significant voltage drop, particularly important when powering motors or pumps that draw high inrush currents.

3.3.3.3 Regulator Technology Conclusion

For boosting the battery voltage to drive the pump, switching regulation made the most sense. We needed something efficient, capable of handling high current under load, and robust enough to deal with varying input conditions from the battery. A boost converter gave us the flexibility and performance we were looking for without compromising battery life or system stability.

For stepping down voltage to power the ESP32 and sensors, we also went with switching technology. While linear regulators were considered during early prototyping, the efficiency losses weren't justifiable for a system that needs to run reliably off limited solar input. A buck converter offered better performance across the board—especially when it came to maintaining a clean, stable 3.3 V rail with minimal power wasted as heat.

3.3.3.4 Buck Converter Selection

To step down voltage from the 2S battery pack, we needed stable 5 V and 3.3 V rails to power the ESP32, sensors, logic-level hardware, and indicators. These rails are critical for system control and wireless communication, so efficiency, stability, and layout simplicity were all part of the decision process.

During early prototyping, we used the LM2596-5.0, a fixed-output buck converter capable of handling over 2 A. It's widely available in a through-hole package, easy to tune, and forgiving in less-than-ideal breadboard layouts. It gave us a straightforward path to validating the 5 V subsystem without worrying about thermal design or noise sensitivity. For the 3.3 V rail, we paired it with an AMS1117-3.3 linear regulator. The AMS1117 handled the ESP32 and sensor loads well, and although it's inefficient, the simplicity and availability made it a practical choice for bench testing.

For the final design, we replaced both of these parts with a single switching solution: the TPS62162, a compact synchronous buck regulator capable of delivering 1 A continuously at 3.3 V. It features excellent efficiency across the entire battery voltage range (6.4–8.4 V), and its low quiescent current makes it ideal for solar-powered applications. The internal compensation, tight voltage regulation, and strong reference design support from TI helped streamline the PCB layout and minimized our external part count.

Table 3.3.3.4.1 Buck Converter Comparison.

Part Number	**Type**	**Output Voltage**	**Max Current**	**Package**	**Pros**	**Cons**
LM2596-5.0	Buck (Fixed)	5 V	2–3 A	Through-hole	Easy to prototype, tolerant of wide input range	Lower efficiency, large footprint
AMS1117-3.3	Linear Regulator	3.3 V	~800 mA	TO-220	Plug-and-play, reliable for testing	Inefficient, not scalable
MP1584	Buck (Adjustable)	Adjustable	2–3 A	SMD	Compact and capable	SMD-only, noisier, limited support
TPS62162	Buck (Adjustable)	Adjustable	1 A	SMD (WSON)	High efficiency, low quiescent draw, clean output	Requires careful layout, SMD-only

We selected the TPS62162 for the final PCB to handle the 3.3 V rail directly. Its efficiency, compact footprint, and ease of integration made it the best choice for our system's long-term reliability and energy goals. The 5 V rail was removed from the final design to reduce power loss and simplify the power architecture, as all components were verified to operate reliably on 3.3 V.

3.3.3.5 Boost Converter Selection

The 12 V rail in our system is dedicated to powering the submersible pump. While the pump's rated current is only 350 mA at 12 V, we needed a converter that could handle startup conditions and short bursts of higher current especially during dry-start tests or when running near the max rated head. Our battery voltage ranges from 6.4 V to 8.4 V, so the boost converter had to not only generate 12 V reliably but also maintain that voltage without significant sag during transient loads.

For early testing, we tried off-the-shelf boost modules using the MT3608 and XL6009. These were useful for validating the general power architecture, but both fell short under load. The MT3608 struggled to reach 12 V at all once the pump started pulling current, and it would frequently brown out or reset. The XL6009 performed slightly better, but efficiency dropped off quickly and ripple became a concern. Neither option offered the control or protection features we'd want for a long-term solar-powered system.

We ultimately selected the TPS61088, a high-efficiency, synchronous boost converter from TI. It's designed to handle several amps of continuous current and includes cycle-by-cycle current limiting, soft start, and thermal protection. With a switching frequency around 600 kHz, it supports smaller inductors and faster transient response, which helped minimize ripple during pump switching events. TI also provides a full reference layout and verified BOM, which made it easier to model thermal behavior and component sizing during design.

Table 3.3.3.5.1 Boost Converter Comparison.

Part Number	**Type**	**Output Voltage**	**Max Output Current**	**Package**	**Pros**	**Cons**
MT3608	Boost	Adjustable	~1–1.5 A (realistic)	SMD module	Small, cheap, easy to test	Underperforms under load, voltage sag
XL6009	Boost	Adjustable	~2 A	SMD module	Higher current handling, quick to prototype	Noisy output, low efficiency

TPS61088	Boost	Adjustable	2–3 A+ sustained	SMD (VQFN)	High efficiency, tight regulation, solid protection	Requires custom layout, SMD-only

The pump's power draw might seem modest on paper, but accounting for startup transients and ensuring stable system operation meant we needed more headroom than the raw numbers suggested. The TPS61088 gave us the confidence to deliver that 12 V rail cleanly, even under varying load and battery conditions, while staying efficient enough to fit within the energy constraints of a solar-charged system.

3.3.4 Charge Controllers

Charge controllers are an essential component in solar-powered systems, acting as the intermediary between the solar panel array and the battery bank. Their primary function is to regulate the voltage and current coming from the solar panels to ensure safe, efficient battery charging without overcharging, over-discharging, or damaging connected loads. Without a charge controller, fluctuations in solar irradiance or unexpected spikes in panel output could significantly reduce battery lifespan or result in system instability. For systems like C.H.A.N.O.S., where reliable off-grid operation is a design priority, selecting the appropriate charge controller is critical to maintaining power continuity and extending battery health.

Table 3.3.4.1 Charge Controller Technology Comparison.

Feature	**PWM**	**MPPT**	**Shunt/Series Regulator**
Efficiency	70–85%	95–99%	<70%
Panel-Battery Voltage Matching	Required	Not required	Required
Complexity	Low	High	Very Low
Cost	Low	Moderate to High	Very Low
Ideal Use Case	Small fixed-voltage setups	Variable or high-voltage arrays	Obsolete/legacy systems

3.3.4.1 PWM Pulse Width Modulation Controllers

Pulse Width Modulation (PWM) charge controllers are one of the most common and cost-effective solutions for regulating power between a solar panel and a battery. PWM controllers operate by rapidly switching the connection between the solar panel and the

battery on and off, adjusting the duty cycle to control the voltage and current being delivered. This method essentially brings the panel voltage down to match the battery's voltage, resulting in a simple and reliable form of charge regulation.

PWM controllers are best suited for small-scale solar applications where panel and battery voltages are closely matched. They are less efficient than Maximum Power Point Tracking (MPPT) controllers because they do not track the panel's optimal power output. Instead, they tend to lose potential energy, especially in cold or low-irradiance conditions where the panel voltage is significantly higher than the battery voltage. Despite these limitations, their lower cost, durability, and minimal maintenance requirements make them a practical choice for systems with modest energy demands.

For systems that prioritize simplicity and cost control, PWM controllers remain a viable option. However, in our analysis of power reliability and solar harvesting efficiency, we identified that the energy loss associated with PWM regulation would limit our battery's ability to maintain a 24-hour operational cycle, especially during low-sunlight days. Therefore, while PWM technology was reviewed and modeled during our early-stage design comparisons, we ultimately sought a higher-efficiency alternative for our implementation [8].

3.3.4.2 MPPT Maximum Power Point Tracking Controllers

Maximum Power Point Tracking (MPPT) charge controllers are designed to optimize the energy harvest from solar panels by continuously adjusting the electrical operating point of the modules to maximize power output. Unlike PWM controllers, which simply lower the panel voltage to match the battery, MPPT controllers use a DC-DC converter to track and convert excess panel voltage into usable current. This results in significantly improved efficiency, especially in environments with varying temperature and sunlight conditions.

MPPT controllers are particularly advantageous when the solar panel voltage is much higher than the battery voltage, such as in colder climates or when using higher-voltage panels to charge a 12V or 24V battery system. Under these conditions, MPPT units can extract 10–30% more power compared to PWM alternatives, making them ideal for systems where maximizing energy harvest is critical. The trade-offs, however, include increased cost, greater physical size, and more complex circuitry that may require additional thermal management and system calibration.

Our system design includes a compact solar panel with a relatively low wattage, and we expect variable sunlight exposure throughout the day. To make the most of available solar input and ensure consistent battery charging for 24-hour operation, the efficiency gains of an MPPT controller proved critical during our evaluation phase. The additional cost was justified by the improved performance and ability to maintain higher charge levels, especially in partial-shade or low-light scenarios [9].

3.3.4.3 Other Controller Types

Shunt charge controllers are among the earliest and simplest forms of solar charge regulation. These controllers operate by diverting—or "shunting"—excess current away from the battery once it is fully charged. Typically, the excess energy is dissipated through a resistive load, such as a heat sink or dummy resistor bank. While basic in function, shunt regulators are effective in preventing overcharging but do not maximize energy harvest or adapt to changing solar conditions. They are usually found in older or ultra-low-cost systems where efficiency is not a primary concern.

In terms of architecture, shunt controllers use either mechanical relays or solid-state switches to regulate the charge flow. Due to their binary on/off behavior and lack of fine voltage control, they can cause voltage oscillations and are generally considered unsuitable for modern lithium-based battery chemistries, which require more precise charge termination and thermal protection.

Legacy linear regulators and non-MPPT analog controllers also fall into this category. These often lack dynamic tracking capabilities and tend to be optimized for lead-acid batteries. Their lower cost and simplicity once made them common in off-grid applications, but their inefficiency—especially under variable solar conditions—limits their relevance in today's energy-constrained IoT and autonomous systems.

Given the energy efficiency goals and the need to support lithium-ion battery chemistry, shunt and legacy controllers were ruled out early in our selection process. Their inability to regulate efficiently or accommodate battery-specific charge profiles made them incompatible with our system requirements[13][14].

3.3.4.4 Conclusion for Charge controllers

Maximum Power Point Tracking (MPPT) charge controllers are designed to optimize the energy harvest from solar panels by continuously adjusting the electrical operating point of the modules to maximize power output. Unlike PWM controllers, which simply lower the panel voltage to match the battery, MPPT controllers use a DC-DC converter to track and convert excess panel voltage into usable current. This results in significantly improved efficiency, especially in environments with varying temperature and sunlight conditions.

MPPT controllers are particularly advantageous when the solar panel voltage is much higher than the battery voltage, such as in colder climates or when using higher-voltage panels to charge a 12V or 24V battery system. Under these conditions, MPPT units can extract 10–30% more power compared to PWM alternatives, making them ideal for systems where maximizing energy harvest is critical. The trade-offs, however, include increased cost, greater physical size, and more complex circuitry that may require additional thermal management and system calibration.

Our system design includes a compact solar panel with a relatively low wattage, and we expect variable sunlight exposure throughout the day. To make the most of available solar input and ensure consistent battery charging for 24-hour operation, the efficiency gains of

an MPPT controller proved critical during our evaluation phase. The additional cost was justified by the improved performance and ability to maintain higher charge levels, especially in partial-shade or low-light scenarios.

3.3.4.5 Charge controller Selection

While the CN3791 served as our primary charge controller for breadboard testing and early-stage validation, we ultimately selected the BQ24650 for our final system design. Both controllers support 2S lithium-ion charging and include built-in MPPT functionality, but the BQ24650 offers significantly more control and configurability, which became important as the design matured.

The BQ24650 supports precise tuning of charging parameters, including input voltage tracking, current regulation, and battery float voltage. It also offers features like input overvoltage protection, programmable precharge/termination thresholds, and support for thermistor-based temperature monitoring. These capabilities gave us more flexibility to protect the battery and tune performance across different lighting and load scenarios. We also found it better suited for simultaneously charging and powering the system—something that proved critical as we began integrating higher current components like the boost converter and pump.

Compared to the CN3791, the BQ24650 requires more external components and careful PCB layout, but the tradeoff was worth it for the efficiency, reliability, and tuning options it provides. We also evaluated simpler options like the BQ24210, but those were geared more toward single-cell packs or USB-level input voltages, which didn't align with our panel or battery configuration.

Table 3.3.4.5.1 Charge Controller Comparison.

Part Number	**MPPT**	**Battery Chemistry**	**Max Cells**	**Input Voltage**	**Key Features**	**Limitations**
CN3791	Yes	Li-ion	2S	4.5–28 V	Simple integration, few external parts	Limited control, fixed logic thresholds
BQ24650	Yes	Li-ion/LiFePO4/LA	Up to 6S	5–28 V	Highly configurable, robust protection, MPPT	More complex to implement, SMD-only
BQ24210	No	Single-cell Li-ion	1S	4.35–6.5 V	Compact, USB charging, low part count	Not compatible with 2S batteries or solar

We selected the BQ24650 for the final custom PCB because it gave us a more robust, flexible platform for battery charging under variable solar conditions. The additional design effort was justified by the long-term gains in performance, safety, and system resilience.

3.3.5 USB Charge Controllers

In addition to the solar-based charge controller used for the 2S Li-ion pack, the system also required a compact, reliable solution for USB-based charging and power input during debugging, firmware flashing, and indoor testing. For this interface, we selected the MP2672A, a highly integrated single-cell Li-ion battery charger IC designed for USB-powered embedded systems.

Although the MP2672A is not part of the main solar charging path, it serves as an essential subsystem for prototyping, microcontroller bring-up, and powering the ESP32 during software development. Its integrated power-path management, USB compliance, and minimal external component count make it significantly easier to implement compared to solar-oriented MPPT chargers.

The MP2672A provides a complete charging solution from a 5 V USB input, supporting up to 3 A programmable charge current depending on configuration. The device automatically handles pre-conditioning, constant-current (CC), and constant-voltage (CV) phases, along with termination behavior optimized for Li-ion chemistry. This allows safe and predictable charging without requiring the designer to implement external control logic. Built-in system power-path circuitry ensures that the load (in this case the ESP32 and sensors) can be powered even when the battery is depleted, a characteristic that proved especially useful during firmware development and debugging.

Part Number	**Cells**	**Topology**	**Max Charge Current**	**Power-Path**	**Cell Balance**	**Notes**
MP2672A	2S	Boost charger (5 V → 8.4 V)	~2 A	Yes (NVDC load sharing)	Yes	Best choice for 2S USB charging + system load

TP4056	1S	Linear	1 A	No	No	Simple/cheap but 1S only, no load sharing
MCP73831	1S	Linear	500 mA	No	No	Low-power 1S designs; not suitable for 2S

The MP2672A was selected as the USB charge controller because it strikes an ideal balance between features, safety, integration, and ease of PCB implementation. While simpler alternatives such as the TP4056 or MCP73831 offered lower cost, they lacked critical system-level features like power-path management and robust thermal regulation. For development activities, consistent ESP32 power availability and safe, predictable battery charging were higher priorities than minimal cost.

Together with the BQ24650 solar MPPT charger, the MP2672A forms a complete charging architecture that supports both off-grid solar operation and USB-powered development, ensuring flexibility throughout the entire lifecycle of C.H.A.N.O.S.

3.4 Communication

3.4.1 Internet Communication

A web server is required in order to handle communication between the ESP32 microcontroller and product user over any distance. Therefore, the communication protocol between the server and ESP32 or user must have web capabilities, which excludes personal area network (PAN) protocols like IEEE 802.15. The communication protocols with the ability to interface with the web are both wired and wireless: Wi-Fi, 4/5G, DOCSIS (cable/HFC), fiber-optic, and ethernet. Wi-Fi was quickly chosen due to the low complexity and built-in SoC support while providing sufficient functionality. Although wired communication protocols may be more secure, additional algorithms and protocols allow for secure Wi-Fi transactions to occur. Wired communication may also be more consistent, but any disconnection or loss of Wi-Fi should not prevent the system from completing the timer-based watering of the plant. Lastly, wireless communication is much more convenient for the user, as additional equipment will not be required, and the system can be set up anywhere with coverage.

The following wireless communications protocols are natively supported with the ESP32 WROOM 32 microcontroller: Bluetooth, ESP-NOW, MQTT, and IEEE 802.11 (Wi-Fi). Bluetooth, a form of a personal area network, is a protocol for which connection with a

web server is much more complicated compared to Wi-Fi. Similar to HTTP, MQTT is an application layer protocol generally placed on top of the TCP/IP protocols. However, HTTP is better suited for web server interactions. Many libraries, both third-party and Espressif, support implementation of the HTTP protocol. ESP-NOW is an alternative to Wi-Fi, but the former is usually used for internet-of-things (IoT) communication rather than communication with a web server. Wi-Fi, along with HTTP and TCP/IP, are the most common protocols used when performing web server communication.

Table 3.4.1.1 Wireless vs. Wired Communication Comparison.

	Wireless Communication	Wired (cable, fiber-optic, ethernet)
ESP32 WROOM 32 Built-In Support	Yes; IEEE 802.11, Bluetooth, ESP-NOW	No
Secure	Less Secure	More Secure
Consistent	Less Consistent	More Consistent
User Convenience	More Convenient	Less Convenient

Table 3.4.1.2 Wireless Network Standard Comparison.

	Bluetooth	IEEE 802.11 (Wi-Fi)
Network Range / Type	Personal Area Network	Wide Area Network
Power Usage (TX / RX) (mA)	130 / 95	180 / 95

https://www.hivemq.com/blog/mqtt-vs-http-protocols-in-iot-iiot/

Table 3.4.1.2 Application Layer Protocol Comparison.

	HTTP	MQTT	ESP-NOW
Overhead	Higher (with more messages)	Lower (with more messages)	Low
Familiarity / Documentation	Most	Less	Least
Common Use Case	Web Interaction	IoT	IoT

Ultimately, Wi-Fi (IEEE 802.11) was selected primarily due to the built-in support provided by the ESP32 SoC and achievement of the system goals and objectives. Also, Wi-Fi is better suited for web server communication than the other communication protocols supported in the ESP32.

3.4.2 Serial Communication

Only one of the peripheral devices requires a serial communication interface; the light sensor requires an I2C interface with the microcontroller. The remaining peripheral devices use and/or return analog voltage signals with the microcontroller. In addition, the ESP32 module must be programmed after being placed on the PCB. The microcontroller can be programmed via a UART interface. In order to simply and efficiently program the microcontroller, a USB-UART adapter will be used to connect the computer with programming capabilities.

3.4.2.1 USB-UART Conversion

The ESP32 microcontroller only needs to be programmed during development and manufacturing of the system, so a USB-UART adapter cable was considered. However, in order for the project to meet the PCB requirements, a USB-UART conversion circuit was implemented instead. While researching USB-UART conversion integrated circuits, priority was given to components which had better documentation, such as open-source circuit designs. It is also necessary that the IC and circuit are compatible with the ESP32 microcontroller.

The investigated IC's were the MCP2200, CP2102, and FT231X. Each IC has been used in an open-source schematic available online. The MCP2200 was used in a design by sam_moshiri hosted on the instructables website [122]. This design just used the MCP2200 USB-UART subcircuit and a 5V-3.3V subcircuit using a regulator [122]. The CP2102 IC is used within many ESP32 development kits. In an official dev kit schematic released by Espressif, a CP2102 IC is illustrated along with an auto-reset circuit and a 5V-3.3V regulator for USB power supply [135]. Additionally, the CP2102 was recommended by Dr. Weeks and the usage/schematic of the CP2102 according to the manufacturer's data sheet was taught during lecture. Lastly, the FT231X was used in the "SparkFun ESP32 Thing" designed by Jim Lindblom [124]. Although this design is generally an ESP32 development, it includes subcircuits for USB-UART conversion, voltage regulation, and auto-reset of the ESP32 [125]. The auto-reset subcircuit allows for the ESP32 to be automatically reset after installing software to the microcontroller.

Due to the vast amount of resources and in-person aid for the CP2102, including an official Espressif schematic, the CP2102 was selected as the USB-UART Converter IC.

Table 3.4.2.1.1 USB-UART Converter IC Comparison.

	sam_moshiri	Espressif	SparkFun
USB-UART IC	MCP2200	CP2102	FT231X
Compatible with ESP32	Unsure	Yes	Yes
Additional Functions/Subcircuits	Voltage regulator	Voltage regulator; auto-reset	Voltage regulator; Auto-reset

3.5 Software

3.5.1 Web Development Stacks

3.5.1.1 LAMP

The LAMP stack is a popular web development stack due to the fact that it is open source, well supported, and easy to implement. The software technologies typical to a LAMP stack are Linux, Apache, MySQL, and Perl / Python / PHP.

3.5.1.1.1 Frontend

Unlike most other web stacks, LAMP does not specify a frontend software. Typically, the frontend for a LAMP stack is a combination of JavaScript and HTML. This typically results in simpler webpages that utilize static HTML with some JavaScript functionality, which makes for a more straightforward programming experience at the expense of sacrificing on UI and UX. These web pages typically focus on simple, static web pages, rather than pages that make use of more complex, modern features available in JavaScript-based frameworks. Overall, this provides for a simpler, but less extendable frontend framework for developing web UI and UX.

3.5.1.1.2 Backend

Perl, PHP, and Python refer to the possible backend languages that can be used with the LAMP stack. Each of these languages have their own advantages and disadvantages, though for the purposes of the software stack they are all viewed as interchangeable softwares. Perl is a fairly robust scripting language, mainly used with older LAMP stack-based projects. While the language is fairly mature and well supported, it can be difficult to write easily readable code, making it not an ideal choice for more modern project where readability is almost as important as code efficiency. Python is a modern scripting language, heavily used for AI taskloads as well as backend development, due to the large number of libraries that exist for Python which help speed up implementation of new features. However, the language is slow, resulting in a potentially unresponsive backend. PHP is an extremely popular language to use for backend development in LAMP-based projects, mainly due to its flexibility and robustness, as well as its efficiency when working with web applications. Compared to Perl and Python, PHP is far better optimized for web development and is incredibly responsive.

3.5.1.1.3 Database

MySQL refers to the database software used in the LAMP stack. MySQL uses a structured, relational database model that is excellent for storing and working with data. This means that MySQL stores data in a method similar to a table, which allows the database to define relations between rows and columns of data. This makes MySQL very easy to use and interact with when working with a MySQL database, as it is easy to navigate to and select the piece of data that is needed. MySQL is also a very mature and flexible database management system, allowing for good flexibility with what can be stored and a mature ecosystem for integrating with other software components in the stack. Overall, MySQL is a robust and secure database solution that leverages the relational capabilities of a traditional structured database to provide an efficient method for storing and retrieving stored data.

3.5.1.1.4 Platform

Linux refers to the server operating system of the LAMP stack. While there are alternative options for server operating systems, such as FreeBSD or Windows Server, Linux is a mature platform for server grade software with a large support base and a wide range of options for different software stacks, such as the LAMP stack, as well as offering package management tools such as apt for Debian-based Linux systems or dnf for Red Hat-based systems which simplify installing software and dependencies needed to build and run a server. Linux also offers good support for server hardware, and offers a lot of flexibility for different security solutions, depending on the nature of the web application, such as SELinux, which help to mitigate potential security vulnerabilities in the application. Apache refers to the HTTP server software, used for delivering content to a client machine from the remote server. Similar to the Linux operating system, apache is a flexible and mature software that offers good extensibility and support for virtually and web application. Overall, the combination of Linux and PHP work together to create a robust, stable platform for hosting and serving LAMP software.

3.5.1.2 MERN

The MERN stack is a web development stack that emphasizes the use of JavaScript for programming both the frontend and backend services. The main components of the MERN stack include MongoDB, ExpressJS, ReactJS, and NodeJS, where JS refers to JavaScript-based libraries.

3.5.1.2.1 Frontend

ReactJS refers to the front end framework used with the stack. Through React, UI elements can be easily created and configured to create a webpage. React also makes it easy to create interactive elements that update in real time as the user interacts with them, and can utilize other JavaScript libraries as well, such as Bootstrap, a popular JavaScript library that makes it easy to configure web app UI to properly support a variety of screen sizes. React also supports building native applications for mobile devices, allowing for both web applications and native mobile apps to be built simultaneously. Overall, React is a modern and extendable web framework that makes it easy to create complex and reactive UI elements in JavaScript.

3.5.1.2.2 Backend

ExpressJS refers to the back end framework used with the MERN stack. Express serves as a simple, extensible framework for building web APIs and serving web content through JavaScript. Express allows for developers to easily define API functions as well as define how those functions are called by the client through defining the HTTP requests that will service each function, as well as how the client is routed during those requests. Overall, Express serves as a modern, simple way to define an API and define how the client can interact with said API and make requests to send and retrieve data from a server.

3.5.1.2.3 Database

MongoDB serves as the database software for this stack. MongoDB offers flexibility compared to traditional database software solutions in that data is not structured like in a traditional SQL (Structured Query Language) database. This allows for more flexibility in storing different types of data or populating specific fields that may need to be populated on a per-data-point basis. Because of this, MongoDB works well for databases with lots of different attributes, as it is able to reduce the overhead of requiring each data point to have every attribute defined or nulled. Overall, MongoDB serves as a flexible database solution that works well for storing unstructured data.

3.5.1.2.4 Platform

NodeJS refers to the platform that runs the software stack. NodeJS mainly provides a runtime environment for JavaScript programs, allowing for the execution of JavaScript code, as well as a package manager, known as npm (node package manager). Because NodeJS is platform independent, i.e. does not rely on a specific OS, it is incredibly flexible. This makes testing easy, as NodeJS can simply be installed on any machine, and npm can be used to quickly install any necessary dependencies. This means that it is very easy to install everything needed to run the software stack locally and test locally as well.

3.5.1.3 PERN

The PERN stack is an alternative to the MERN stack that utilizes PostgreSQL instead of MongoDB for the database software. PostgreSQL, also known as Postgres, is a SQL-based database solution that offers an advanced featureset, similar to MongoDB, while maintaining a traditional SQL structure for storing data. This makes the PERN stack a great option for web development, as it minimizes some of the issues that exist in a regular MERN stack while maintaining the JavaScript-based stack that makes MERN so useful. Overall, PERN is a flexible, modern web development stack that leverages the strengths of MERN while replacing the database solution with a more traditional, reliable yet modern software that works well with the rest of the stack.

3.5.2 Containerization

3.5.2.1 Docker

Docker is a common containerization program that allows for OS level dependencies and programs to be easily installed and run across a variety of systems and environments. The main benefit of using Docker comes in the form of the Dockerfile, a configuration file that allows for a container's details to be specified and modified with ease. Dockerfiles utilize OS images, which run as containers on a host system, to be installed from a source, typically from an internet repository such as a Nexus repository. The Dockerfile then can be configured to install dependencies necessary for running the application, as well as mounting necessary directories and even basic deployment automation through scripting tools available within the configuration file. Overall, Docker is a robust containerization method that is useful for streamlining development on both local and production machines, and is a great way to ensure that there is no mismatch between dependencies, pathing, or any other environment differences between two machines.

3.5.2.2 Podman

Podman is a direct alternative to docker that serves as a drop-in replacement. Podman can utilize docker configuration files as well as container images, and even provides a similar command line interface to docker. The difference between podman and docker is that podman does not require a server daemon or root-level privileges to operate. Because podman does not utilize a centralized daemon like docker, it is able to run with a reduced containerization overhead, allowing it to be more lightweight and less resource intensive when running on a server with limited resources. Podman also does not require root-level privileges to run, which helps increase security by preventing code running in the container from affecting the system at the root level. This allows us to better secure our web server against attacks that seek to exploit vulnerabilities in libraries or in our own logic that is running in our container, as it prevents these attackers from gaining root-level access to our server. Overall, podman is a great alternative to docker for containerization that combines the flexibility, portability, and ease-of-use of docker with enhanced security features and reduced overhead and resource consumption.

3.5.3 Embedded (ESP32) Programming Language

The C/C++ programming languages were selected due to the support of the ESP-IDF (IoT Development Framework) and Arduino framework. The ESP-IDF contains methods and libraries developed by Espressif, which are capable of performing all of the functions of an ESP32 microcontroller. The Arduino IDE supports native and third-party libraries which are widely used and maintained by developers.

The C/C++ programming languages also offer many benefits over Python that are most important in embedded software development. Python has noticeable overhead due to the fact that many Python libraries and methods are programmed using the C language. This leads to lower performance and higher memory usage when compared to C/C++ whose methods use libraries at a lower-level of abstraction [126]. Although this lower level of abstraction may make development more difficult, a significant amount of ESP32 software is usually already implemented in the Arduino or ESP-IDF frameworks.

Table 3.5.3.1 Embedded Programming Languages for ESP32 Comparison.

	C/C++	Python
Framework	ESP-IDF/Arduino	MicroPython / CircuitPython
Abstraction	Lowest / Low	Highest
Performance / Overhead	Better	Worse
Memory Usage	Lower	Higher

3.5.4 ESP32 Integrated Development Environment (IDE)

The Arduino IDE was selected over VS (Visual Studio) due to the higher-level abstraction of the Arduino IDE methods and libraries. This allows for faster design and

programming. However, VS Code may be necessary if methods within the ESP-IDF (IoT Development Framework) are required at any point. Thus, it is still important to have quick access to VS Code and be able to translate the code between the two IDEs / frameworks. The Espressif IDE was not selected due to our unfamiliarity with the tool.

Table 3.5.4.1 ESP32 Integrated Development Environment Comparison.

	Arduino IDE	Visual Studio Code	Espressif IDE
Familiarity	Moderate	Most	Least
Support for ESP-IDF	No	Yes	Yes
Abstraction	Highest	Low	Lowest

Chapter 4 - Standards and Design Constraints

4.1 Standards

4.1.1 IEEE 802.11

The IEEE 802 set of standards includes the IEEE 802.11 standard, which lays out specifications for certain medium access control (MAC) sublayer and physical layer procedures and protocols that make a type of wireless local area network (WLAN) communication between network-connected devices possible [104]. Introduced in 1997, the standard is periodically revised on an irregular basis [104]. In the datasheets on the ESP32 microcontroller, the ‘b/g/n’ or ‘ax’ revisions are often supported.

The standard utilizes information from various other standards, including Federal Information Processing Standards (FIPS), Internet Engineering Task Force (IETF), International Organization for Standardization (ISO), IEC, and ITU. The standard begins with a general overview of the protocol [104]. Important characteristics of wireless networks are discussed, including distinctions between wireless and wired LANs [104]. Two types of service set architecture are described: basic and extended [104]. Extended service sets require a distribution system to connect STA devices, but only include the STA devices within the set [104]. The two distinct groups of services required by STA devices and DS technology are described [104]. Requirements for a single MAC protocol are described, including the functions, services, MAC frames, and MAC service data units [104].

In addition to the vertical communication between the MAC sublayer and physical layer, specifications for layer management components are also provided [104]. Layer management entities exist in both the MAC and physical layers to communicate with a station management entity [104]. In terms of security, the protocol contains specifications for algorithms used before and after the creation of robust security network associations (RSNA/WPA2), including wired equivalent privacy (WEP) and entity authentication for

pre-RSNA or temporal key integrity protocol (TKIP), CTR with CBC-MAC protocol (CCMP), and key management for RSNA security [104].

Although the MAC sublayer is specified to work with arbitrary implementations of the physical layer, services and primitives of the physical layer are presented in order to facilitate proper MAC communication between peers [104]. Specifications are also provided for various spread spectrum techniques within the physical layer in the 2.4GHz frequency band, such as frequency-hopping spread spectrum (FHSS) and (high rate) direct sequence spread spectrum (DSSS) [104]. In addition to the 2.4GHz band, physical layer specifications are outlined for infrared physical layers and orthogonal frequency division multiplexing (OFDM) at the 5 GHz frequency band [104].

4.1.2 HTTP

The IETF publishes standards for HyperText Transfer Protocol (HTTP) via Request For Comment documents. RFC 9110 and RFC 9112 detail the specifications for HTTP/1.1 architecture/semantics and syntax, respectively [105-106]. This standard was selected due to the compatibility of HTTP/1.1 with ESP32 application programming interfaces (APIs) for Arduino [107].

The RFC 9110 standard describes specifications for the architecture of HTTP, separate from the syntax of and subsequently consistent with any HTTP revision [106]. HTTP is an stateless, application-layer protocol for interoperable communication between other arbitrary, higher-level programs [106]. HTTP specifies request-response message transactions within a client-server network model consisting of user agents, intermediaries, and the origin server [106].

The standard declares specifications regarding naming conventions for uniform resource identifiers (URIs), including specifications for HTTP and HTTP Secure (HTTPS) [106]. The standard specifies what and how field names can be declared and field values (tokens) can be assigned [106]. Fields can be placed in the header and trailer of HTTP messages and are the primary way to transmit metadata [106]. Common rules for specific field name-value pairs are included [106]. The standard describes an abstract composition for HTTP messages compatible with all major versions of HTTP, including the framing structure, control data, header, content stream or message body, and trailer [106]. Messages are specified such that the server can fully decode any message without considering prior message history (stateless) [106]. Specifications for the routing of HTTP request messages are made, related to the field pairs defined by the client and actions performed by intermediaries and the server [106]. The routing of response messages is generally the reverse of the request sequence and has briefer, looser specifications [106].

Further specifications are given for the fields related to message content and metadata. The request methods are defined and classified as ‘safe’ and/or ‘idempotent’ [106]. The document specifies proper usage of the methods by the clients and actions performed by the server on the target resource (URI) identified in the request [106]. Requirements for particular header fields used to introduce context in request messages are listed for both

the client and server [106]. Although the HTTP protocol does not specify a rigid format for content data to be transferred, content negotiation is required to enable communication between the client and server [106]. The standard specifies three types of patterns used for content negotiation: ‘proactive’, ‘reactive’, and ‘request content’ [106].

Additional, optional types of request messages are specified [106]. Conditional requests allow for a transfer to occur if certain prerequisite conditions are met [106]. Range requests allow for a transfer of a subset of data, which is necessary when connections are broken up or the client/server has insufficient memory [106]. In terms of security, the standard establishes a framework for user authentication when sending requests to the server via certain header fields and response messages [106]. The standard includes more security specifications in section 17 [106]. Most of these specifications improve security on the client- or server-side interface [106]. Instead, in order to improve the security during data transfer and between intermediaries, the HTTPS protocol was established which applies Transport Layer Security (TLS) cryptographic algorithms on the transmitted data [106]. Standards for the TLS protocol are described in the RFC 8446 standard released by the IETF.

The RFC 9112 standard includes specifications regarding the syntax of HTTP/1.1 communication that must, should, or can be followed by message senders and receivers (parsers) [105]. The standard describes a syntactic structure that all HTTP messages should follow, including a start line, header/field, and message body [105]. The structure and parsing of fields are discussed later and include the name and value of the field [105]. The standard describes a structure for requests, including the method and four types of request targets [105]. HTTP responses should contain a status line within the message; the status line consists of a three digit status code (e.g. 200) and noncompulsory description of the status (e.g. OK) [105]. Message bodies are allowed for request messages and some response messages [105]. Two important fields related to messages/message bodies are discussed: transfer-encoding and content-length [105]. The transfer-encoding field can be used to delimit the message structure and also format messages into chunks for streams of irregular size [105]. The content-length field may be used if the transfer-encoding field is not present [105]. This field can also be used to indicate the size of the message content and demarcate the framing boundaries of the message [105]. The management of connections between HTTP and TCP protocols is specified, especially with regard to persistence, pipelining, concurrency, and closing of connections [105]. The creation and closure of TLS connections in order to support HTTPS communication is briefly outlined [105].

The HTTP Standard defines many commonly used message types that can be specified by the client. Among these many methods, five different types are viewed as the most common methods used by HTTP messages. These methods are designed in order to handle data transmission in a simple way, focusing on one way communication, where either data is sent from client to server, or data is received by the client from the server. The methods are as follows:

1. GET: The GET method is a common request that asks the server to retrieve a piece of data that it has and send that data to the client in response. This method allows the client machine to retrieve a remote resource, such as a web browser retrieving a page to display.
2. POST: The POST method is a request that sends data from the client to the server. This method is useful as a way to upload data from a local machine to a remote machine, for example uploading a video stored on a local computer to a remote video hosting platform.
3. PUT: The PUT method is another request that sends data from the client to the server. Where the PUT method differs from the POST method is in the manner in which the server handles the received data. The PUT method specifies that the new incoming data must replace existing data.
4. DELETE: The DELETE method is used to delete a resource from the server. This can be used in cases such as a user deleting their account.
5. PATCH: The PATCH method is another request that sends data from the client to the server. Where POST does not replace old data, and PUT entirely replaces old data, PATCH allows the client machine to specify that only specific parts of a piece of existing data should be overwritten.

The HTTP Protocol also defines status codes, which are integer values returned by a server in its response to the client's request. These codes take the form of a 3 digit number, with the value of the hundreds place signifying the request status. These codes are especially useful for indicating to a client machine what the status of their request is, and have a multitude of uses, such as testing to ensure that the message was successfully received in a development environment, or informing an end user that the server they are trying to access is down or that the link that they are trying to access is invalid.

1. 1xx: Response status codes beginning with a value of 1 indicate that the server wishes to provide further information to the client. An example is a status code of 101, which indicates that a protocol switch has been requested by the client and agreed to by the server.
2. 2xx: Response status codes beginning with a value of 2 indicate a successful request. The most common 2xx status code is 200, which signals to the client that the request was successfully processed by the server.
3. 3xx: Response status codes beginning with a value of 3 indicate that redirection is necessary. This means that the content that is being requested may be used or otherwise unavailable at the URL that is being accessed. An example is status code 301, which signifies that the content at the given URL has been moved permanently, and further requests for this content should be directed to another, given URL.
4. 4xx: Response status codes beginning with a value of 4 indicate a clientside error with the request. This could be a result of a number of errors with the client's request, such as a syntax problem or an error with the format of the request. One of the most well-known HTTP error codes falls under the 4xx series of codes, that being the 404 code, which signifies that the requested resource was not able to be found.
5. 5xx: Response status codes beginning with a value of 5 indicate a server-side error with the request. These codes encompass a large amount of errors, similar to the 4xx

codes, but for errors relating to the server, such as a 503 error which defines an error where the server is either too overloaded to handle the request or has gone down.

Overall, the HTTP Standard provides a robust and convenient protocol for applications running within a client system to communicate with a server. Because of its wide adoption, it has almost singlehandedly defined the modern networking communication approach between a client machine and a server machine, and serves as a simple but effective method for requesting and transferring data between these two machines.

4.1.3 PCB

Many standards exist for the design of printed circuit boards (PCBs). The International Electrotechnical Commission (IEC) is one organization that has developed a few standards for PCB design, assembly, and coating. IEC 61191 is a standard on "printed board assemblies" [108]. IEC 61086 is a standard on "Coatings for loaded printed wire boards (conformal coating)" [109]. The Global Electronics Association, formerly known as the IPC (or Institute of Printed Circuits), is arguably the most prominent commercial standards body for PCB design, testing, and manufacturing. These standards include IPC-2221 for "Generic Standard on Printed Board Design" and IPC-A-600 for "Acceptability of Printed Boards" [110].

The cost of these standards are quite prohibitive with respect to the budget of this product, described in Chapter 10 of this document, as pricing for the IEC TR 61191-8:2021 is approximately $312 USD [111]. Similar to the IEC, the IPC sells their standards at a similar price.

Alternatively, many non-accredited resources and articles are available online that detail PCB design practices. Although these resources may not be as highly accredited as IPC standards or may feature guidelines rather than required specifications, they will suffice for the goals and scope of this project and better fit within the budget. If this product were to be commercially released and sold, then standards documentation would likely be purchased to ensure that this design is in accordance with common practices.

4.1.4 ESP Hardware Guidelines

Espressif, the manufacturer of the ESP32 module, has released a hardware design guidelines document for a set of ESP32 modules which includes the ESP32-WROOM-32 [112]. Part of this document describes the electrical components that are external to the chip, but integrated into the ESP32 design. Although some of these components, such as the RF antenna, are already integrated in the ESP32-WROOM-32 module, other external components are not already integrated. For example, the ESP32 module accommodates a 32.768 kHz crystal clock with dedicated pins. The guidelines document details the proper connections that need to be made with the clock module. Furthermore, not all pins on the ESP32 chip will be utilized and subsequently require specific connections. The document also makes recommendations concerning pins, such as the addition of optional capacitors or resistors, in order to improve the stability of the module. Most importantly, however, this document also details the PCB layout of the ESP32 module. As the central part of the PCB, these recommendations must be considered when laying out the power supply, the

layers of board, electrode types and sizes, the direction of the RF antenna, and physical characteristics of the board and traces. The document also contains suggestions for troubleshooting issues regarding performance and layout.

4.1.5 TCP

TCP, which stands for Transmission Control Protocol, is a reliable communication standard protocol used in the transport layer of the OSI Model. Whereas the application layer defines how applications can send and receive data, the transport layer deals with how data is sent between machines. The goal of protocols that fall under this layer is to define lower level communication details than can be defined by higher level standards, such as how data is broken down into packets, how these packets are transmitted, etc. TCP is one of the two main protocols used at this layer, and is known for ensuring reliable in-order communication, at the expense of speed. Because of this focus on reliability, TCP is the main protocol used transporting HTTP data packets between client and server machines.

TCP works through sending packets between two machines. These packets represent small portions of data. Transmission of these packets using the protocol is accomplished through a 3-stage process. The first stage of this process is the Connection phase, where a process known as "handshaking" occurs. Traditional handshaking involves a 3-way handshaking process as follows:

1. SYN: The initial packet, known as a SYN packet, is sent from the client to the server. This packet will contain a specific, randomly-generated number, which will be the starting sequence number for the communication. We will call this number X. The SYN packet serves as a request to synchronize the client and the server for data transmission.
2. SYN-ACK: This packet is sent by the server in response to the SYN packet sent by the client. This also contains a number, which will be equivalent to X+1. This number is known as the acknowledgement number. By sending this packet, the server is acknowledging and agreeing to the request sent by the client.
3. ACK: This packet is sent by the client in response to the server's SYN-ACK packet. It serves as an acknowledgement from the client that it has received the server's response. The sequence number for this packet should be X+2. After this packet, the connection process has completed and data transmission can begin.

For data transmission, the TCP protocol ensures that data must be transmitted in-order, with retransmission when packets are lost, dropped, or corrupted during transmission. It ensures this through the use of sequence numbers. The receiver of the data can process these packets and ensure that all data has been received by ensuring that there are no missing or misplaced sequence numbers. Once a missing number is detected, it will stop processing packets. As packets are processed by the receiver, it will send ACK packets with a corresponding sequence number to the processed packet. When processing stops, it will retransmit the ACK with the sequence number of the last packet that it processed. The sender, upon receiving the retransmitted ACK, can understand that a packet was lost and retransmit the data, allowing for the transmission to continue. Because of this strict guideline, TCP transmission is generally slower than other available protocols, however,

it is more reliable and ensures that all data is correctly transmitted, making it useful for a service like a web application where speed is not as important as correctness of information.

For integrating TCP into a web application, the bulk of the work is already done by existing HTTP libraries, though the application developer must still specify the port that will be used by the web server. Once this port is specified, the application can listen on this port for incoming transmissions and receive those transmissions through the TCP protocol.

The termination process for a TCP transmission, there are multiple different methods that are used depending on how the connection is closed, but also depending on many other methods such as the platform of the device that is closing the connection. Typically, however, connections should be closed using a multi-way handshake method, similar to how connections are opened. A 3-way handshake implementation of closing a TCP connection typically involves the sending machine sending a FIN packet to the receiver. Upon processing this packet, the receiver sends a FIN-ACK packet, which the sender should respond to with a final ACK packet. This 3-way handshake method is implemented in a similar manner to the 3-way handshake that is traditionally used to open a connection. However, this is not always how connections are closed, as there are many methods that can be used to close a connection, such as a sudden drop by one of the machines, which will leave the connection in an unfinished state where some data may not have finished transferring.

Overall, TCP is a reliable and efficient data transmission method that is well suited to our use case, which will mainly be sending and receiving HTTP requests. Because of its robust transmission method, we can ensure that data is efficiently and completely transferred between devices.

4.1.6 Wiring Standards

Many of the peripherals of the system will be connected to the main PCB by wiring. According to the requirements of the Senior Design rubric, these wires will need to be connected to the PCB or sensor probes through soldering. For example, the soil moisture sensor will consist of electrical components populated on the PCB and capacitive probes placed in the soil. The wires used in this design will be within contact of sunlight throughout much of the day. The sunlight may also increase the amount of heat consumed by the wire. They will also be adjacent to water and sources of humidity. Due to the use of Wi-Fi, electromagnetic interference may also need to be considered. Most importantly, the wires will need to have the proper voltage and current values.

Similar to the PCB standards, IPC also releases standards for wiring. Many of these standards appear in the prominent IPC-A-610 standards document “Acceptability of Electronic Assemblies”. This document discusses common practices of installing, routing, and soldering of wires. The IPC and Wiring Harness Manufacturers’ Association (WHMA) also established IPC/WHMA-A-620 or “Requirements and Acceptance for Cable and Wire Harness Assemblies”. This document describes standards for the

cleaning, soldering, crimping, termination, splicing, and routing of wires. However, similar to the PCB standards, both documents cost almost $300 dollars, which is out of the budget range.

4.2 Constraints

4.2.1 Economic Constraints

The components and subsystems of the product will generally not economically constrain the design and implementation. The microcontrollers do not economically constrain the project, as the ESP32 family of modules and SoCs are some of the most cost-efficient microcontrollers commercially available, costing often only several dollars. The cost of sensors is slightly more expensive than ESP32 modules. For example, the soil moisture sensors cost about $6 per unit. When designing the PCB, these components will require resistors, capacitors, and integrated circuits to be populated and soldered onto the board. The lithium ion battery cells cost about $5-$7 per unit, and at least two batteries will be required. Extra copies of all materials should be purchased due to any discrepancies or issues that may arise while testing and before demonstrating the system. Boards for the CN3791 charge controller can be purchased for as low of a price as $4-$5, but some sellers list the component at $20. The most economically constraining electro-optical component in the system is the P108 Voltaic Systems solar panel. This panel is sold by online retailers for about $45-$50 dollars. Due to this price, only one or two solar panels will likely be purchased.

The software components and subsystems of the product will cost very little. Free and open-source software (FOSS) will be utilized at every opportunity. There are a plethora of FOSS available for the ESP32. Most of the FOSS used for this product will be libraries for the ESP32, both from the manufacturer (Espressif) and reliable third-parties. These libraries will probably handle lower-level protocols, such as TCP/IP and HTTP/S, and cryptographic/security algorithms. An integrated development environment is also required to design software for the ESP32 module. Some of the most notable IDEs for embedded development are the Arduino Software IDE or Visual Studio (VS) Code, both of which are free to use. Both require extensions or repositories to be linked to the IDE in order to use libraries and commands compatible with the ESP32 hardware. Espressif provides these extensions and repositories free of charge. The only economic constraint for the software is likely a cloud server used for access from different networks, which would likely cost several dollars per month. However, if the user remains on the same Wi-Fi network as the ESP32, it would not cost anything to transmit HTML web pages and data between the ESP32 and the user's network-connected device.

Finally, there may be some economic constraints when it comes to the physical material of the product, largely if the pot is manufactured by the design team, instead of relying on the client to utilize their own pot. For example, if a pot is to be 3D-printed, then filament costs will need to be considered, but the cost of a 3D printer will likely not be considered as one can be provided by parties associated with the designers.

4.2.2 Environmental Constraints

In an article by Current Results, based on a report by the National Climatic Data Center, it was estimated that the major cities in Florida experienced days with 3.6 - 8.4 hours of sunlight for approximately two thirds of the year [113]. In an article by Unbound Solar, a vast majority of the contiguous United States receive, on average, 4.5-6 hours of sunlight with an average of 1000 W/(m^2) [114]. Given that the indoor plant is likely located by only one or two windows, the amount of direct sunlight received by the system is approximately half of the reported findings. A safe minimum average hours of sunlight would be about 1.8 - 2.25 hours. Furthermore, the energy collected using the solar panel must be stored such that it will be able to, at maximum capacity, power the system for at least two days, assuming correct/regular functioning of the other components.

Other characteristics of the weather do not appear to constrain the design of the system. For example, it would be reasonable to assume that the room's temperature and humidity are regulated by the owner to levels healthy for the plant. The interior setting of the plant and system make this feasible, since temperature and humidity can be controlled by a thermostat. However, it should be noted to the consumer that such factors will not be regulated or measured by the system.

There should be effectively no adverse effects of the product on the environment, largely due to the low power consumption of the components and renewable power source of solar power.

4.2.3 Sustainability Constraints

One constraint related to sustainability is the limited amount of water that can be supplied to the plant. Although the water used to irrigate the plant can be reused by draining the water back into the supply tank, the water must be filtered properly in order to avoid recycling bacteria into the soil. Despite the reuse of drainage water, the duration that the system can continuously irrigate the soil will still be primarily limited by the size of the water supply tank. The U.S. Travel Association found that, in 2017, the average employee travelled for only 8 of 17 total vacation days [115]. Hence, the water supply tank should be large enough to irrigate the plant for at least 14 days.

Another related constraint is the maintenance and calibration of the sensors. The percentage of moisture in the soil is calculated relative to measurements by the sensor in completely dry (contacting only air) and completely wet (submerged in water) conditions. Over time, the sensors may need to be recalibrated due to changes in the components of the system, such as integrated circuits (IC) or capacitors. Exposure to humidity may also accelerate the degradation of components and increase frequency of recalibration. Additionally, the use of a new type of soil or plant would also require recalibration of the soil moisture sensor. Consequently, the sensors requiring recalibration must be accessible to the user.

4.2.4 Health and Safety Constraints

Lithium ion batteries may pose a significant safety and fire hazard if improperly designed, integrated, or utilized. The batteries must operate within a certain temperature

range and avoid heat. A charge controller shall be used to prevent the lithium ion battery from supplying or receiving too much voltage or at too fast of a rate. Lithium ion batteries should not be placed within direct contact of sunlight or near flammable materials. Lithium ion batteries are also sensitive to physical damage, such as vibration or punctures. Lithium ion batteries may also pose a safety hazard when in contact with water, which is one of the primary concerns with the C.H.A.N.O.S. design. Danger caused by these hazards can be avoided by following the specifications listed in the supplier's datasheet. This includes using proper insulation and packaging of the batteries within the system, particularly to prevent humidity from the pot or watering system. Lithium ion batteries must also be properly disposed of through certain procedures.

In comparison, the materials and components within monocrystalline solar panels or cells are generally safe and do not pose notable safety hazards. The P108 Voltaic Systems solar panels will have a nominal voltage of 17.34V and power of ~10W [116]. These values do not raise safety or fire concerns, although precaution must be taken to avoid shorting the solar cell component.

In addition to fire hazards, the temperature of the system also constrains the design. Solar panels reflect sunlight and thus radiate heat. Lithium ion batteries, even without approaching temperatures that cause fires, will increase in temperature during usage. Although all electrical components transfer electrical energy to heat, processors, such as those on the ESP32, can generate a notably large amount of heat. However, the use of low power modes and lack of software complexity should help reduce the heat generated within the system. Still, the temperature of the internal electrical components constrain the material that will be used to design and manufacture the pot, as the filament should be moderately heat resistant, such that contact with the client's skin will not inflict pain or injury.

Although large voltages and currents are not generally used within the rest of the design, proper insulation of wires and conductors must be implemented in order to prevent the user from receiving electrical shock when handling the system. There should be a method of powering down the system so that the user can perform actions such as replacing the soil/plant or refilling the water supply tank. There should also be an ergonomic and safe way for the user to handle sensors during calibration and setup.

Safety constraints concerning wireless transmission must also be considered. While many concerns surrounding the safety of human exposure to Wi-Fi networks have grown in contemporary history, the World Health Organization (WHO) stated, in 2006, that exposure to electromagnetic waves does not affect health if exposure is in accordance with the limits set by the International Commission on Non-Ionizing Radiation Protection (ICNIRP) [117]. In a 2008 statement by the ICNIRP, the limit on radiated power of an EMF signal was reportedly set at 30 dBm by the Federal Communications Commission (FCC) for the 2.4GHz band [118]. According to Espressif documentation on the ESP32 D0WD SoC, the maximum transmission power is 20.5 dBm [73]. Furthermore, the use of the ESP32 in low-power modes reduces the period of wireless transmission. In

conclusion, the ESP32 antenna and C.H.A.N.O.S. software is designed in accordance with established safety standards.

4.2.5 Security Constraints

The use of Wi-Fi also introduces security concerns. Malicious actors may try to breach wireless communication transactions via packet sniffing. Sensitive data, such as the SSID and password for a private Wi-Fi network, will be transmitted during the setup of the system. Thus, measures must be taken to ensure that the transfer of data between the ESP32 module and the web server/application are protected.

In addition to the TCP/IP protocol stack implemented in ESP32 modules, Espressif also offers an application protocol interface (API) for HTTPS [119]. Furthermore, the ESP32 module has dedicated hardware for execution of cryptographic algorithms such as AES, SHA-2, RSA, ECC, and random number generation (RNG) [73]. Many open-source libraries are available which provide software that implements these algorithms, such as Mbed TLS. Furthermore, the ESP-IDF programming guide provides references for implementing the HTTPS protocol, which increases the security of HTTP communication using TLS algorithms.

Finally, specific protection schemes, such as password and username authentication will be implemented in the C.H.A.N.O.S. software design. When the ESP32 module acts as an access point, a unique SSID and password will be generated by the designers/manufacturers, who will provide the user with this information via secure methods. Another security checkpoint can be established by collecting a username and password given by a user when creating their account in the web application, before the product is ordered/delivered. This username and password can be hardcoded within the ESP32 non-volatile (flash) memory and required to be input by the user before sending their Wi-Fi network information. ESP32 modules also offer flash encryption.

4.2.6 Time Constraint

Time management is critical when working on a group project in Senior Design. Setting up a clear and collaborative system early on helps keep everyone aligned. For example, using shared tools such as Google Docs allows team members to contribute their parts, offer feedback, and track progress collectively. Maintaining a group chat also helps ensure everyone stays updated with class announcements and changes. Additionally, setting a recurring weekly meeting time that works for all members can greatly improve coordination and task delegation.

As mentioned in our lectures, consistently working on the SD1 paper throughout the semester is key to avoiding procrastination and the stress of last minute work. There were examples in class of a group that had to stay up all night in the UCF lab to finish their project, which left them exhausted and at risk of failing. Poor time management can delay project completion, jeopardize your final grade, and potentially result in retaking the course, which would delay graduation, lower your GPA, and lead to additional costs.
External factors such as Florida's hurricane season, for example, can cause power outages and internet disruptions. Therefore, it's important to stay ahead of deadlines and build in

buffer time for unexpected events. The SD1 course is structured to help manage this workload with staggered deliverables, including the 10-page, 60-page, and 120-page submissions. Regular check-ins with your assigned professor also ensure you're meeting milestones and staying on track for a passing grade.

It's also wise to source project components early to avoid delays. Platforms like Amazon offer affordable options, but the shipping times can be long. Alternatively, making items yourself can be a quicker option such as the use of a 3d printer to make our project casing. Looking ahead to SD2, we will have more time to work on our project compared to what we did in the summer semester. Even though we may have more time it is critical to use it wisely. With this in mind, our group intentionally selected a project idea that is realistic to complete within the next 4 month timeframe. In summary, respecting time as a valuable resource can prevent many potential setbacks. Planning carefully, working consistently, and preparing for delays are the keys to a successful Senior Design experience.

Chapter 5 - Applications of ChatGPT and other Similar Platforms

Transformer (GPT) architecture. These models are designed to understand and generate human-like responses in natural language. Their primary purpose is to assist with tasks such as answering questions, summarizing documents, drafting code, writing content, and more.
LLMs operate through three core phases:

1. Pretraining, where the model learns language patterns from massive text datasets;

2. Fine-tuning, where it is further trained on more specific or curated content;

3. Reinforcement Learning with Human Feedback (RLHF), where human reviewers help guide and optimize model responses.

Despite their usefulness, LLMs have notable limitations. They don't "understand" in a human sense — they rely on predictive pattern recognition rather than true comprehension. They can also reflect biases from their training data, sometimes generate incorrect or fabricated information (a phenomenon known as *hallucination*), and typically lack real-time memory or awareness of past interactions unless integrated with specialized memory systems [15]

Scenario 1 - "Write Arduino-compatible code for an ESP32 that reads soil moisture data every 15 minutes and sends it via HTTP to a custom web dashboard."

For this test, I asked two large language models, GPT-4o (OpenAI) and Gemini 2.5 Pro (Google), to write Arduino-compatible code for an ESP32 that reads soil moisture every 15 minutes and sends it via HTTP to a custom web dashboard. This prompt ties directly

into one of the core functions of our senior design project, C.H.A.N.O.S., which automates irrigation using sensor data, a web interface, and solar-powered hardware.

Both models gave solid code that could definitely be used as a base. GPT-4o kept things simple. It handled analog input, used millis() for timing, and sent a basic HTTP POST with the data. It's a great starting point, readable, easy to follow, and good for fast prototyping. But it missed a few things we'd need for actual deployment, like power-saving features and a cleaner way to build the JSON. It also didn't include any setup info, so you'd need to figure out pins, dashboard paths, and payload formatting on your own.

What really stood out to me in Gemini's response was how complete it felt. It used deep sleep to save power between readings, built the code using arduino json, and broke the code into clean, reusable chunks. It also had solid comments throughout and included setup instructions explaining the pins, server config, and how to adjust the endpoint. That kind of guidance saves you a ton of time during testing and integration, especially when you're bouncing between hardware and code.

One of the biggest benefits of using LLMs in a project like this is how easy it is to iterate. I could ask Gemini or GPT, "Add some debugging output for Wi-Fi issues," or "Retry the HTTP post if it fails," and it would update the code instantly. That kind of back-and-forth saves hours, especially when you're trying to troubleshoot or tweak something quickly.
That said, you can't just copy and paste and call it done. LLMs don't always match your exact setup, and sometimes they guess wrong on things like pins or assume you're using default libraries. You still have to review everything and test it yourself.

In this case, Gemini's answer was just more aligned with what we actually need for CHANOS. GPT-4o helped get the ball rolling, but Gemini gave us something that's actually close to ready for deployment. Since we're running off solar, care about long-term reliability, and need clean integration, Gemini's extra depth made a real difference.

Table 5.1. Language Learning Model (LLM) Comparison.

Feature / Focus	**GPT-4o**	**Gemini 1.5 Pro**
Basic Functionality	Moisture reading, timing, and HTTP POST handled cleanly	Same core features, handled correctly
Power Management	Not included	Uses deep sleep for low-power operation
JSON Handling	Manual string formatting	Uses json for clean, structured payload
Code Structure	Flat script, no functions	Modular functions with clear purpose

Comments & Readability	Simple and clean, not overly detailed	Well-commented with context for each step
Setup Instructions	None	Includes notes on GPIO pins, server paths, and payload
Error Handling / Debugging	Basic output only	Includes Wi-Fi retry logic and HTTP response handling
Deployment Readiness	Good for prototyping	Closer to plug-and-play for real deployment
Flexibility for Iteration	Responds well to follow-ups	Responds well and adds helpful suggestions
Best Fit For	Quick testing and getting started	Deployment on a solar-powered field unit like CHANOS

Scenario 2 - "Give me a block diagram and explanation of how the sensors, ESP32, solar power, battery, and web dashboard work together in a smart irrigation system."

When comparing ChatGPT and Gemini's responses to the smart irrigation system prompt, both gave helpful answers but took different approaches. ChatGPT focused more on what we're actually building. It laid out the components we're using: moisture sensor, ESP32, solar panel, battery, and web dashboard and explained how they connect and work together. It was clear, well-structured, and didn't waste time on features we're not including. I could take most of what it wrote and drop it straight into our documentation or project presentation without major changes.

Gemini, on the other hand, gave a longer, more in-depth response. It included extra sensors like rain detection and temperature/humidity monitoring, and went deeper into topics like MQTT communication, battery chemistry, and power management modes. In some ways, that's useful. It gave me ideas for features we might add down the line or questions we should think through. But for our current system, it drifted too far outside scope. I'd need to cut back a lot of what was there to make it fit what CHANOS is actually doing.

The biggest difference is that ChatGPT gave a tighter, more usable response right away, while Gemini leaned toward a more ideal version of a smart irrigation system, whether or not all of it was relevant. That extra detail could be helpful if we were building a commercial product or writing a technical deep dive, but it's overkill for what we're doing now.

Using language models like these for the CHANOS project has been a net positive. They help move things forward faster. If I need to explain how a moisture sensor ties into ESP32 logic or how we're powering the system, I can get a full draft in minutes instead of spending half a day putting it together. It also helps with cross-functional communication. If one of us is building the hardware and another person is working on the web dashboard, this gives everyone a shared understanding of how things work together without needing to go back and forth too much.

There have also been times where the output pointed me toward something I hadn't considered yet, like adding logic for power-saving or double-checking the way we handle sensor thresholds. So even when the response isn't perfect, it pushes the thinking forward.

That said, it's not a magic solution. These tools don't know what our real-world limitations are. They don't know our timeline, our part list, or the level of complexity we can actually support in six months. Sometimes the suggestions they make aren't practical, and I still have to filter what's worth keeping. Gemini in particular can go overboard with feature creep. And while the diagrams help early on, we still have to redraw them ourselves for anything polished.

Overall, these models are useful for speed, structure, and clarity. For CHANOS, I'd keep using ChatGPT for documentation, overviews, and architecture questions. It's better when I just need to get the essentials right. Gemini is more useful when I want to explore options or dig into why something works the way it does. As long as I stay in control of the direction and use them as a tool, not a crutch, they're a solid part of the process.

Scenario 3 - "For a solar-powered smart irrigation system like CHANOS, evaluate the different types of small pumps that could be used. Consider options such as submersible pumps, peristaltic pumps, diaphragm pumps, and solenoid valves. Compare them based on power consumption, flow control, ability to run dry, ease of integration with an ESP32, and reliability in unattended operation. The pump must be battery-compatible, capable of moving water from a small reservoir, and operate briefly every few hours. Which type would be most appropriate, and why?"

Asking an LLM to help compare pump options for a system like CHANOS can actually be pretty useful, especially when you're early in the process and trying to lay out what's even worth considering. It gives you a decent overview fast and you don't have to dig through random forum posts or scroll through 10 product pages just to get a feel for the tradeoffs. For example, both ChatGPT and Gemini pointed to peristaltic pumps as a strong option and called out valid reasons such as low power, safe to run dry, easy to control. That matches what I've seen so far, so it helps confirm I'm on the right track.

That said, this kind of thing only goes so far. The biggest issue is the model has no clue what the rest of your system looks like. It doesn't know what voltage rails you have, how much current your drivers can handle, or even whether you've already picked a specific

battery. So when it says “low power,” it’s not referencing your actual numbers—it’s just pulling from general info. It’s good for brainstorming, not decision-making.

Also, sometimes it gives you confident-sounding answers that aren’t 100% right. Like, it might say solenoid valves aren’t a fit because of high current draw, which is true for some types, but there are low-power latching versions too. It might gloss over that. Same thing with diaphragm pumps, it might say they’re overkill, but if you’re running a longer distance or need self-priming, they could actually make sense. You’ve got to fact-check everything if you’re serious about using it in your build.

Looking at Gemini’s response specifically, it kind of just agreed with what was already said. It repeated the reasoning for picking a peristaltic pump and didn’t really add anything new. That’s fine if you’re just looking for validation, but not super helpful if you’re trying to explore new angles or pressure test the idea. ChatGPT’s answer gave more detail on each pump type and explained where they work and where they fall short. That made it more usable for narrowing things down.

LLMs like this are a great tool when you’re stuck or trying to move quickly, but they don’t replace actually understanding your system or testing the parts. They help clarify your thinking, but they won’t make the call for you and they definitely won’t catch something like your pump pulling too much current for your transistor or your battery dying overnight. Good for the early phases, but you still have to own the final decisions.

Scenario 4 - “Design a simple and low-power water level sensor circuit that can detect when a reservoir is low, and interface it directly with an ESP32. Recommend components, explain the circuit, and describe how the ESP32 should read the signal.

Asking an LLM to help design a water level detection circuit for CHANOS ended up being useful in some ways, but there were also some clear limits. Both ChatGPT and Gemini gave the same general answer: use two stainless steel probes and a simple NPN transistor setup to detect when the reservoir runs low, then feed that into a GPIO on the ESP32. That idea works, and it lines up with a basic, low-power solution that could actually be implemented.

But neither one gave a circuit diagram. If you’re trying to build fast or get something ready for a schematic or PCB, not having a clear visual slows you down. The models explained it in text, which helps if you already know how to wire it up, but it leaves too much room for guesswork.

The other issue is that the answers don’t take into account the rest of the system. They assume you have spare GPIOs, that you’re fine on 3.3V power, and that there’s no conflict with anything else running on the ESP32. In our case, that’s not always true. The models don’t know how tight the pin count is or how much headroom we actually have in the battery budget.

Component suggestions were also pretty generic. They gave the right general idea: basic resistors and a transistor, but didn't suggest part numbers or address any edge cases like signal noise, probe corrosion, or what happens if the water is dirty. No mention of alternative sensing methods either, like float switches or capacitive sensors, which might actually be better depending on the environment.

Between the two responses, ChatGPT was more practical. It listed the parts, explained how they connect, and told you what the ESP32 should look for in the signal. You could take that and try it on a breadboard right away. Gemini mostly just agreed with the concept and restated the benefits. It didn't bring anything new to the table or offer different options. It's fine if you're just looking for reassurance, but it's not really helpful if you're trying to move the design forward.

Using AI for this kind of question is good for getting ideas on the table or confirming that you're headed in the right direction. But it's not plug-and-play. You still have to work through the integration, test the circuit, and make sure it fits into the system you're actually building. It's a tool, not a final answer. It saves time upfront, but it doesn't take the place of real design work.

Chapter 6 - Hardware Design

6.1 Hardware

The hardware design of the C.H.A.N.O.S. system is centered around delivering autonomous plant care through efficient sensing, sustainable power delivery, and low-maintenance operation. To achieve this, the system integrates a solar-powered battery architecture, moisture and light sensing, automated water delivery via a microcontroller-controlled pump, and modular PCBs for streamlined fabrication and testing.

Key design considerations included off-grid energy reliability, durable enclosure materials, minimal energy usage, and component compatibility with long-term unattended operation. All components—sensors, actuators, regulators, and structural materials—were selected to balance performance, simplicity, and manufacturability. Each subsystem (power, sensing, actuation, and control) was iteratively tested on a breadboard before integration into a custom PCB layout designed to support long-term use in a compact, indoor or semi-outdoor environment.

6.1.1 Materials and Overall Design

The physical design of C.H.A.N.O.S. was built with durability, modularity, and environmental exposure in mind. Since the system operates around live plants and water, we prioritized materials that could handle occasional splashes, moderate humidity, and sunlight exposure without degrading over time.

The main enclosure and pot were 3D-printed using PLA for ease of prototyping and iteration. While PLA isn't the most water-resistant material, its low cost and accessibility

made it a practical choice for our initial design. The structural frame was designed to support internal component separation keeping water-handling components like the pump and tubing away from the electronics in the lower shell.

The internal layout was split into functional zones. The top section houses the plant and soil moisture sensor, with a ring-shaped water delivery tube routed around the pot for even irrigation.

Below that, a mesh layer separates the soil from the drainage basin, where filtered water can collect and be reused. A dedicated compartment holds the water reservoir, with room for an ultrasonic sensor to measure remaining water levels. Electrical components, including the custom PCB, pump, charge controller, and battery, are mounted in the lower section of the enclosure, isolated from moisture. To maximize solar exposure, the solar panel is mounted on the rear face of the unit and angled slightly to improve capture efficiency when placed near a window. Cable routing and connector placement were kept clean to simplify assembly and reduce maintenance.

6.1.2 MCU

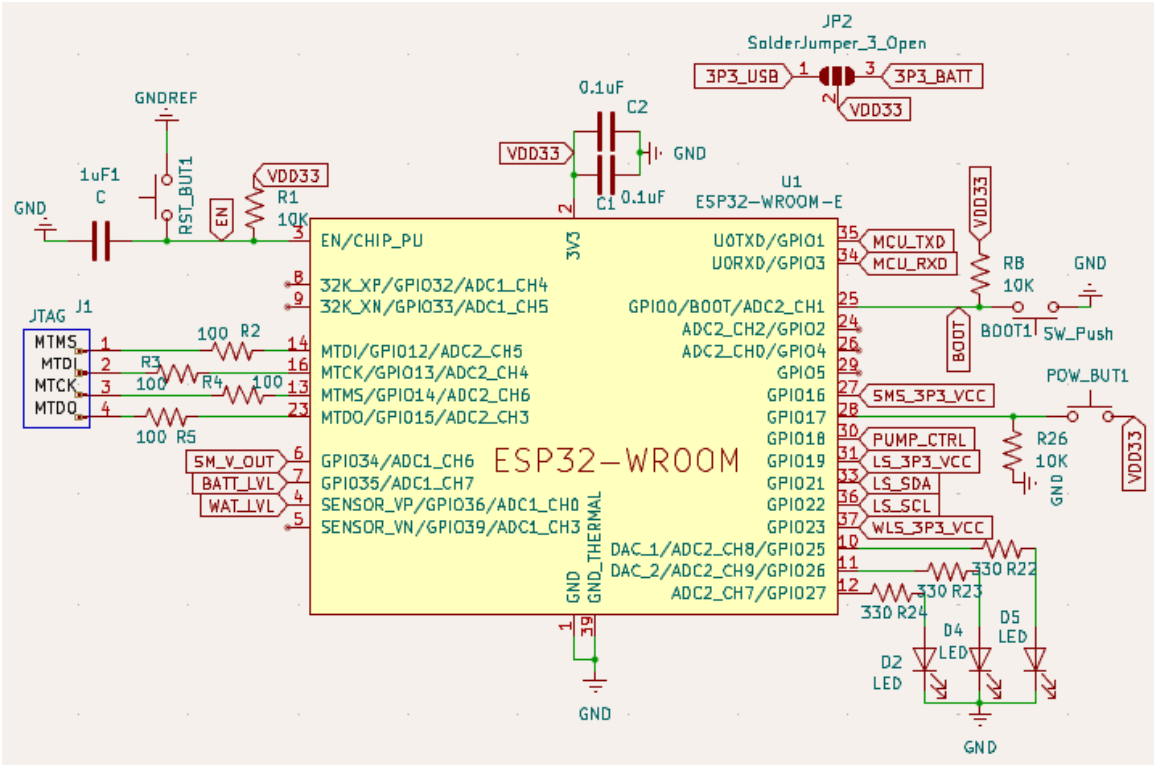


Figure 6.1.2.1 Microcontroller Schematic.

The ESP32-WROOM-32 series of microcontroller modules consists of the ESP32-D0WD chip integrated with several peripheral devices, such as internal RC and crystal oscillators, an RF antenna, and a SPI flash memory module. Within the ESP32-WROOM-32 datasheet, Espressif uses a schematic to show required components

and connections that must be made when designing a system using the ESP32-WROOM module. The datasheet details that the designer should connect pins 13, 14, 16, and 23 to a JTAG connector pin interface. The ESP32-WROOM modules are pre-flashed with ESP-AT software, which specifies pins 12-15 as a SPI slave interface, which means that the JTAG interface is not available to use in ESP32-WROOM modules [128]. During the PCB design, these pins were connected to a through-hole jumper connection in order to allow for possible alternate uses. The datasheet specifies certain ground connections for pin 1, 15, 38, and 39. The datasheet specifies decoupling capacitors for the 3.3V power supply to pin 2.

The datasheet details connections for a UART interface with TX/RX/GND corresponding to pins 34, 35, and any ground pin, respectively. This UART interface is necessary to download software to the ESP32 microcontroller. A component for UART to USB conversion is described in section 3.1.14.1, which will be used to connect our laptops to the UART interface. The datasheet specifies pin 25 to be connected to a switch; this switch must be activated in order to download software to the microcontroller.

Lastly, the datasheet specifies that the enable pin (pin 3) on the microcontroller should be connected to the output of a series RC circuit connected to the 3.3V power supply. The recommended values for the resistor and capacitor are 10KΩ and 1 uF, respectively, but the datasheet advises adjusting these values depending on the power-up timing scheme of the microcontroller. The timing scheme involves two requirements: t_{STBL} and t_{RST} have a minimum requirement of 50 us. The CHIP_PU (enable pin) voltage must reach $V_{IL,nRST}$ (0.6V) at least 50 us after VDD reaches a certain threshold voltage (t_{STBL}) or after the CHIP_PU voltage is discharged to 0.6V (t_{RST}). The charging equation for a capacitor in series RC circuit is listed in equation 6.1.2.1. By substituting the known values into this equation, the time that the capacitor is charged from 0V to 0.6V was obtained using equation 6.1.2.2. This value is significantly greater than 50us, so the recommended values were used in our microcontroller design.

Equation 6.1.2.1: $V_{cap} = V_i(1 - e^{-t/RC})$

Equation 6.1.2.2:

$$t_{0V-0.6V} = -RC\ln(-(\frac{V_o}{V_i} - 1)) = -10^4 \cdot 10^{-6}\ln(1 - \frac{0.6}{3.3}) = 0.002$$

Labels for the peripherals were also used to abstract their design from the microcontroller schematic. These labels represent connections and should be shown in the schematic for the respective peripheral. However, during PCB design, the corresponding GPIO pin number was printed on the silk-screen layer because the peripherals may need to be moved to different GPIO pins.

A section of LEDs were added for pins 10-12. These LEDs will allow for faster debugging and testing of the custom PCB design. They may also signal certain states or activities to the user.

6.1.3 Pump

We are using the Adafruit Peristaltic Liquid Pump, which is ideal for precise liquid handling applications. The pump operates using peristaltic tubing, which works by mechanically compressing flexible tubing to push fluid forward. This design ensures that the liquid remains in the tubing, making the pump suitable for applications requiring cleanliness or chemical resistance.

The pump is powered by a DC motor, typically requiring 12V at around 200-300 mA during normal operation. Because of this, it must be controlled through a motor driver circuit or a power MOSFET, rather than directly from a microcontroller.

A key benefit of this pump is its self-priming capability and ability to run dry without damage, making it a low maintenance pump. The included silicone tubing is flexible, allowing for easy integration into a variety of fluid control systems.

We control the pump by switching power to the motor using a logic level MOSFET and a PWM signal from a microcontroller, allowing us to adjust the speed and flow rate. A flyback diode is included across the pump terminals to protect the circuit from voltage spikes generated by the motor's inductive load.

This pump is well suited for our needs as a small scale liquid delivery system. Below is our schematic showing the connections between the MCU GPIO pin, the pump driver and the pump itself.

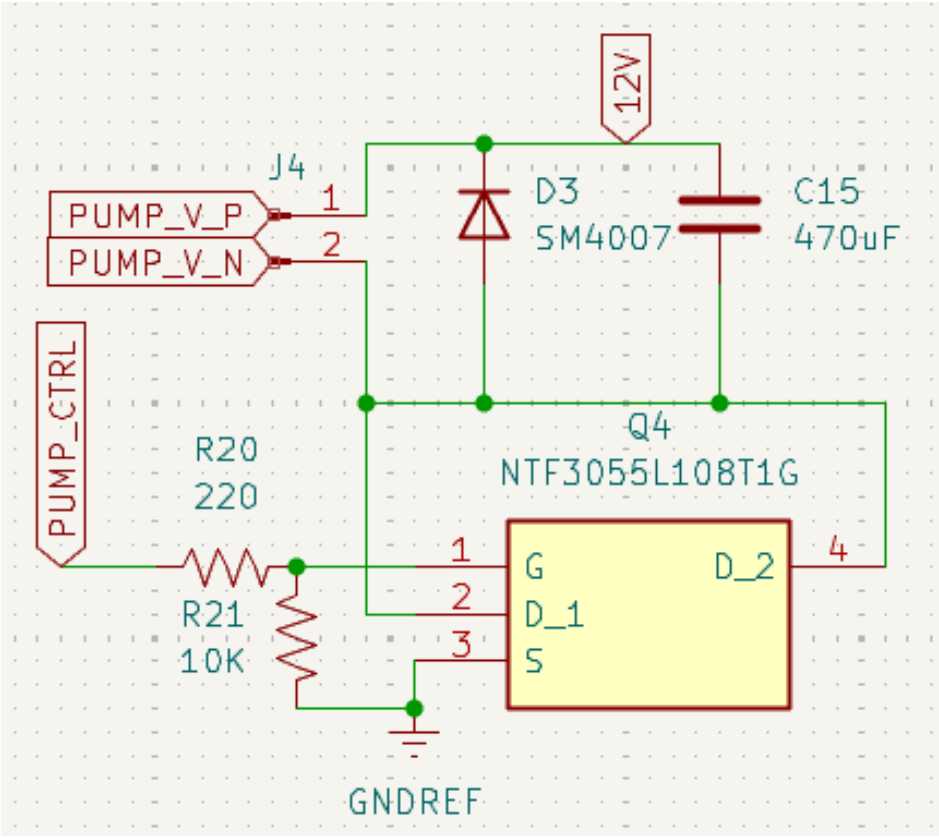


Figure 6.1.3.1 Pump Driver Schematic

6.1.4 Sensors

6.1.4.1 Light Sensor

A light sensor is an important part to the C.H.A.N.O.S. project as it measures solar intensity, giving us values which allow us to make informed decisions of when to move our plant during what is considered to be harmful light exposure. As stated in chapter 3, part selection, we decided to go with the BH1750 for our light sensor. It was chosen for its accuracy, reliability and compatibility with the ESP32 MCU. It provides direct lux readings over a wide range specifically (1 to 65,535), making it ideal for the monitoring of ambient light conditions that may impact plant health. The BH1750 uses the I2C communication protocol, which simplifies wiring and reduces the number of GPIO pins needed for integration.

Its low power consumption and compact footprint make it a well suited component for battery operated or space constrained design like our project. In comparison to other sensors including light dependent resistors, photodiodes, and the more complex TSL2561, the BH1750 offered the best balance between performance and ease of integration. Below is the pcb schematic for the sensor.

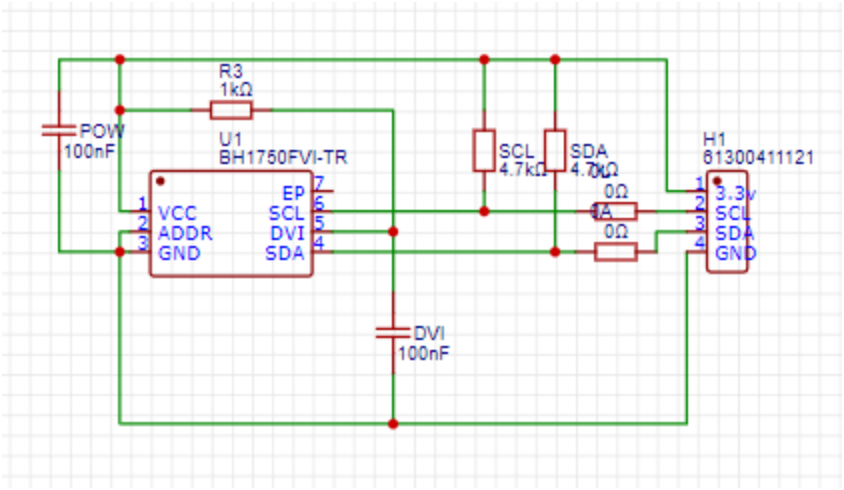


Figure 6.1.4.1.1 Light Sensor Schematic.

The BH1750 operates within a supply voltage range of 2.4V to 3.6V, making it compatible with the system's 3.3V rail or 3.3V pin on the ESP32. According to the datasheet, the sensor's supply current is typically 120 uA, with a maximum of 190 uA well within the limits of the available power supply.

- R4 and R5 serve as I2C bus pull-up resistors. The recommended value for these resistors is 4.7 kohms, which offers a good compromise between signal integrity and power consumption. Pull-up resistors are necessary to ensure that the SCL and SDA lines remain HIGH when not actively driven LOW by the ESP32.
- ADDR is typically left at its default logic level by connecting it to ground.
- R1 and C1 function as a low-pass filter to suppress electrical noise and protect the sensor from voltage transients.
- SCL (Serial Clock Line) and SDA (Serial Data Line) are used for I2C communication between the sensor and the microcontroller.
- DVI (Device Input Voltage) provides the internal operating voltage for the sensor's core circuitry.

6.1.4.2 Soil Moisture Sensor

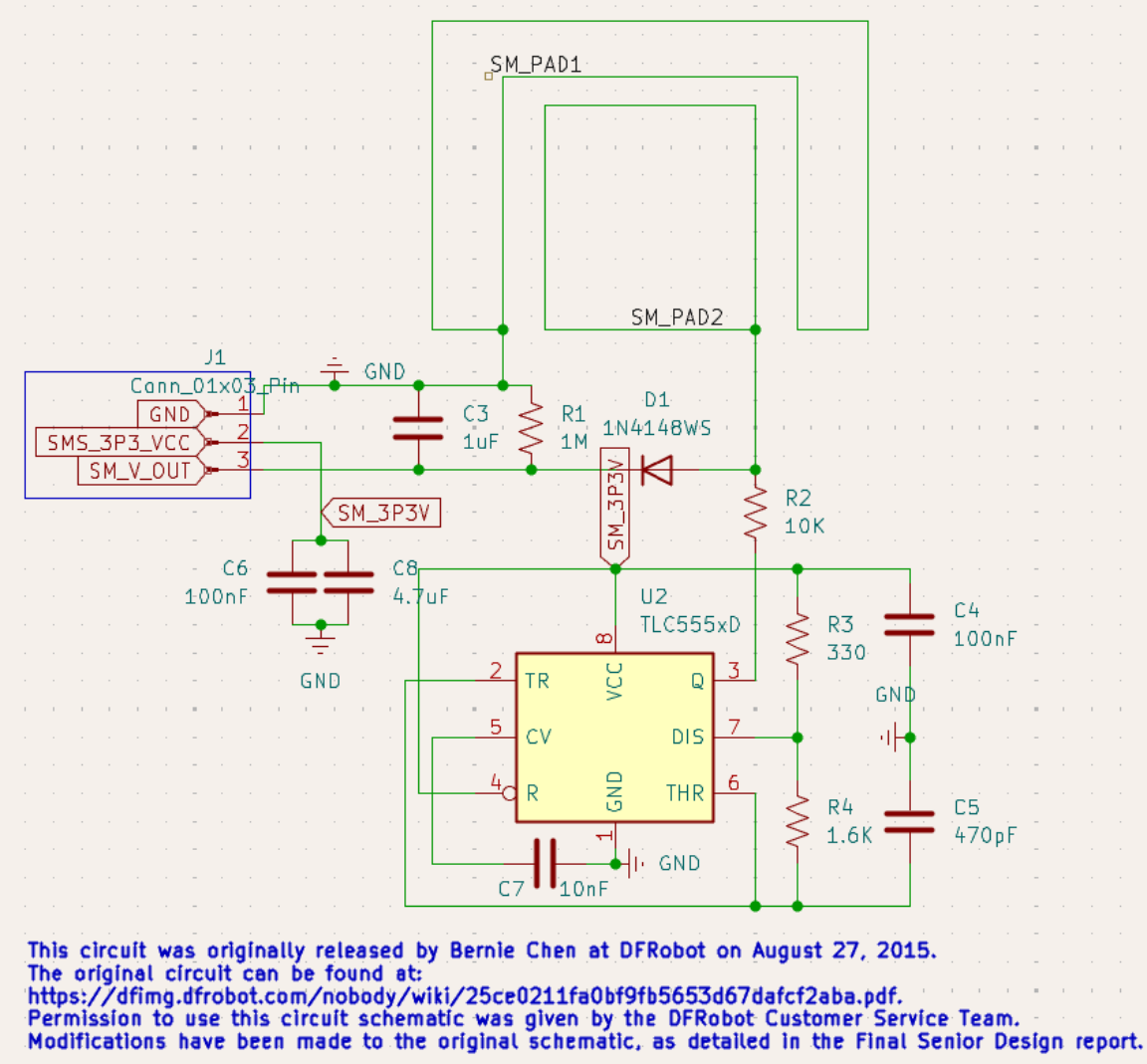


Figure 6.1.4.2.1. Soil Moisture Sensor Schematic.

The soil moisture sensor selected in section 3.2.7 was an open-source design by DFRobot. The schematic for the sensor is freely available [129]. This schematic was used to design a circuit describing the sensor using KiCad Schematic Editor software, shown in Fig. 6.1.4.2.1. The sensor uses two integrated circuits: TLC555CDR and XC6206P302 [130] [131].

The XC6206P302 is a positive voltage regulator produced by Torex. The supply voltage can be between -0.3V-7V and supply current is at most 3uA. The supply voltage signal in our schematic comes from the GPIO pin on the ESP32 microcontroller. The nominal output voltage is 1V less than the input voltage. Since the ESP32 GPIO pins nominally supply 3.3V, the output of the XC6206P302 would be 2.3V. The output current is between 300mA-500mA. In the sensor design, this output signal is connected to the input (pin 8) of the TLC555CDR IC. The recommended input voltage for the TLC555CDR is between 2V to 15V. However, the minimum voltage that the ESP32 GPIO pins may

supply is possibly 2.3V or 3V, depending on the ESP32 package used [73]. Considering the nominal output voltage of the XC6206 is 1V less than the input, the minimum voltage input to the TLC555CDR could be 1.3V or 2V. In order to avoid potential complications resulting from low input voltage to the TLC555CDR, the XC6206 was removed from the soil moisture schematic. Instead, the 3.3V supply from the ESP32 GPIO is connected directly to the TLC555 input pin. The decoupling capacitors were kept in order to provide additional stability.

The TLC555CDR is a timing circuit developed by Texas Instruments. In the "astable operation" configuration seen in Fig. 6.1.4.2.1, the trigger and threshold pins and capacitor are tied together at the same node. When the voltage at the trigger (pin 2) is less than the trigger level (1/3 Vcc), the timer output (pin 3) is set high and the discharge (pin 7) is open and has high impedance to ground. When set high, the output voltage is about 75%-80% to 95% of VDD. When the trigger voltage is less than 1/3 Vcc and the threshold is greater than 2/3 Vcc, the output is set low and the discharge is shorted to ground; thus, the 470pF capacitor discharges until the voltage is less than the 1/3 Vcc trigger level. When set low, the output voltage is at most 10%-15% of VDD. The durations of the high/low output signals are determined by the values of the resistors connected to the discharge pin and the capacitor connected to the trigger/threshold pins.

Equation 6.1.4.2.1:

$$t_{OH} = 0.693(R_8 + R_9)C_7 = 0.693(330 + 1600)470 \cdot 10^{-12} = 628.6 \text{ ns}$$

Equation 6.1.4.2.2: $t_{OL} = 0.693(R_9)C_7 = 0.693(1600)470 \cdot 10^{-12} = 521 \text{ ns}$

Equation 6.1.4.2.3: $f = \frac{1}{t_{OH}+t_{OL}} = 870 \text{ kHz}$

Equation 6.1.4.2.4: $D = \frac{t_{OH}}{t_{OH}+t_{OL}} \cdot 100\% = 55\%$

The output of the timing IC is connected to the anode of the 1N4148WS diode via a 10K resistor [132]. This diode has a forward voltage of at most 1V, though it is typically around 0.7V. The cathode of the diode is connected to pin 12 on the ESP32 microcontroller, which has ADC capabilities. The voltage at this pin should vary up to approximately 2.3V (given a forward voltage drop of 0.7V), depending on the capacitance between the two copper pads located in the top layer of the sensor PCB.

The ground signals are connected with a copper pad spread around the outer edge of the top layer of the sensor's PCB. Whereas, the point between the anode of the diode and the 10K resistor is connected to a copper pad in the middle of the top layer of the PCB. These pads essentially act as the plates of a capacitor, creating an RC circuit between the 10k resistor from the output of the timer to ground. Since the timer output is a square wave signal with a high period of 628.6ns, the capacitance between the copper pads determines the maximum voltage achieved before the square wave signal is low and the "capacitor" discharges. Increased moisture within the soil will increase the dielectric constant, which increases the capacitance between the pads. This will increase the time constant of the

RC circuit, increasing the amount of time required for the capacitor to charge to the equivalent voltage level and decreasing the average voltage with respect to ground.

The signal passes through a diode. The cut-off voltage of the diode, approximately 0.7V, prevents the low value of the square wave (~0.3-0.5V) from passing and instead outputs 0V as the low value of the square wave. A parallel RC circuit is placed after the diode output. This parallel RC circuit acts as a low-pass filter since the impedance of the capacitor decreases as frequency increases.

6.1.4.3 Water level Sensor

The water level sensor in this system is implemented using a Ximimark horizontal float switch, which acts as a digital input to detect the presence or absence of water at a specific, fixed point. This simple yet reliable sensor changes its orientation, closing or opening the circuit based on whether water lifts the float or not. To ensure accurate readings and to prevent false triggers caused by electrical noise, the sensor is connected to the microcontroller's GPIO pin through a pull-up resistor and a decoupling capacitor:

- A 4.7kΩ pull-up resistor connects the GPIO pin to 3.3V, ensuring a known HIGH state when the switch is open.
- A 0.1µF ceramic capacitor is placed between the GPIO pin and ground to filter noise and debounce the signal, ensuring the elimination of repeated false readings.
- When the float switch closes (indicating low water level), it pulls the GPIO pin to LOW.

This setup allows the microcontroller to reliably detect when the water level drops below the float's position, triggering certain actions such as disabling the pump to prevent dry running.

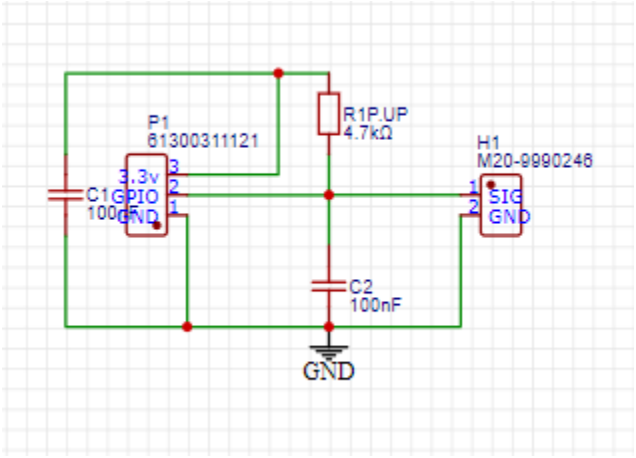


Figure 6.1.4.3.1 Water Level Sensor Schematic

6.1.5 Electrical Power System

The electrical power system was designed to support fully off-grid operation using solar energy and a rechargeable battery setup. At the core of the design is a 2-cell (2S) lithium-ion battery pack using protected Panasonic NCR18650GA cells. Each cell

provides 3500 mAh of capacity and supports discharge currents up to 5 A, which gave us the headroom we needed to handle both continuous loads and short inrush currents—especially from the pump. The pack's nominal voltage is 7.4 V, with a maximum charge voltage of 8.4 V.

For charging, we went with the BQ24650 from Texas Instruments, a dedicated MPPT buck charge controller built specifically for solar-powered systems. This chip stood out for its ability to dynamically regulate charge current based on input voltage, letting us extract the most power from the panel under changing light conditions. It supports up to 28 V on the input and offers configurable charge voltage, current, and MPPT setpoints. We set the MPPT tracking point to about 17.5 V to match the panel's Vmp. The BQ24650 can safely charge the battery while the system is under load, which was key for keeping everything running during daylight hours.

The panel we selected was the Voltaic Systems P108, a 3.5 W monocrystalline panel with an 18 V maximum power voltage and 22.5 V open-circuit voltage. While compact, it still provides enough headroom to trigger MPPT charging and keep the batteries topped off. Its ETFE coating makes it more durable than standard PET-laminated panels, which helps in hot or humid environments. The panel also comes with a 2.1 mm barrel connector and standard mounting holes, which made it easy to integrate into our system layout.

From the battery, power is distributed through two separate regulated rails. The 12 V rail powers the peristaltic pump and is generated using a TPS61088 boost converter. This chip was selected for its ability to provide high current at stable voltage, even when the battery voltage drops below 7 V.

The 3.3 V rail supplies the ESP32 and all system sensors. Rather than regulating this directly from the battery, we used a two-stage setup: first a buck converter down to 5 V, then a TPS62162 buck converter to bring that down to 3.3 V. The TPS62162 was chosen for its efficiency, low quiescent current, and clean output under light loads. This approach helps isolate noise and gives us tighter control over power delivery to the microcontroller. We intend to use a separate PCB for the PSU and another for the MCU and other.

Here's a summary of the key electrical components used in the power system:

Table 6.1.5.1 Electrical Component Comparison.

Component	**Function**	**Max Current**	**Input Range**	**Notes**
BQ24650	MPPT Li-ion Charger	4.0 A (adj)	5 V – 28 V	MPPT set to ~17.5 V
TPS61088	12 V Boost Regulator	2.5 A	2.7 V – 12 V	High-efficiency boost

TPS62162	3.3 V Buck Regulator	1.5 A	4 V – 17 V	Clean low-noise output
Panasonic NCR18650GA x2	2S Li-ion Pack	5 A	6.4 V – 8.4 V	Protected, 3500 mAh
Voltaic P108 Panel	Solar Input	3.5 W	Voc: 22.5 V	Vmp: 18.0 V

6.1.5.1 Charging and Power Storage

Charging and power storage are handled by a protected 2S lithium-ion battery pack supported by a dedicated MPPT-enabled charge controller. We used the BQ24650 from Texas Instruments due to its built-in support for solar input, programmable Maximum Power Point Tracking (MPPT), and constant-current/constant-voltage charging profile. It is a synchronous buck charger that supports input voltages up to 28 V and provides high-efficiency charging for 2-cell lithium-ion configurations.

The controller uses a programmable voltage divider at the MPPSET pin to set the MPPT threshold. According to the datasheet, the internal reference is 1.2 V. The equation used to set the MPPT voltage is:

$V_MPP = 1.2V \times (1 + R1/R2)$

To track a panel with a Vmp around 17.5 V, we selected R2 = 100 kΩ and solved for R1:

$R1 \approx 1.36\ M\Omega \rightarrow$ chosen value: 1.37 MΩ

Charge voltage was set using a resistor divider connected to the VFB pin. The internal reference voltage for VFB is 2.1 V, and the target battery voltage is 8.4 V:

$V_BAT = 2.1V \times (1 + R_top/R_bottom) \rightarrow R_top = 150\ k\Omega, R_bottom = 50\ k\Omega$

Table 6.1.5.1.1 Charging and MPPT Component Values.

Function	Equation	Target Value	Resistor Values Used
MPPT Setpoint	$V_MPP = 1.2V(1 + R1/R2)$	17.5 V	R1 = 1.37 MΩ, R2 = 100 kΩ
Charge Voltage Regulation	$V_BAT = 2.1V(1 + R_top/R_bottom)$	8.4 V	R_top = 150 kΩ, R_bottom = 50 kΩ
STAT1/STAT2 Pull-Ups	N/A	N/A	10 kΩ to 3.3 V
SRP/SRN Differential Cap	N/A	N/A	0.1 µF

The battery pack uses two Panasonic NCR18650GA cells in series. Each cell provides 3.5 Ah at a nominal 3.6–3.7 V, for a total pack energy of: E = 2 × 3.5 Ah × 3.7 V = 25.9 Wh.

Estimated Daily Usage:

- ESP32 + sensors: ~7.9 Wh

- Pump: ~0.12 Wh

- Total: ~8.0 Wh/day

Estimated Days of Autonomy: 25.9 Wh ÷ 8.0 Wh/day ≈ 3.2 days

The board we used during development was the BQ24650EVM-639, which we modified by replacing the MPPT and feedback resistors with our custom values. The board's status pins, onboard FETs, and layout provided a reliable and solderable test platform before committing to a custom design.

[Insert BQ24650 Board Image Here]

Table 6.1.5.1.2 Charging and Storage Electrical Characteristics.

Parameter	Value	Notes
Battery Chemistry	Li-ion (NCR18650GA x2)	2S protected pack
Nominal Battery Voltage	7.4 V	3.7 V per cell
Max Battery Voltage	8.4 V	4.2 V per cell
Rated Battery Capacity	3.5 Ah per cell	In series configuration
Total Energy Capacity	25.9 Wh (nominal)	Based on 7.4 V × 3.5 Ah
Charge Termination Voltage	8.4 V	Set via resistor divider
Charge Regulation Tolerance	±0.5%	Per BQ24650 datasheet
MPPT Setpoint Voltage	~17.5 V	Based on Vmp of panel
Max Input Voltage (Solar)	28 V	Absolute max for BQ24650
Typical Panel Input (P108)	18 V Vmp / 22.5 V Voc	Matched to MPPT
Estimated Daily Energy Usage	~8.0 Wh/day	ESP + pump
Estimated Days of Autonomy	~3.2 days	With no sunlight

6.1.5.2 Voltage Regulation

Voltage regulation is handled in two stages: a 12 V boost rail for the peristaltic pump and a 3.3 V buck rail for the ESP32 and sensors. These regulators were selected to support independent rails with low ripple, efficient operation, and thermal stability under load.

12 V Boost Converter – TPS61088

To generate the 12 V rail from the 2S battery pack, we selected the TPS61088, a high-efficiency, high-current boost converter from Texas Instruments. This converter is designed to operate with input voltages as low as 2.7 V and can deliver over 2 A at 12 V, which gives us plenty of headroom for both steady-state operation and inrush current when the pump activates.

The total power output required by the peristaltic pump was calculated using:

$P = V \times I = 12\ V \times 0.3\ A = 3.6\ W$

To estimate daily energy consumption:

$E_pump = 3.6\ W \times (120\ sec / 3600\ sec/hr) = 0.12\ Wh/day$

We designed the circuit using TI's application notes and Webench guidance. Key supporting components include a 68 µH inductor, a 1N5822 Schottky diode, and 100 µF input/output capacitors to reduce ripple and support current delivery.

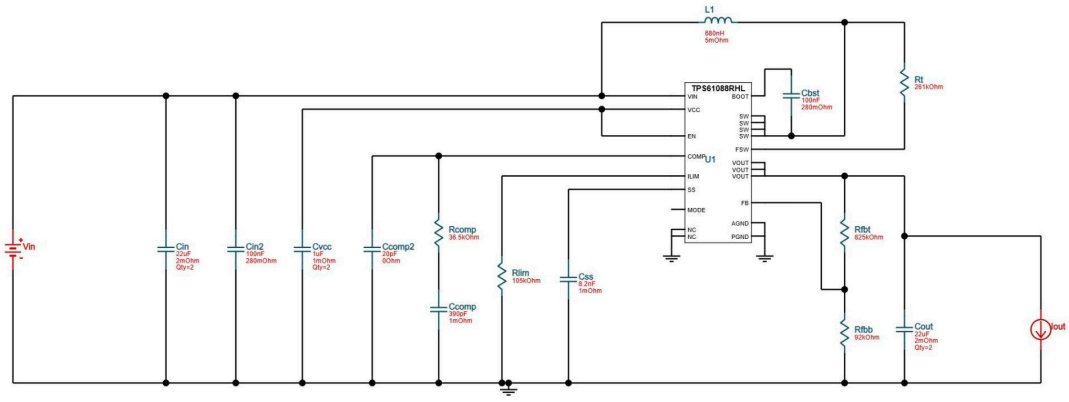


Figure 6.1.5.2.1 Boost Converter

Table 6.1.5.2.1 Pump Electrical Characteristics.

Parameter	Value	Notes
Supply Voltage	12 V	Regulated by TPS61088

Steady-State Current	250–300 mA	Varies with head pressure/load
Peak Inrush Current	~350 mA	During startup
Daily Runtime	~2 minutes	Four 30-second watering cycles
Daily Energy Usage	~0.12 Wh	Based on 3.6 W × (120/3600) hr

3.3 V Buck Converter – TPS62162

The 3.3 V logic rail is regulated using the TPS62162, a synchronous buck converter optimized for high-efficiency, low-noise embedded designs. It steps down from a 5 V intermediate rail and powers the ESP32 microcontroller along with various GPIO-connected sensors.

The total estimated power draw of the 3.3 V rail was calculated as:

$P = V \times I = 3.3\ V \times 0.1\ A = 0.33\ W \rightarrow E = 0.33\ W \times 24\ hr = 7.9\ Wh/day$

However, due to sleep cycles and GPIO-based sensor toggling, the actual draw is likely closer to 2.5–3.0 Wh/day.

The TPS62162 was chosen over common LDOs like the AMS1117 due to its superior efficiency (~95%) and much lower quiescent current, which is critical for battery conservation. It supports up to 1.0 A continuous current, with integrated protections including short-circuit shutdown and thermal overload.

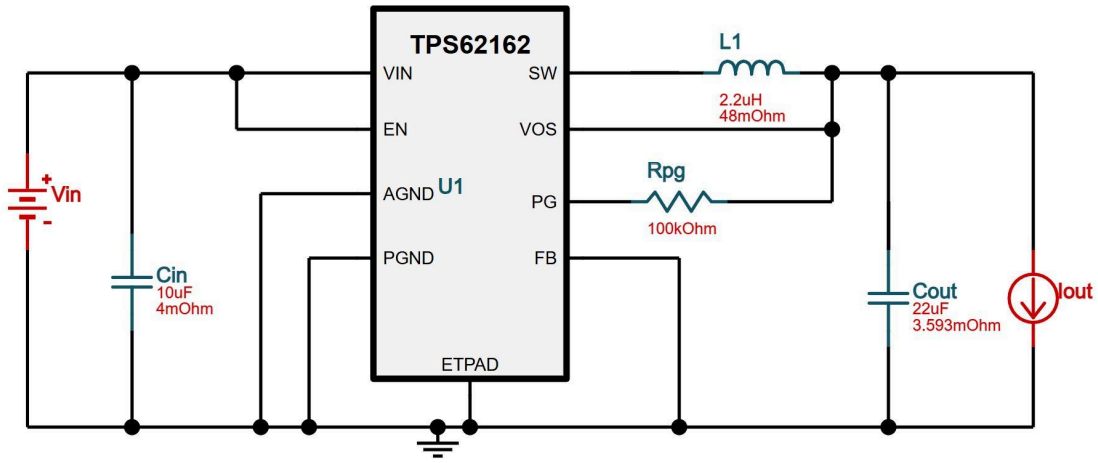


Figure 6.1.5.2.2 Buck Converter

Table 6.1.5.2.2 Microcontroller + Sensor Rail Characteristics.

Load Component	Supply Voltage	Typical Current
ESP32 DevKit V1	3.3 V	60–100 mA
Light Sensor (I²C)	3.3 V	<2 mA
Soil Moisture Sensor	3.3 V (GPIO)	~10–15 mA
Water Level Switch	3.3 V (GPIO)	Negligible
Total (average)	3.3 V	~100–150 mA

Table 6.1.5.2.3 Regulator Summary Comparison.

Parameter	TPS61088 Boost (12 V Rail)	TPS62162 Buck (3.3 V Rail)
Input Voltage Range	2.7 V – 12 V	3 V – 17 V
Output Voltage	12 V	3.3 V
Max Output Current	>2 A continuous	1 A continuous
Switching Frequency	1.2 MHz (typical)	1.25 MHz (typical)
Efficiency (typical)	~92% @ 1 A	~95% @ 0.5 A

Protection Features	OCP, OTP, UVLO, soft-start	SCP, UVLO, thermal shutdown
Package Type	3 mm × 3 mm QFN	2 mm × 2 mm QFN

6.1.6 Final Overall psu design

The power system is built to support off-grid operation using a 2S lithium-ion battery pack, solar charging, and two regulated voltage rails. The design consists of three key stages: charging and storage, voltage regulation, and GPIO signal interfacing.

The charging stage begins with the Voltaic Systems P108 panel, a monocrystalline 8.6 W solar module with a Vmp of 17.8 V and Voc around 22 V. This panel feeds into the BQ24650 charge controller, a synchronous buck converter with integrated MPPT. MPPT behavior is tuned via a resistor divider targeting 17.5 V. A second divider sets the battery regulation voltage to 8.4 V, appropriate for a 2S lithium-ion pack. Charge current is limited to 0.5 A using an 80 mΩ sense resistor, which matches the panel's maximum power point current. The system uses a protected 2S pack made from two Panasonic NCR18650GA cells (7.4 V nominal, 8.4 V max), selected for high energy density and current capability.

Voltage regulation is handled by two downstream regulators. A TPS61088 boost converter steps the battery voltage up to 12 V for the peristaltic pump, using a 10 µH inductor, 100 µF output capacitor, and a Schottky diode. The converter supports over 2 A output current and operates at up to 2.2 MHz switching frequency. For logic and sensing, a separate buck regulator generates 5 V, which is then stepped down to 3.3 V via a TPS62162 synchronous buck. This 3.3 V rail powers the ESP32 and environmental sensors. Each regulator is supported by appropriate input/output capacitance and follows layout guidance from TI's datasheets and EVMs.

A GPIO-accessible voltage sense pin is routed from the VFB node (at the junction of the battery divider resistors) to an ESP32 ADC. This allows firmware-level monitoring of battery level without relying solely on charger status signals. Pull-up resistors are placed on STAT1 and STAT2 to drive LEDs for basic visual charge state feedback. Additional headers are included for development, testing, and future expansion. The entire system is implemented on a custom PCB with careful attention to grounding, power trace width, and noise-sensitive routing.

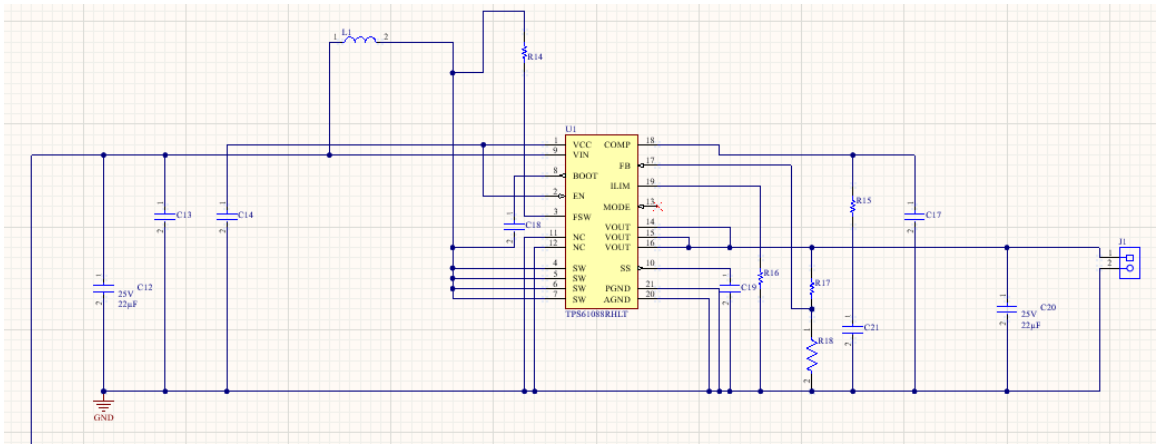


Figure 6.1.6.1 12V Boost Converter Schematic.

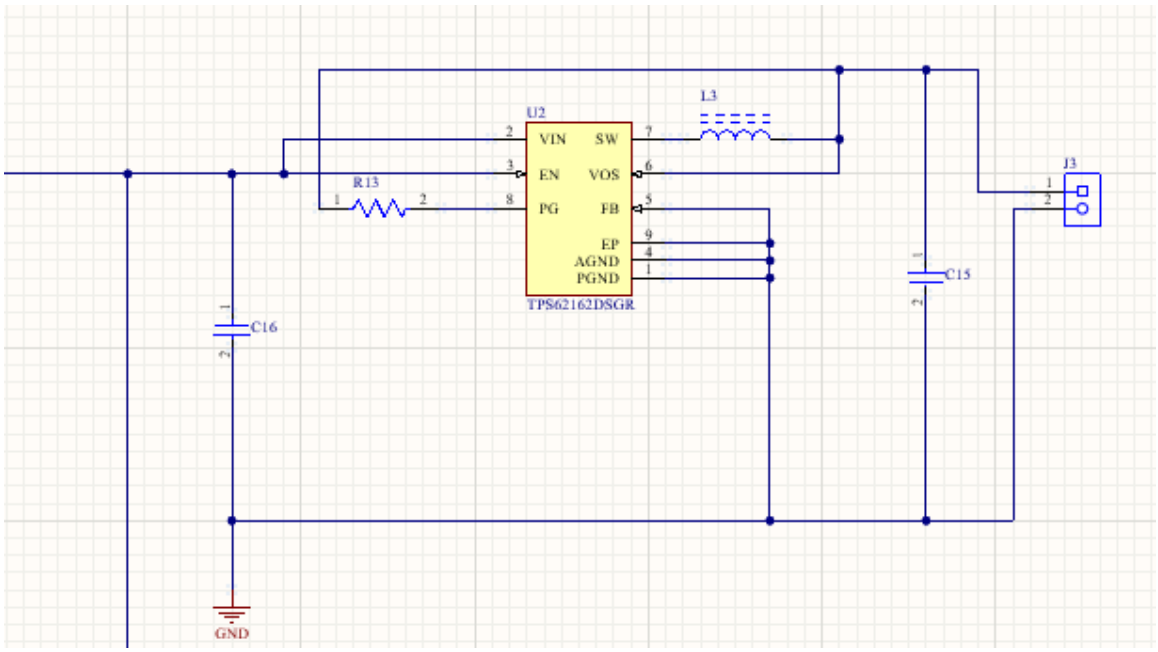


Figure 6.1.6.2 3.3V Buck Converter Schematic.

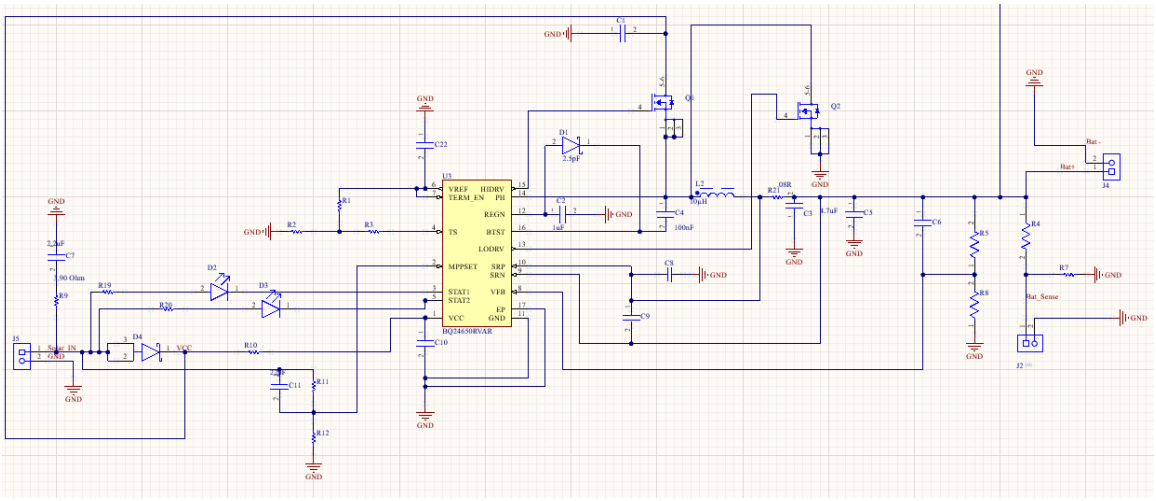


Figure 6.1.6.3 Charge Controller Schematic.

Description	Designator
Capacitor, Ceramic, 10 µF, 50 V, X7R, 1210	C1
Capacitor, Ceramic, 10 nF, 25 V, X7R, 0805	C10
Capacitor, Ceramic, 22 µF, 100 V, X5R, 0805	C11
Capacitor, Ceramic, 22 µF, 25 V, X5R, 1206	C12
General Purpose Ceramic Capacitor, 0805, 100nF, 10%, X7R, 0.15, 25V	C13
Capacitor, Ceramic, 1 µF, 25 V, X7R, 0603	C14
Chip Multilayer Ceramic Capacitors for General Purpose, 1206, 22uF, X7R, 15%, 10%, 10V	C15
Chip Multilayer Ceramic Capacitors for General Purpose, 0805, 10uF, X5R, 15%, 20%, 25V	C16
Capacitor, Ceramic, 20 pF, 50 V, X7R, 0805	C17
General Purpose Ceramic Capacitor, 0805, 100nF, 10%, X7R, 0.15, 25V	C18
Capacitor, Ceramic, 8.2 nF, 16 V, X7R, 0805	C19
Capacitor, Ceramic, 1 µF, 50 V, X7R, 0805	C2
Capacitor, Ceramic, 22 µF, 25 V, X5R, 1206	C20
Capacitor, Ceramic, 390 pF, 50 V, X7R, 0805	C21
Capacitor, Ceramic, 1 µF, 25 V, X7R, 0805	C22
Capacitor, Ceramic, 10 µF, 50 V, X7R, 1210	C3
Capacitor, Ceramic, 0.1 µF, 50 V, X7R, 0805	C4
Capacitor, Ceramic, 4.7 µF, 50 V, X7R, 1210	C5
Capacitor, Ceramic, 22 pF, 50 V, C0G, 0603	C6

Multilayer Ceramic Capacitors 2.2µF ±10% 25V X7R SMD 0805	C7
Capacitor, Ceramic, 1 µF, 50 V, X7R, 0805	C8
Capacitor, Ceramic, 0.1 µF, 16 V, X7R, 0805	C9
DIODE SCHOTTKY 40V 380MA SOD523	D1
LED, Green, SMD, Clear Lens	D2
LED, Green, SMD, Clear Lens	D3
Diode, Schottky, 40 V, 10 A, POWERDI5	D4
Terminal Block, 2-Pin, THD, 5 mm pitch, 13.5 A, 400 V, Phoenix 1729128	J1
PCB Terminal Block, 13.5 A, 400 V, 2-Pin THD, RoHS, Bulk	J2
PCB Terminal Block, 13.5 A, 400 V, 2-Pin THD, RoHS, Bulk	J3
PCB Terminal Block, 13.5 A, 400 V, 2-Pin THD, RoHS, Bulk	J4
PCB Terminal Block, 13.5 A, 400 V, 2-Pin THD, RoHS, Bulk	J5
Inductor, Shielded, 0.68 µH, 20%, 4.4 A Isat, 26 mΩ DCR, 1210 (Murata)	L1
Inductor, Shielded, 10 µH, 5.5 A, 34 mΩ, SMD	L2
Inductor, Shielded, 2.2 µH, 4.5 A, 38 mΩ, 6.0×6.0 mm, SMD	L3
NTMFS5C430NLT1G Single N−Channel Power MOSFET 40V, 200A, 1.4mΩ Power MOSFET 40 V, 1.5 mohm, 200 A, Single N-Channel	Q1
NTMFS5C430NLT1G Single N−Channel Power MOSFET 40V, 200A, 1.4mΩ Power MOSFET 40 V, 1.5 mohm, 200 A, Single N-Channel	Q2
Resistor, 5.23 kΩ, 1%, 0.1 W, 0603	R1
Resistor, 10 Ω, 1%, 0.5 W, 1206	R10
Resistor, 499 kΩ, 1%, 0.125 W, 0805	R11
Resistor, 36.0 kΩ, 1%, 0.125 W, 0805	R12
RES Thick Film, 100kΩ, 1%, 0.125W, 100ppm/°C, 0805	R13
Resistor, Thick Film, 261 kΩ, 1%, 0.5 W, 1206	R14
Resistor, Thick Film, 36.5 kΩ, 1%, 0.125 W, 0805	R15
Resistor, Thick Film, 1.05 MΩ, 1%, 0.125 W, 0805	R16
Resistor, Thick Film, 825 kΩ, 1%, 0.125 W, 0805	R17
Resistor, Thin Film, 92 kΩ, 0.1%, 0.125 W, 0805	R18
Resistor, 10 kΩ, 5%, 0.125 W, 0805	R19
Resistor, 10 kΩ, 5%, 0.125 W, 0805	R2
Resistor, 10 kΩ, 5%, 0.125 W, 0805	R20
Resistor, Current Sense, 0.08 Ω, 1%, 0.5 W, 0805	R21
Resistor, 100 Ω, 1%, 0.125 W, 0805	R3
RES 100K OHM 1% 1/8W 0805	R4
Resistor, Chip, 300 kΩ, 1%, 1/8 W, 0805	R5

Chip Resistor, 47 KOhm, +/- 1%, 0.125 W, -55 to 155 degC, 0805 (2012 Metric), RoHS, Tape and Reel	R7
Resistor, Chip, 100 kΩ, 1%, 1/8 W, 0805	R8
Resistor, 3.9 Ω, 1%, 0.125 W, 0805	R9
IC REG BOOST ADJ 10A 20VQFN	U1
Buck Step Down Regulator with 3 to 17 V Input and 3.3 V Output, -40 to 85 degC, 8-Pin WSON (DSG), Green (RoHS & no Sb/Br)	U2
High Efficiency Synchronous Switch-Mode Charger Controller- Solar Battery Charger, 5 V, -40 to 85 degC, 16-pin VQFN (RVA), Green (RoHS & no Sb/Br)	U3

6.1.7 USB-UART Conversion Circuit

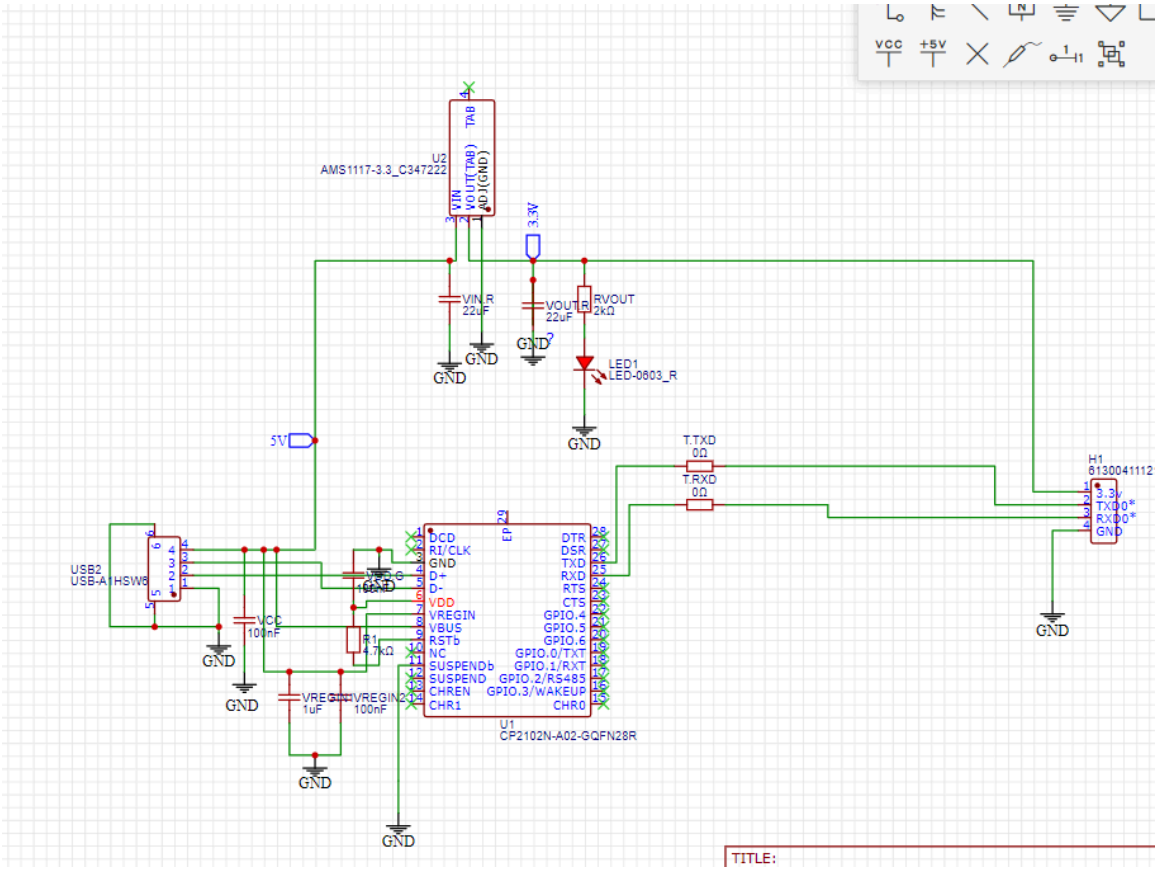


Figure 6.1.7.1 USB-UART Conversion Schematic.

The USB-UART conversion circuit was developed based on the similar subcircuit within the “SparkFun ESP Thing” design by Jim Lindblom and released under the Creative Commons Attribution Share-Alike 4.0 License [133]. The USB interface includes four pins: VCC (5V input), D+/- (data), and GND. The data pins D- and D+ are connected to the output of an RC circuit for which the input is the USBDM and USBDP output pins of the FT231XS IC [133]. The output data signals from the FT231XS are the TXD (transmission) and RXD (receiving) signals which must be connected to the RXD and TXD pins of the MCU, respectively [133]. This is because the data transmitted from the

USB (via the FT231XS) is received by the MCU, and vice versa. The 5V signal supplied by the USB is connected to the VCC (5V) input pin of the FT231XS IC [133]. Decoupling capacitors and a fuse are used to prevent noise and sudden overcurrent from compromising the system, respectively. The 5V signal from the USB interface is also an input to the AP2112K voltage regulator, which produces a 3.3V, 600mA output signal that may be used to power the ESP32. This voltage regulation subcircuit was altered from the schematic to just align with the recommended configuration in the AP2112 datasheet. The RC circuit for the MCU power supply pin is already present in the MCU schematic. The 3V3OUT, VCCIO, and active low RESET pins are connected at a node along with a bypass capacitor to GND [133]. This ensures that power is supplied to the IO pins and that the reset is not triggered.

The inverted RTS (request to send) and inverted DTR (data terminal ready) output pins on the FT231XS are used to control and auto-reset for the ESP32 [133]. The signals are inverted from the RTS and DTR signals, so they output a voltage of approximately 3.3V when RTS or DTR signals are low (the inverted are high). Both are connected to the base terminals of npn transistors via a 10K resistor, which decreases the voltage by about 1.25V (0.125uA*10K=1.25V).

One transistor is connected to the !DTR resistor at the base terminal; the !RTS pin at the emitter terminal; and the EN/CHIP_PU ESP32 pin at the collector terminal. The EN/CHIP_PU pin on the ESP32 should be pulled up to 3.3V minus the small voltage drop across the 10K resistor. When the RTS and DTR signals are high, !RTS and !DTR are low, and the base and emitter terminals are both approximately 0V. The transistor is in a cutoff state because Vbe = 0. When DTR is high and RTS is low, !DTR is low and the !RTS is high, so there is 0V at the base and ~2.1-3.3V at the emitter. Thus, the transistor is again in a cutoff state because Vbe and Vbc are negative. When the DTR and RTS signals are both low, !DTR and !RTS are both high, so there is 2.1V at the base and ~2.1-3.3V at the emitter. Again, Vbe and Vbc are negative, so the transistor is cutoff. Finally, if the DTR is low and RTS is high, then !DTR is high and !RTS is low, so there is 2.5V at the base and 0V at the emitter. The transistor is in a forward active state because $V_{bc} < 0$ and $V_{be} > 0$. This allows current to flow from the collector to emitter, essentially shorting the reset pin of the MCU. This causes a reset to be triggered, and is released when both DTR and RTS signals are low (!DTR and !RTS are high), which occurs after the MCU is programmed and the boot switch is released.

A similar analysis can be applied to the second transistor which is connected to the boot/GPIO 0 pin at the collector, the RTS pin at the base, and the DTR pin at the emitter. The boot switch is active low, meaning the voltage at the pin is 0 when the switch is pressed. When DTR and RTS are both low, !DTR and !RTS are high, so there are 3.3V at the emitter, 2.1V at the base, and 0V at the GPIO0 pin. While this would cause the transistor to be in reverse active state, it has practically no effect on the design because the other transistor is already in a cutoff state. If DTR is high and RTS is low and the boot switch is not pressed, there is 0V at the emitter, 2.1V at the base, and ~2.1-3.3V at the collector. The second transistor is forward active, but this would not affect the first transistor which is already in a cutoff state. Lastly, the second transistor may be saturated

if the switch is pressed, DTR is high, and RTS is low. However, because the collector and emitter are both 0V, saturation would not change the voltage at the base terminal of the first transistor.

Table 6.1.7.1 Auto-reset Signals and States.

DTR	RTS	Boot Pressed	Transistor 1	Transistor 2
0	0	N	Cutoff	Cutoff
0	0	Y	Cutoff	Reverse Active
0	1	Y/N	Forward Active	Cutoff
1	0	N	Cutoff	Forward Active
1	0	Y	Cutoff	Saturated
1	1	Y/N	Cutoff	Cutoff

In conclusion, it seems that the DTR signal is low and the RTS signal is high during reset, so that the enable pin on the ESP32 is connected to ground [127]. At the end of the auto-reset, the DTR signal goes high, which causes the capacitor at the enable pin to begin charging from the 3.3V source. After the voltage at the enable pin begins increasing, the RTS signal goes low [127]. If the BOOT switch is correctly pressed, then its corresponding transistor is saturated and the GPIO0/BOOT pin on the ESP32 is shorted to ground. However, if the BOOT switch is not pressed, the transistor is only in a forward active state, and the ESP32 pin may not reach the sufficient voltage level. After serial communication is complete and the ESP32 is programmed, the DTR and RTS signals should both be low, and the transistors are both in a cutoff state [127]. Note that the output pins on the FT231XS and base/collector terminals of the transistors are the inverted DTR and RTS signals.

Chapter 7 - Software Design

7.1 System Initialization Process

This section describes the system-level abstraction of the initialization software procedure. The scope of the section includes the user's actions, the actions and state of the ESP32 microcontroller, and the actions and state of the web server. The purpose of this section is to describe the initialization process in a manner that achieves the goals and objectives detailed in section 2.3 while also permitting a flexible, open-ended software implementation of the web server/application and ESP32 microcontroller.

7.1.1 Initial Setup

For our embedded application, our initialization process is one of the most important aspects of the program, as it ensures that the device is connected to wifi and has necessary plant information available to it.

The program which controls the pot begins in a pre-initialization mode, where power usage is lowered and no communication is available. If a previous configuration exists, the program immediately reads from this configuration to return to its running state in the event of a sudden loss of power that interrupted a previously running program. Initialization mode is triggered by a physical button located on the pot, referred to from here on as the "pairing button." Once the pairing button is pressed, the pot will startup its initialization services, such as wireless connectivity. The pot will start by advertising itself as a WiFi access point. The user, upon connecting to this access point, will be taken to a login page, where they will give the pot information to connect to their WiFi network, and be redirected to the server's account login or creation page. Once the user has been forwarded to the server, the pot will then attempt to contact the server itself, in order to pass along its information as well as the information from the user, namely the ip information of the user's device, which can then be used by the server to seamlessly link the user's account to a given pot. After this point, the pot will exit its initialization mode, and store information such as WiFi credentials in a file, which can be utilized should the pot ever experience a total loss of power which requires a restart of the program upon reboot.

Once the pot is initialized, it will enter a waiting mode, where it will listen to the server for details regarding the plant that will be placed into the pot. The necessary details for this plant, once sent to the pot, will then be used to initialize necessary parameters, such as defining the timer intervals and moisture levels to maintain. These plant parameters do not need to be written to a file like other initialization parameters, however, as the pot will be able to send a request to the server should it need to reset its plant parameters on a power loss event.

Once the pot receives necessary configuration parameters for the plant from the server, it will populate these parameters, initialize necessary objects, such as the SensorData object and the SensorReaders, and perform its first sensor readings, which will be reported to the server for the user to view. It will then enter running mode, where it will enter a low power mode until awoken by the timer to read from the sensors on the pot, or by another event that requires exiting the low power mode.

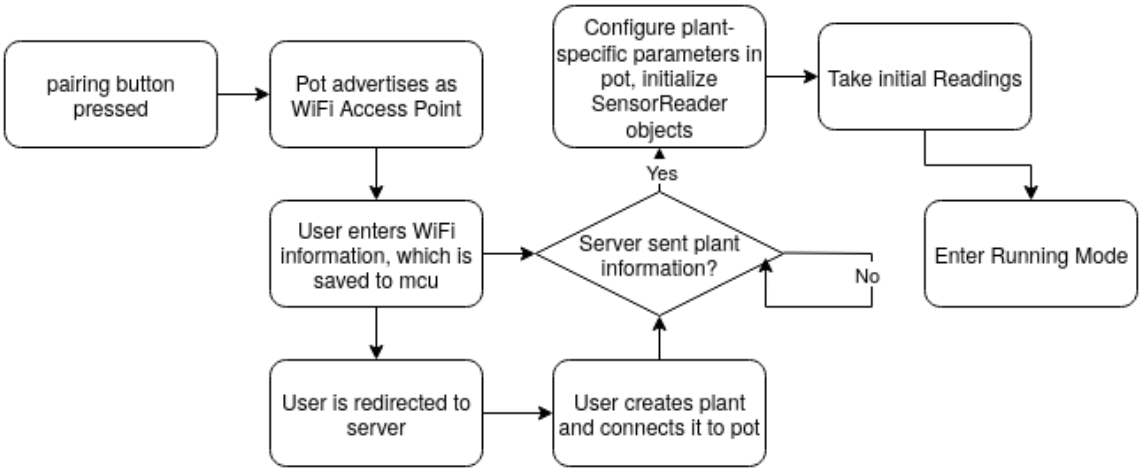


Figure 7.1.1.1 Initialization Setup Flowchart.

7.1.2 Restoring State After a Power Loss Event

Restoring state after power loss begins with the program restarting in its base pre-initialization mode. Upon entering pre-initialization mode, it will first check if a configuration file containing necessary network connection details exists. If this file exists, it will use the information in the file to reconnect itself to the internet.

After the pot has reconnected to the network, it will contact the remote server to request details of the plant which it contains. If no plant exists, the pot will enter waiting mode, and listen to the server for incoming plant details. If a plant exists, the server will send over plant information, which the pot will use to repopulate fields which were lost. The pot will then reenter running mode.

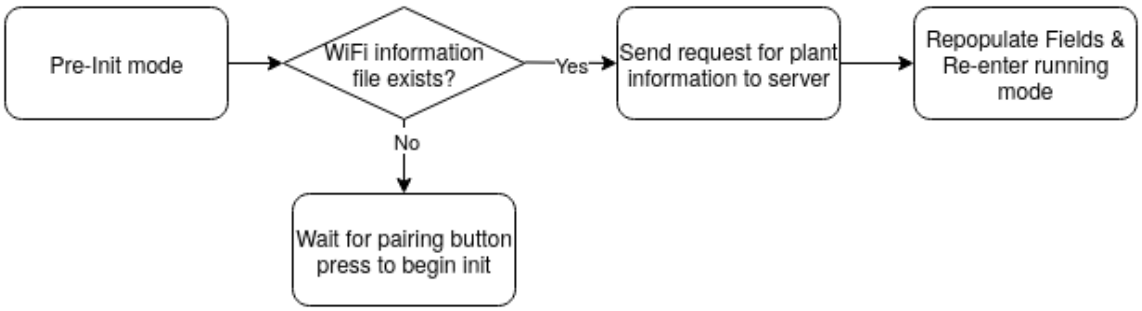


Figure 7.1.2.1 Power-Loss Restoration Flowchart.

7.2 Microcontroller Software Architecture

This section describes the software states and procedures implemented in the ESP32 microcontroller software. The scope of the section is mostly limited to the microcontroller and its interactions with peripheral components, such as sensors. The purpose of this section is to summarize the blocks of code and programming statements used to achieve the goals and objectives detailed in section 2.3. The SETUP mode and its corresponding states are a model for the implementation of the initialization process

described in section 7.1 of this report. Since this section is a lower-level abstraction of the software compared to section 7.1, the following architecture represents the current software loaded into the microcontroller and will be updated accordingly.

The microcontroller generally operates between the SETUP and OPERATION modes. Each mode is a collection of states and their relationships (transitions). After power-on, the microcontroller begins in the Pre-Initialization state within the SETUP mode. In this state, the microcontroller checks if there is a file or data on the non-volatile memory of the ESP32. This data can be stored in the flash memory of the ESP32 by using libraries such as EEPROM.h or LittleFS. If sufficient data is available (e.g. the SSID and password of the client Wi-Fi network), then the microcontroller enters the ESP32-STA Initialization state. If not, then the microcontroller enters the ESP32-AP state and the required network information is gathered from the user. After connecting to the Wi-Fi network, the ESP32 enters the Sensor Initialization state, where necessary sensor threshold and baseline values are obtained from either the user or sensor components.

After initializing the sensor variables, the ESP32 enters the Sensor Check state within the OPERATION mode. In this state, the sensor values are monitored and the necessary actions are taken to water and maintain the plant. After monitoring the sensors, the microcontroller enters the Sleep state in order to conserve power. The ESP32 microcontroller is woken up by a timer. Everytime the microcontroller is woken up by the timer, the Sensor Update state is entered, where the ESP32 sends HTTP GET requests to obtain the current threshold values stored in the web server. If any values have changed or if it is time to monitor the sensors, the ESP32 re-enters the Sensor Check state.

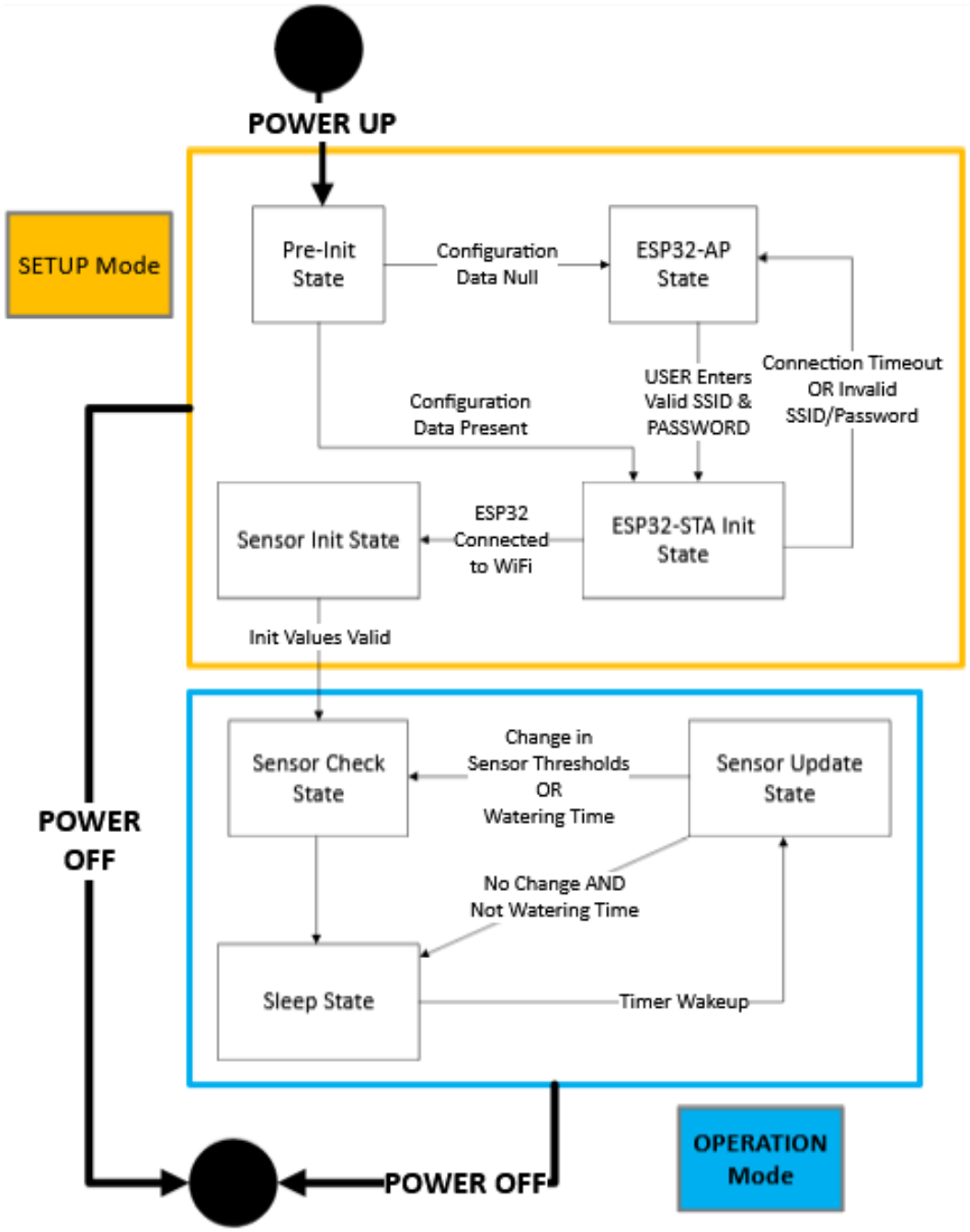


Figure 7.2.1 Overall Software State Diagram.

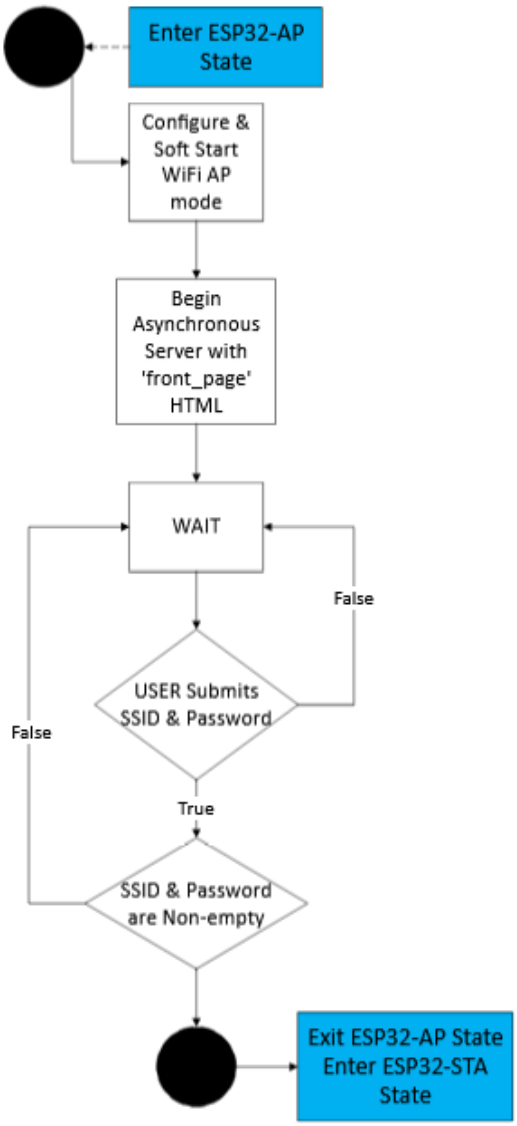


Figure 7.2.2 Activity Diagram for ESP32-AP State.

Upon power-up (whether it is initial power-up or after turning off the microcontroller power), the microcontroller initially enters the ESP32-AP (access point) state within the SETUP mode. In this state, the microcontroller instantiates itself as an access point with a predefined IP address, SSID, and password. The user would be given these values in a secure method, such as a paper delivered along with the product. After the user connects to the ESP32 access point, the ESP32 will use an HTTP connection to display an HTML page in which the user enters the SSID and password of their Wi-Fi network. If the SSID and password entered by the user are not null or empty, then the ESP32 exits the ESP32-AP state and enters the ESP32-STA Init (station initialization) state.

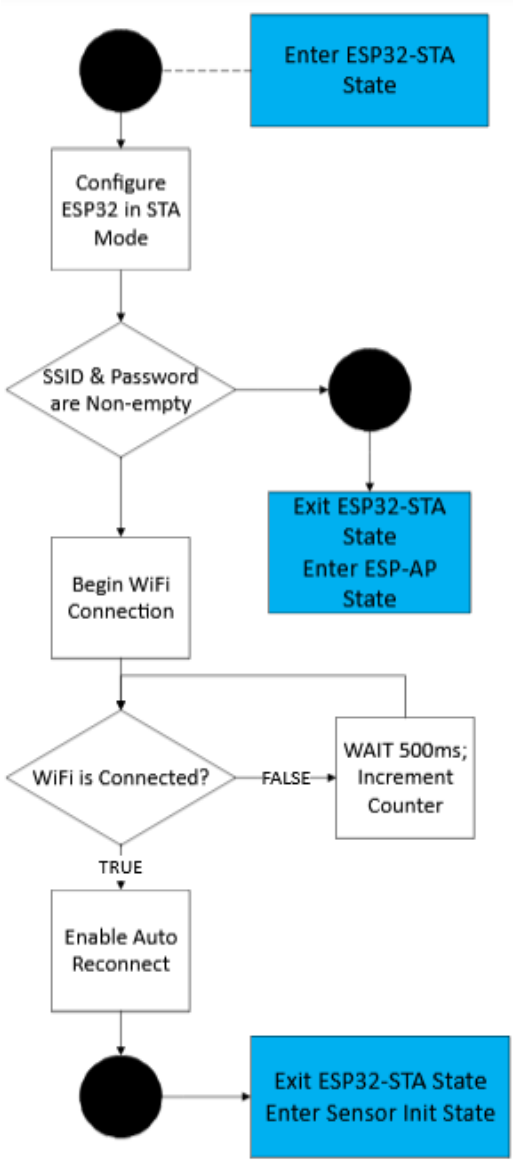


Figure 7.2.3 Activity Diagram for ESP32-STA State.

In the station initialization state, the ESP32 disconnects from its access point mode configuration. The ESP32 configures itself as a station and attempts to connect to the user's Wi-Fi network for a predetermined maximum limit of attempts. Between each attempt, the microcontroller enters a short (< 1 s) delay. If the ESP32 is not able to connect to the Wi-Fi network within the allotted number of attempts, the ESP32 outputs a serial message indicating this failure. Then, it exits the ESP32-STA Init state and re-enters the ESP32-AP state, where the user will need to re-submit or configure its Wi-Fi network or use another network. If the ESP32 microcontroller is able to connect to the user's Wi-Fi network, it exits the station initialization state and enters the Sensor Initialization state.

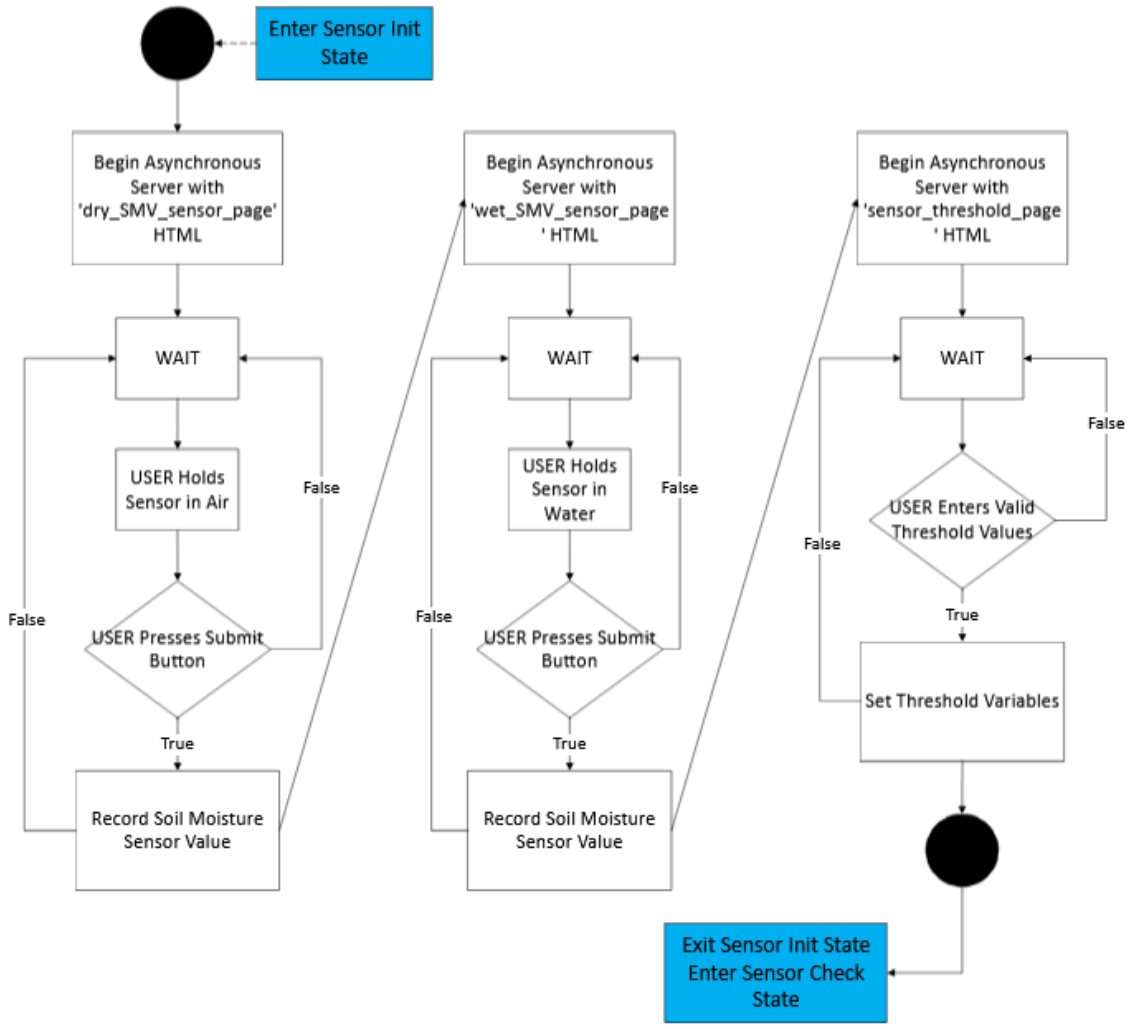


Figure 7.2.4 Activity Diagram for Sensor Init State.

Some sensors must be configured and specific sensor threshold values must be set before the microcontroller enters OPERATION mode. The soil moisture sensor must be configured by setting the soil moisture values when the sensor is completely dry and completely wet. The ESP32 establishes an asynchronous web server to host HTML web pages. One web page corresponds to setting the soil moisture value when dry and the other corresponds to setting the value when the sensor is wet. The first web page asks the user to hang the sensor in the air. The second web page asks the user to submerge the sensor in water. Both web pages have a submit button. When this button is clicked by the user, the ESP32 software calls a function which reads the value from the soil moisture sensor and sets the corresponding variables. A third web page is sent by the ESP32, which allows the user to specify factors such as the desired minimum soil moisture threshold and maximum length of sunlight exposure. After these values are set and correspond to valid input criteria, the ESP32 exits the SETUP mode and enters the OPERATION mode.

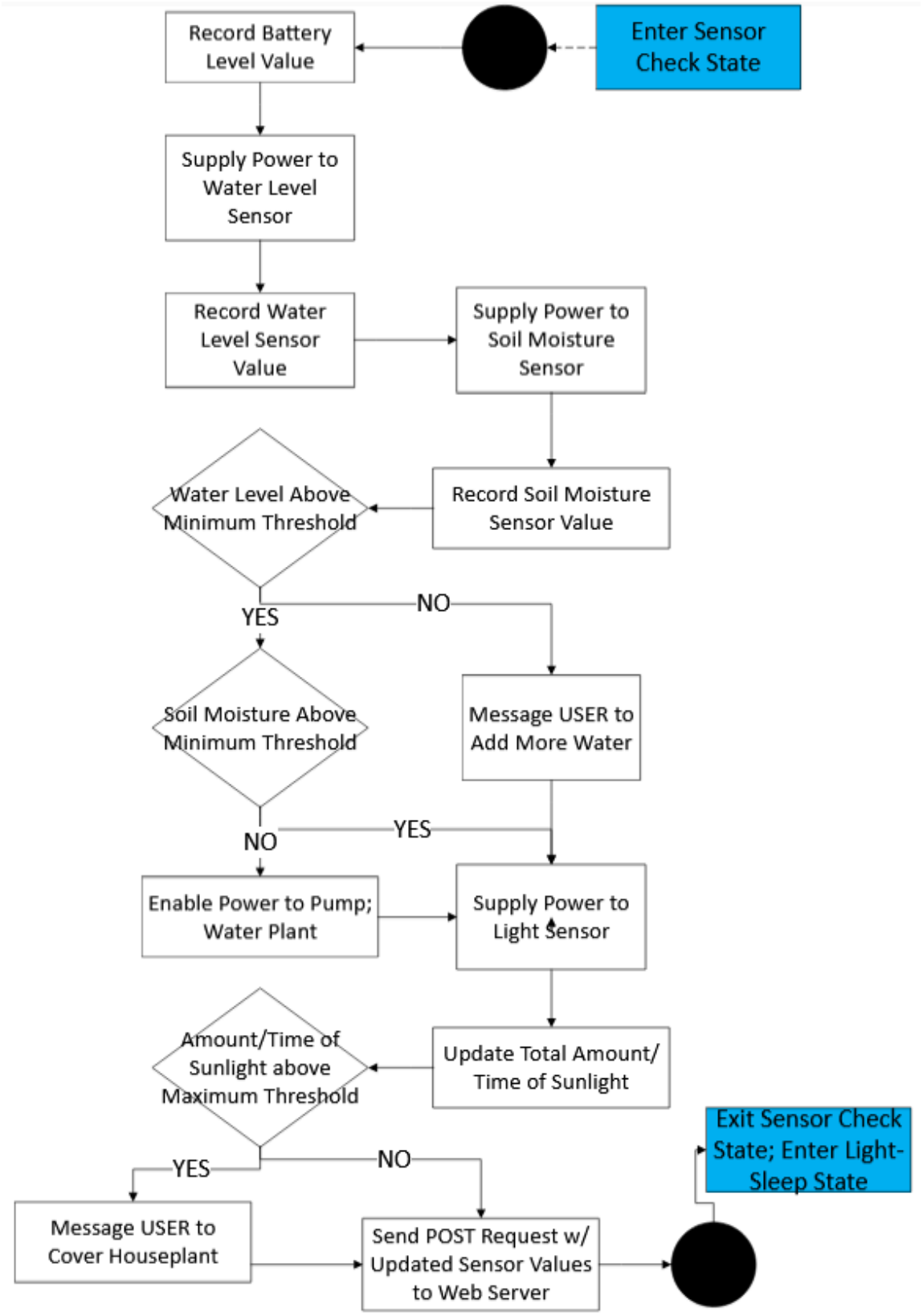


Figure 7.2.5 Activity Diagram for Sensor Check State.

The first state in the OPERATION mode is the Sensor Check state. First the battery level value is recorded. Next, power is supplied to the water level sensor. The water level is recorded and power to this sensor is disabled. Next, power is supplied to the soil moisture sensor, the soil moisture value is recorded, and power to this sensor is disabled. If the water level is below the minimum threshold, then the ESP32 sends a message to the user, likely through a PUT request to the server, that tells the user to add water. No water is supplied to the plant from the pump. However, if the water level is above the minimum threshold and the soil moisture is below its minimum threshold value, then power is supplied to the pump for a given amount of time. If the soil moisture is above its threshold, then no water is supplied to the plant from the pump. Next, power is supplied to the light sensor, the amount or time of sunlight is updated, and the power is disabled. If the sunlight time/amount is above the maximum threshold value, then a message is sent to the user telling them the plant has received an adequate amount of sunlight and to cover the plant or remove it from the sun. Lastly, a POST request is always sent to the web server to update sensor values so that the user can monitor the health of the plant and state of the system.

7.3 Definition of important data classes and structs

7.3.1 Plants

The plant class represents a structure that holds important information regarding a specific type of plant. We currently plan to have two tiers of plant types, one which is global and one which is per user. The fields will be defined as follows:

Table 7.3.1.1 Plant Class Fields.

Field	Type	Present In	Description
name	string	Web application	The human-readable name of the plant. Examples include “Rose” or “Prickly Pear Cactus”. This will be the name that is displayed to the user only on the web application side.
uuid	128 bit integer	Web application	A unique method for identifying plant types. This value should be unique to the plant and should not be visible to the user. It is also not necessary to send this value to the embedded processor, since it does not need to know what the plant type is.

plant_image (url)	string	Web application	An image of the plant. This could be a graphic that was drawn or a real image of the plant. It serves to provide a visual representation of the plant to the user. The image will be represented by a url that is a link to the image, which can then be displayed.
parent_uuid	128 bit integer	Web application	The uuid of the parent plant that this plant object is based on. This is mainly useful for user-defined plants, as the user should be able to take an existing system or user level plant and create a new plant that modifies some of the information. When this occurs, the parent_uuid will point to the plant who's information was modified.
watering_timer	integer	Web application & embedded processor	The amount of time that should pass between watering sessions. This number can be defined in the web application, either by the user or the system itself, and then passed along to the embedded processor to set timers for certain interrupts related to collecting soil moisture and activating the water pump.
maximum_moisture_level	float	Web application & embedded processor	This is the maximum level of moisture that the plant can sustain in its soil. If this value is reached, the pot should no longer pump water to the plant.

minimum_moisture_level	float	Web application & embedded processor	This is the minimum moisture level that can sustain the plant. Dropping below this moisture level will means that the pot must attempt to water the plant, as long as enough water is available as determined by the pot's water level.

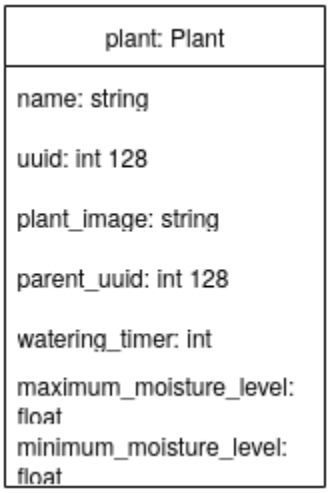


Figure 7.3.1.2 Plant Data Structure Diagram.

With regard to the plant type, most of the information that needs to be stored is purely for the benefit of the user. System-level plant types serve as 'canonical' plant types that custom user level plants can descend from, and the type as a whole serves as a way to show the user which plant is connected to a given pot.

7.3.2 Pots

The pot type is a wrapper of the plant type whose main goal is to serve as a connection between the server of the web application and the pot, although it will also hold information pertinent to the upkeep of the pot that cannot be categorized under the Plant object, such as battery percentage and water level. Compared to the plant, which is heavily user facing with very little information pertinent to the embedded system, the pot contains much more data that is shared between the web application and the embedded system.

Table 7.3.2.1 Pot Class Fields.

Field	Type	Present in	Description

pot_ip_ap	IPAddress	Web application & embedded processor	The IP address of the pot when the ESP32 is configured as an access point. This is used by the server so it knows which address to send information to, and it can also be used to identify which pot is communicating with the webserver when the pot sends data.
pot_ip_sta	IPAddress	Web application & embedded processor	The IP address of the pot when the ESP32 is configured as an station. Used for the same purpose as pot_ip_ap.
battery_percentage	float	Web application & embedded processor	This is the current battery percentage of the pot. This will be reported by the pot to the web application, where it will be shown to the user. It also allows the embedded system to monitor its battery level and implement solutions to lower power as necessary or remove solutions as the battery charge increases.
water_level	integer	Web application & embedded processor	This will report the level of water in the pot. This will be reported to the web application at regular intervals or the request of the server. In the server, the value will be compared to a threshold to determine whether or not to alert the user to low water levels.

current_moisture_level	float	Web application & embedded processor	This will report the current moisture level in the pot. This should be shown to the user to allow them to monitor the health of the plant, and be used in the embedded application in the pot for the purposes of maintaining a balanced moisture level by comparing this value to the maximum and minimum moisture values.
plant	Plant	Web application	This is the plant object corresponding to this pot. This allows the web application to tie a specific plant's information to a specific pot, and can be used to send plant specific information such as water_timing to the embedded processor in the pot.
connection	struct	Embedded processor	This is information used by the pot to connect to the wifi network. This data should be fairly persistent in the lifespan of the program, only changing when the user re-enters pairing mode and connects to a new wifi network.

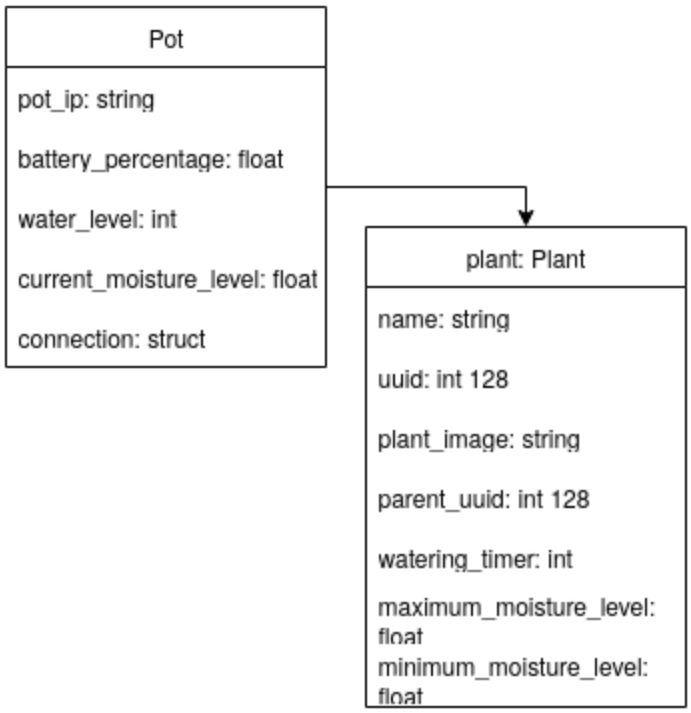


Figure 7.3.2.1 Pot Data Structure Diagram.

7.3.3 Users

Users are a special type of class designed to hold user specific data. This data is only present on the web application, and is used for authentication and determining what the user sees upon entering the application, as well as what they can interact with. The purpose of user data is for ensuring that users can only interact with pots that they own, and can only see information from these pots, rather than just allowing anyone to see any pot. Authentication is also necessary to prevent any private user information from being leaked to anyone besides the user.

Table 7.3.3.1 User Class Fields.

Field	Type	Present In	Description
username	string	Web application	The username is used as part of the authentication process. It serves to provide a human readable unique identifier that the end user can provide as part of the authentication process.

password	string	Web application	The password is another part of the authentication process. It is used to validate that the person trying to access an account with a given username should be accessing said account by confirming that they know the account's password as well.
uuid	128 bit integer	Web application	Unique identifier for users on the backend. This field is mainly for identifying the user in backend logic, as it is much safer than using the string username for identification.
custom_plants	List	Web application	This is a list of custom user-level plants, separate from the system-level canonical plants. This allows users to easily add new pots for plants that they have already provided information for by selecting from the list of existing custom plants.
pots	List	Web application	This is a list of pots attached to a given user. The user will be able to monitor information regarding the pots in this list as well as any plants that exist under the pot.

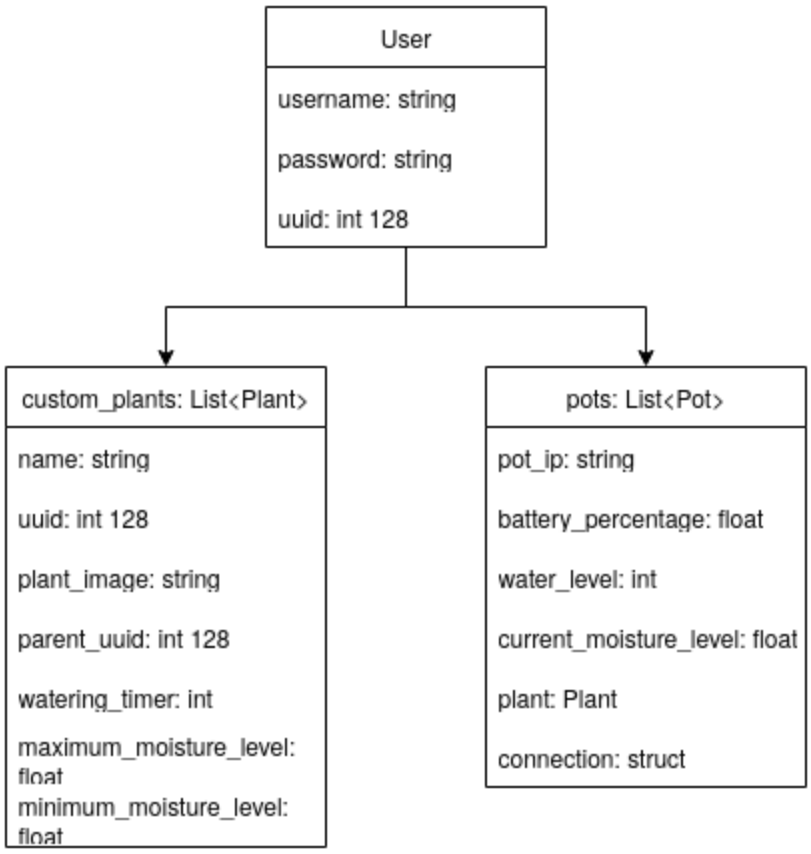


Figure 7.3.3.2 User Data Structure Diagram.

7.4 Reading data from sensors

One of the main jobs that will be performed by the embedded application is reading and processing data from the numerous sensors that we have positioned around the pot. For the purpose of reading data from sensors, we define a common SensorReader class which contains default methods for reading data from sensors and creating an object which encompasses the data received. This class should be a templated class, which is ideal for allowing different SensorReader objects to define their own implementations for reading data from a given sensor and allowing them to format that data into an object that can then be used. This class should contain necessary configuration data, a reference to the sensor_data object, and it should contain a common Read and ReadImplementation function.

The benefits of defining a common SensorReader class mainly come in the form of allowing us to easily and dynamically add more sensors to our design. This is because most of the work is already accomplished by the common class, meaning that adding a new sensor once the class is defined is as simple as defining a Configuration object and creating a ReadImplementation definition for the given sensor. Then, we can define a global list of sensors in our program that we iterate through whenever we need to read sensor data, before handling the resulting sensor data object that we receive.

The SensorData object should be an object defined in static program memory that defines the necessary fields that need to be populated by each sensor. This object should be shared by all sensors, since none of these sensors will populate or overwrite the same fields as another. Using a single SensorData object makes it easier to work with the data, especially for the time when we eventually encode this data into a byte array to be sent to our web server over the wire. This approach also makes it easy to define which fields are populated and with which data said fields are populated with.

Table 7.4.1 SensorReader Methods.

Method	Inputs	Returns	Visibility	Purpose
Read()	none	SensorData	public	The read method is a generic method that will be used to request that information be read from the sensor. The purpose of the read method is to handle the inner function pointer that defines how data is read from the sensor, as well as common setup information for all inner read functions.
ReadImplementation()	SensorData reference	none	private	This method will populate the reference to the SensorData object with data from the sensor. This should be seperate from the common Read() method to allow for common logic to be implemented in that function, while more specialized logic is saved for this function. This function should be defined on a per-object basis.

SensorReader()	SensorConfig, SensorData reference	SensorReader	public	Default Constructor for SensorReader object. Should take in a Config which will set appropriate parameters to be passed on to the sensor configuration register if available or used by the SensorReader itself. It should also contain the function pointer to the internal read function.

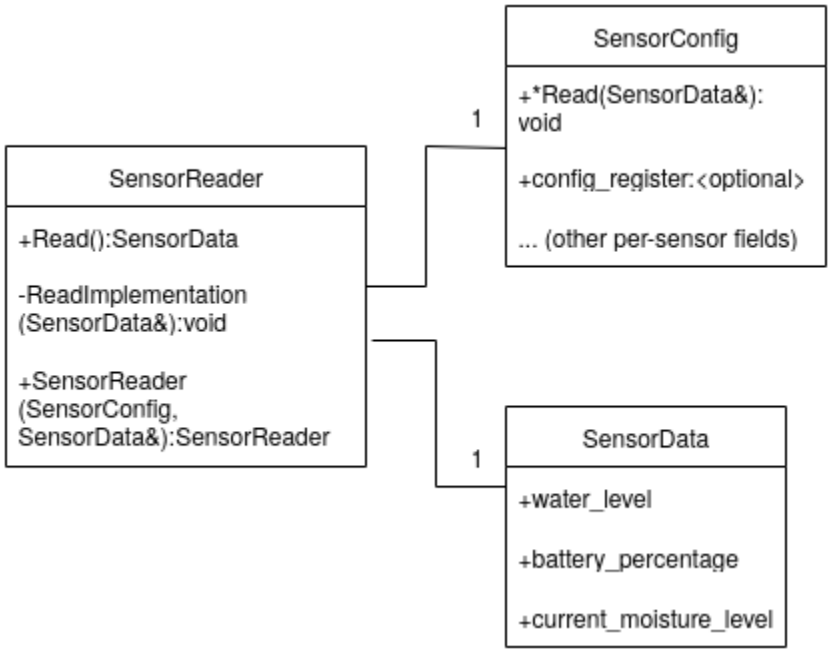


Figure 7.4.2 SensorReader Class Diagram

7.5 Memory Management

For our embedded application, the general guidance for memory management is to prefer the use of static memory both where possible and as much as possible. To accomplish this goal, we will use a statically allocated memory pool, which allows for our program to allocate memory as needed from the stack, while also ensuring that we can stay within a strict memory limit that is necessary when programming for an embedded system with limited hardware resources.

As stated, using a static memory pool has many benefits. Using a static memory pool allows us to use the stack to store data that we have created, rather than the heap. This

allows for faster data accesses in general, as stack data is more commonly stored in cache, which improves latency and allows our embedded application to respond to requests faster. The other benefit of using static data is that it enforces a stricter memory management style than is normally available in C++. Using static memory forces the allocation of a specific amount of memory at compile time that cannot be changed after the process has begun running. This strict memory requirement helps to enforce good programming practices regarding memory management and ensuring that memory which is used by the program is confined to the area occupied by the memory pool and that the programmer takes care to avoid using too much memory or creating a memory leak.

For the implementation of the memory pool, the region of memory should be declared as a contiguous section of non-typed data (such as a char array) with a fixed size at compile time. This region of non-typed memory should be mapped through a data structure (stack or queue) which lists free memory block addresses. When the program requests memory, it must request through the memory pool, which will then use the underlying data structure to find n free blocks, where n refers to the number of data blocks needed by the object, ensures that there is a contiguous region of memory, and creates an object mapped to that region of memory via casting. When the region of memory is no longer needed, the addresses of the memory are pushed back to the underlying data structure, which allows for the memory to be reused for future data.

This algorithm allows for a simple but effective method of memory management that allows for both allocating and freeing memory from a static memory pool, which we make use of in order to manage tight memory restrictions in an embedded processor.

7.6 Communications

7.6.1 Transferring Data

For the purposes of sending data between our microprocessor and our server, we first must define a common format in which data will be sent. For this purpose, we will use JSON, also known as JavaScript Object Notation as a common format for sending data. JSON provides a simple representation of data as key value pairs, which is useful for both interfacing with a database on the server side or converting the data in a native c struct in the microprocessor program. JSON is also very easy to manipulate in both codebases, as many C libraries exist for creating and populating JSON objects from C structs, and JavaScript similarly contains robust functionality for working with JSON data. When sending this data, we will make use of HTTP functionality available both in JavaScript through ExpressJS as well as in C libraries provided by the ESP32. For the purposes of our application, we will mainly make use of HTTP GET and HTTP POST requests, which will be sent by both the server, ESP32, and the client, although other HTTP request methods will be used as necessary.

7.6.2 HTTP Methods

7.6.2.1 Microprocessor

The microprocessor should only be capable of sending HTTP POST and HTTP GET requests. Similarly, the microprocessor should only be able to handle incoming HTTP POST and HTTP GET requests from the server.

In terms of sending requests, the microprocessor should mainly send POST requests to the server. These requests should be used to send sensor data from the microprocessor to the remote server. These POST messages will be configured to be sent by the microprocessor after regular readings at the set time interval, but can also be sent if an urgent notification, such as a low water level notification, needs to be sent. HTTP GET methods will rarely be sent by the microprocessor to the server, however, some defined functionality relies on the use of such methods. Mainly, the state restoration after power loss relies on using the HTTP GET method to retrieve necessary plant information from the remote server.

In terms of receiving requests, the microprocessor will receive HTTP POST and HTTP GET requests from the server. The main functionality of HTTP POST requests will be in sending plant information to the microprocessor for the purposes of initialization of the system. For this, the microprocessor must be programmed to handle such requests. The microprocessor must also handle GET requests where the server requests updated plant information.

7.6.2.2 Server

The server should be capable of sending HTTP POST and HTTP GET requests to the ESP32. The server should not send any requests to the client. The server should be capable of processing HTTP POST and HTTP GET requests from the microprocessor, and should be capable of processing HTTP POST, HTTP PUT, HTTP GET, and HTTP DELETE requests from the client.

For requests that the server sends to the ESP32, the server should mainly send HTTP GET requests. These requests should be sent when the server wishes to update its current information of the pot's status. The ESP32 should, in response, send updated information read from the sensors connected to it. The server should also send HTTP POST requests, mainly the one containing plant information that must be sent to initialize the pot.

For requests that the server sends to the client, this should never be done. This is because traditional HTTP defines only client to server communication, where the client makes a request that is answered by the server. This should not be done in reverse, as it is non-standard and could introduce many dangerous vulnerabilities or bugs.

For requests that the server receives from the ESP32, these should mainly be HTTP POST and HTTP GET requests. The server should receive POST requests when the ESP32 attempts to send updated sensor data. It should use this to update existing data and prepare notifications if necessary. It should receive GET requests in situations such as the

ESP32 requesting plant info after a power loss. It should provide plant information to the microprocessor in such situations.

For requests that the server receives from the ESP32, these should consist of HTTP POST, HTTP PUT, HTTP GET, and HTTP DELETE requests. HTTP POST requests are necessary for populating database entries, such as user information or plant information. HTTP PUT requests will be used for updating existing entries such as overwriting a pot's plant data with new data. HTTP GET will be used for retrieving information from the server's database regarding information from the sensors, as well as for retrieving any information regarding notifications that need to be displayed to the client. HTTP DELETE requests will be used for removing pots from the user or deleting user accounts in the event that the user requests such an action.

7.6.2.3 Client
The client should be capable of sending HTTP POST, HTTP PUT, HTTP GET, and HTTP DELETE requests to the server. It should not be capable of processing any HTTP requests.

For requests that the client sends, it should send HTTP POST requests in the event that the client needs to send general data, such as user credentials or new plant information to the server. It should send HTTP PUT requests when it needs to update existing data, such as changing an existing user's password, or updating an existing plant or pot's information. The client should send HTTP GET requests when it needs to retrieve information from the server, such as plant information, but it should also regularly poll the server for notification updates, in case any notifications need to be sent to the client from the server.

To adhere to a traditional client server model of network communication, the client should not be capable of receiving or responding to any requests from the server.

7.7 Web App Architecture

7.7.1 Client
The architecture of the client will mainly be built using ReactJS, part of the PERN stack that will be used to build the web application. Through React, we can define different UI elements as components, where each component will be responsible for executing one function. For styling elements in the web application, css will be used.

The general flow of the client will start with the user entering the Login / SignUp page, which will request account information. If the user is logging in, their account credentials will be sent to the server, which will either validate or reject their credentials after checking against the database. After this, they will navigate to the web app dashboard. This dashboard view will show notifications, as well as a quick look at any plant pots that the user may have. The main navigation of the app will take place through the sidebar, which will feature available categories for user settings, returning the homepage, and going to the plant list view.

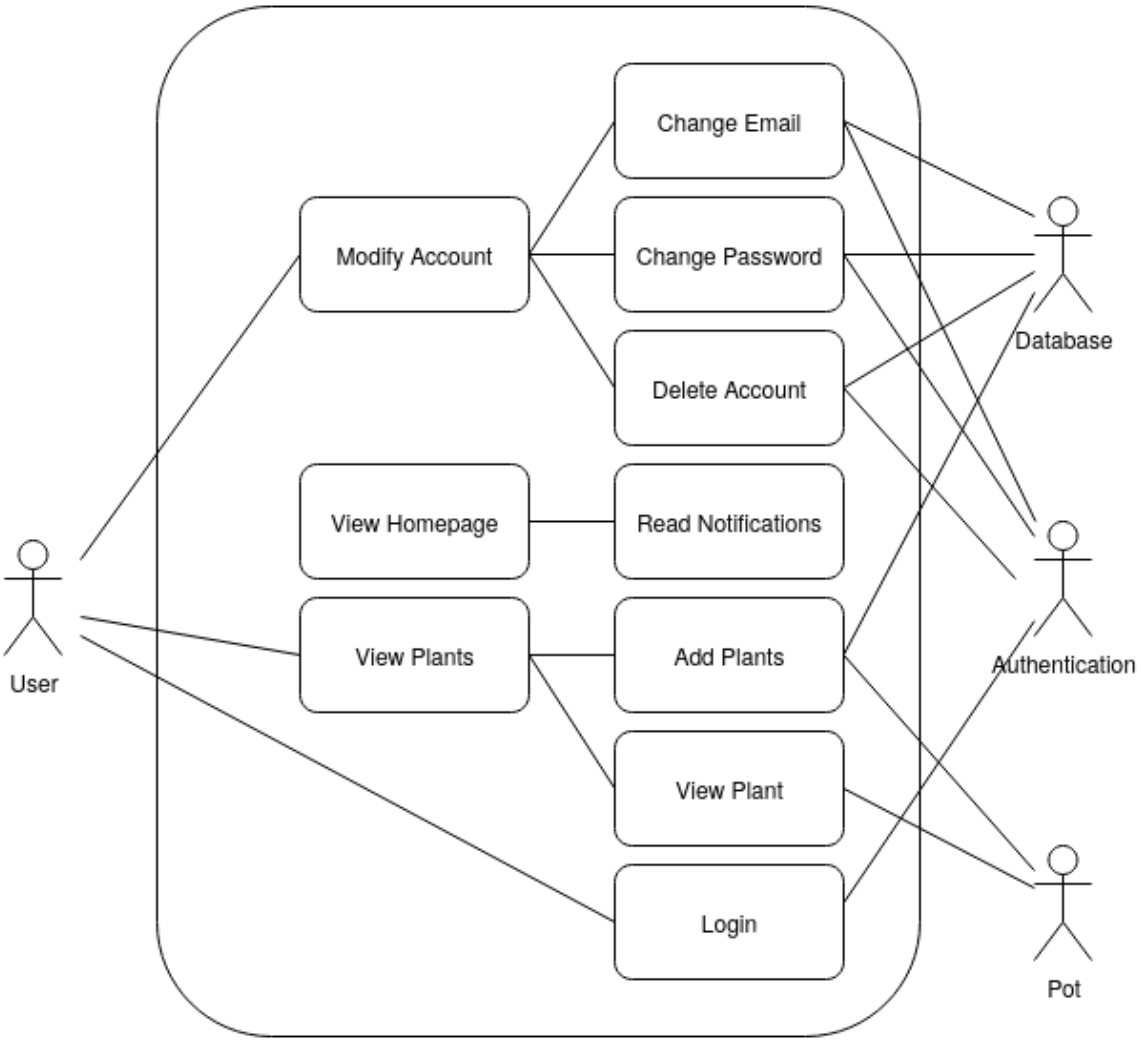


Figure 7.7.1.1 Use Case Diagram for Web Server Client

The plant list view will give an overview of all the user's plant pots, with a more detailed status than what is available on the homepage, as well as the ability to add a new plant. Clicking on any of these plants will maximize the view, and also send a request to the server to update the sensor information. While waiting on sensor information, the client will let the user know that it is attempting to update the current information. The maximized view gives a full overview, with more detailed sensor readouts, as well as remotely available controls available as buttons, such as the ability to force a refresh in data readings.
Adding a new plant should introduce its own dialogue. Here, the user can select from a list of plants, given by the server.

The user settings view will give an option for modifying general user settings. This includes changing the username, email address, or password. This view should also allow the user to delete their account, should they choose.

In order to promote a consistent application UI and UX, a set of design guidelines must be followed. These guidelines, which will be the Human Interface Guidelines of the application, should define the overall style and design approach that is used when designing user-facing elements of the application. The Human Interface Guidelines of the application are as follows:

1. Adaptability: The application should feature an interface that can easily adapt to fit a wide variety of screen sizes. An example would be the ability of the sidebar to be compressed into a single button, which, when activated, can bring up the sidebar. This is useful to allow for vertical screens without much space, such as smartphones, to view the content without the sidebar taking up an inordinate amount of the screen.
2. Content-centric view: The main focus of the application should be the content that the user wants to view. To this end, the use of features that can obfuscate the view of content should be reduced as much as possible. When viewing a page that was navigated to from the sidebar, all content should be displayed without the need for tabs or scrolling if possible. If one of these features is required, then scrolling should be preferred to tabs in order to prevent the obfuscation or hiding of any content.
3. Readability: In order to improve the readability of the application, the color palette of the application should be limited to a small set of colors that allow for good contrast between different elements for all users. Each element should feature a single color, avoiding design patterns such as gradients, in order to enhance readability. Each element should also be a distinctly different color from the element next to it. For example, white text should not appear in front of a light gray background.
4. Accessibility: In order to enhance accessibility, all elements in the application should be accessible using only one tool. That is, the application should be navigable using only a keyboard, or only a mouse, or etc. This helps to make the application easier to navigate for people with physical disabilities. The application should also mandate the use of Alt Text with images, which helps to allow people with vision-related disabilities to still be able to understand the context of the image that is being displayed.

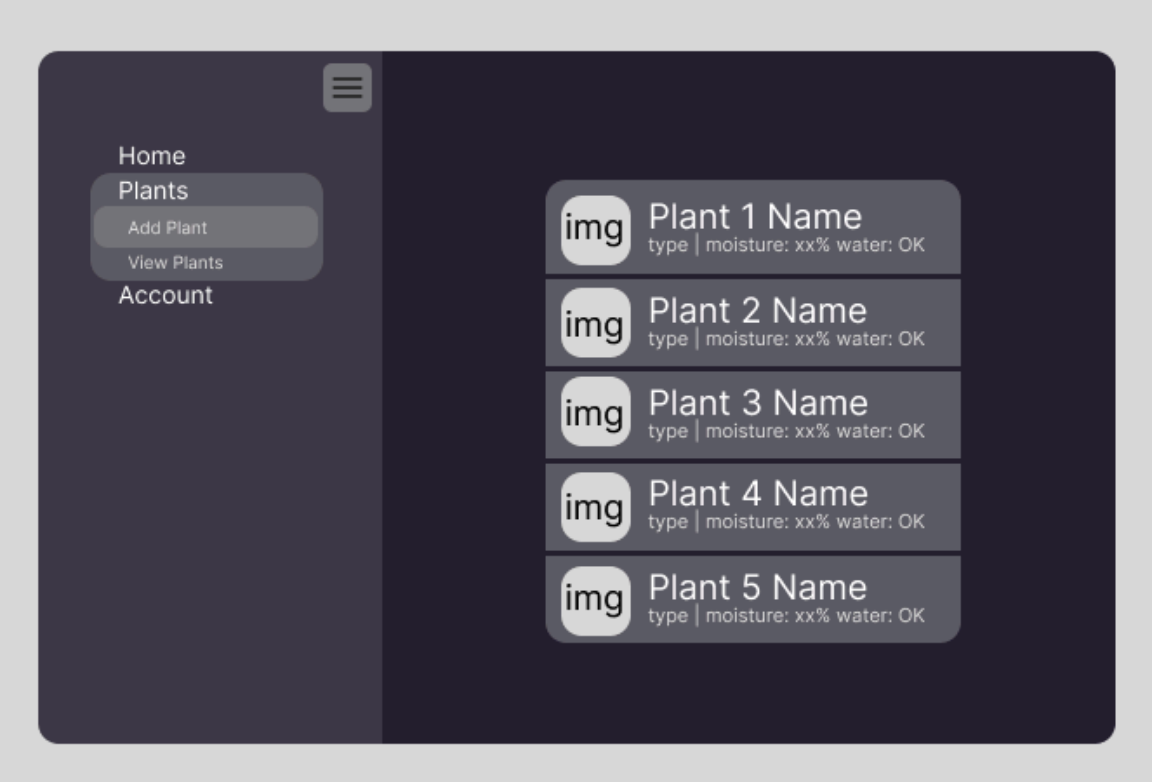


Figure 7.7.1.2 Plant View Mock-Up

7.7.2 Server

The architecture of the server has two sides in that the server most both communicate with the plant pot as well as with the client. To accomplish this goal, the server's main goal must be to process and respond to incoming HTTP requests from both the client and pots. This will be accomplished via the use of an API, or Application Programming Interface. This API will take in requests sent to it and process them using functions available only to the server to retrieve or store necessary data. In order to properly handle both the client and the pot, two distinct APIs must be implemented, one to handle requests from the client, and another to handle requests from the pot.

Each API call will be made to the server using an HTTP request. The server, will use an ExpressJS backend to handle such requests. Upon receiving one, the backend will associate the request with the necessary functions that need to be executed. These functions will mainly consist of calls to the database. From here, the functions will either store or retrieve data from the database, which will be sent back to the requesting machine in the HTTP response from the server.

The server will use Postgres as the database software. The use of Postgres allows for the server to easily query and modify the database as necessary when performing function calls through integration between Postgres and JavaScript. Within the database, most data should be tied to a given user. This user will have different data categorized as belonging to them, that can be retrieved when the client requests that data and the client has been authenticated to access that specific user data.

Authentication should be accomplished using some form of Oauth service, such as Google Authentication, to authenticate a user. This allows for a more secure authentication system for user credentials without needing to implement a custom authentication system. For plant pot authentication, the server should verify that the IP belongs to a given plant pot before sending information to the pot.

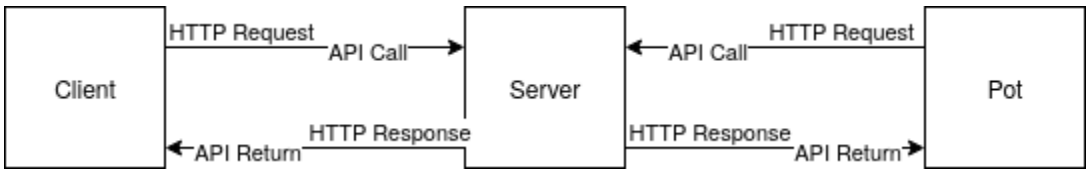


Figure 7.7.2.1 Server API Message Flow

Chapter 8 - System Fabrication / Prototype Construction

8.1 Housing Enclosure

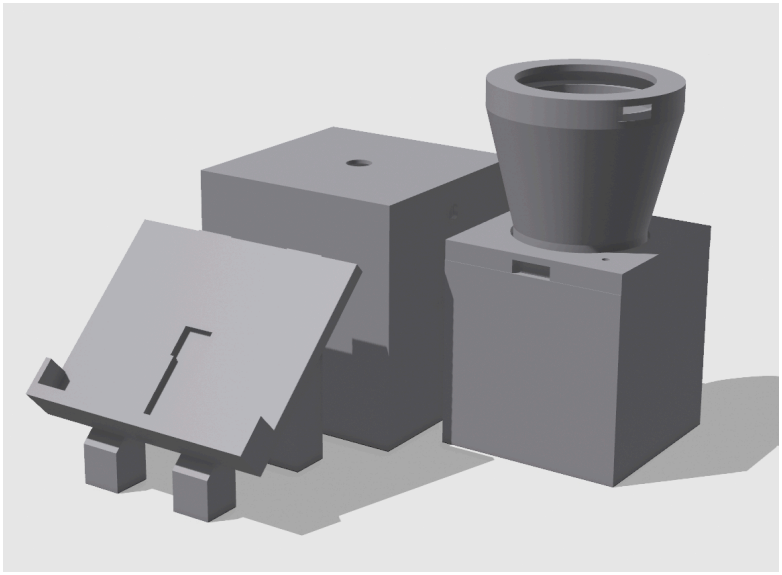


Figure 8.1.1 3D Rendering of Enclosure.

Made to ensure durability, safety, and environmental protection for the project, our enclosure incorporates a custom 3d printed design made specifically to ensure that the various components of C.H.A.N.O.S. work together. The enclosure serves multiple purposes which can be roughly divided into 3 parts, protection for the components, storing of water, and housing for the plant that is being taken care of. During the design process, considerations such as component placement, cable routing, and ease of assembly were taken into account. The overall form, which has remained the same, can be observed in section 2.7 “Prototype Illustration/ Blueprint”.

8.2 PCB Fabrication

The printed circuit boards we make are the foundation for integrating and interconnecting the various electronic components used in the C.H.A.N.O.S. project. This process involves putting the electrical schematic into a layout that defines the physical placement of components and the routing of electrical signals between them. Important considerations during PCB design include signal integrity, power distribution, thermal management, and more. By designing a custom PCB, the project achieves a more compact, reliable, and professional solution compared to breadboard alternatives. The following section outlines the various PCB designs made, the tools used to make them, and the overall placements of the board.

8.2.1 Light Sensor

The BH1750 ambient light sensor was integrated into a dedicated PCB to allow for accurate and stable lux readings in the C.H.A.N.O.S. system. The design includes the BH1750FVI chip, pullup resistors for the I2C communication lines (SCL and SDA), and a decoupling capacitor to reduce voltage noise and improve sensor stability. A 4 pin header was added to the PCB to provide clear and accessible connections for VCC, GND, SCL, and SDA enabling easy interfacing with the main ESP32 board (ADDR was connected to ground).. The PCB was designed as a modular unit that could be easily mounted and connected to the main board using male header pins and soldered jumper wires, supporting both mechanical and electrical integration. This modular approach also allows for easy replacement or repositioning of the sensor depending on application needs. Its placement will be on the outside of the shell in order to get clear and unobstructed readings from the sun. Below we can see one image of all the traces and another as a 2D rendering of what the PCB board could physically look like.

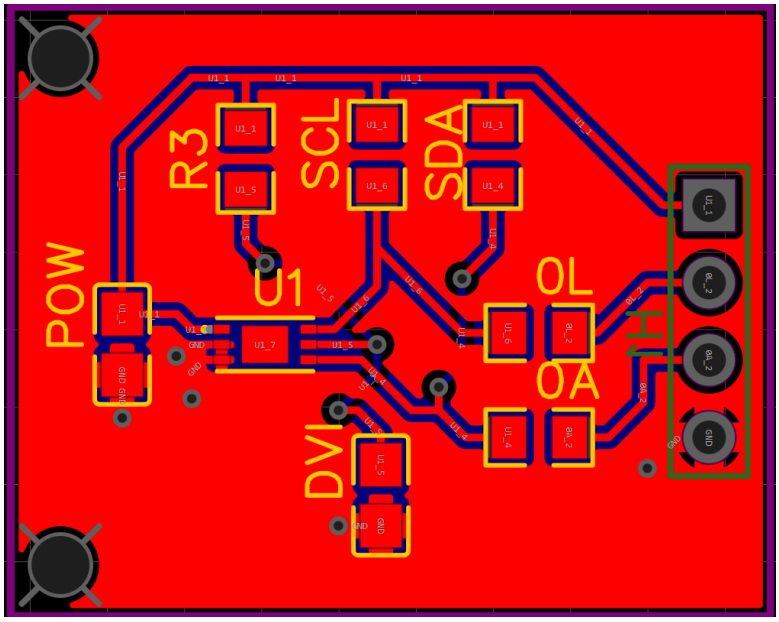


Figure 8.2.1.1 Light Sensor PCB

8.2.2 Horizontal Float Sensor

The horizontal float sensor was incorporated into a custom PCB to detect water level changes within the plant watering system. This sensor functions as a simple switch that closes or opens depending on the water level, making it ideal for monitoring when a reservoir is low. The PCB design features two kinds of male headers. The first is a two pin header which connects the float sensor to the PCB board; the second is a three pin header for the 3.3 volts, GPIO pin, and ground connections.

For better results in the sensor's operation, the PCB includes a pull up resistor and a capacitor, improving signal stability and reducing false triggers. The compact design ensures that the PCB can be positioned close to the water reservoir, while the routing ensures clean signal transmission back to the main control board. The simplicity of the float sensor circuit allowed for straightforward integration, ensuring reliability and ease of maintenance.

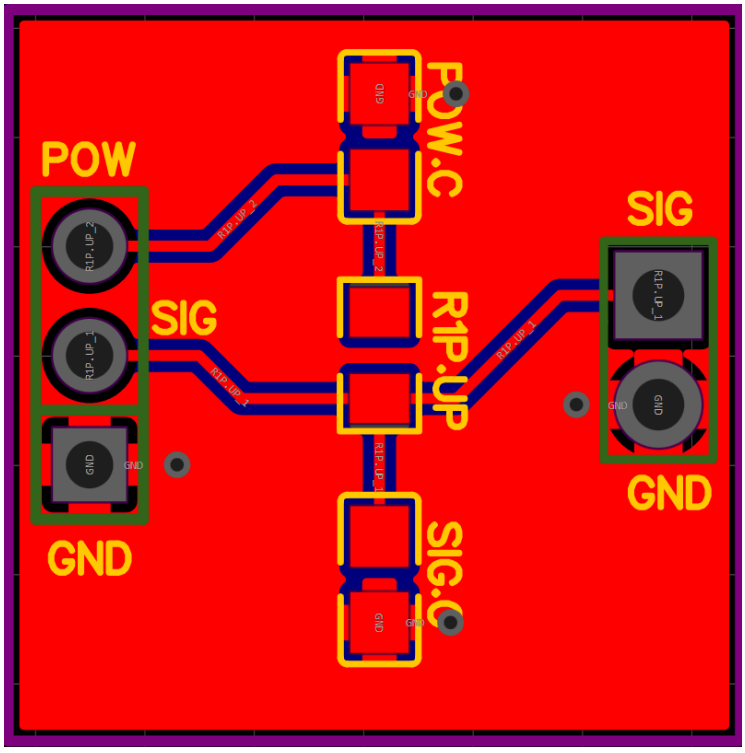
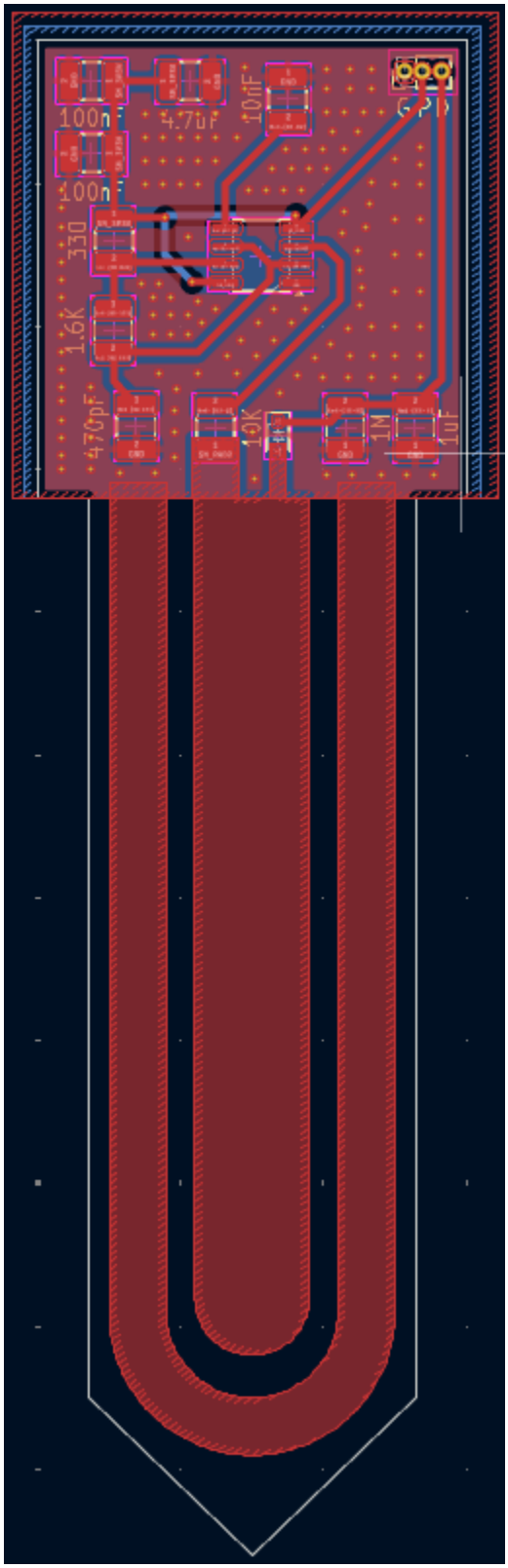


Figure 8.2.2.1 Water Level Sensor PCB

8.2.3 Soil Moisture Sensor



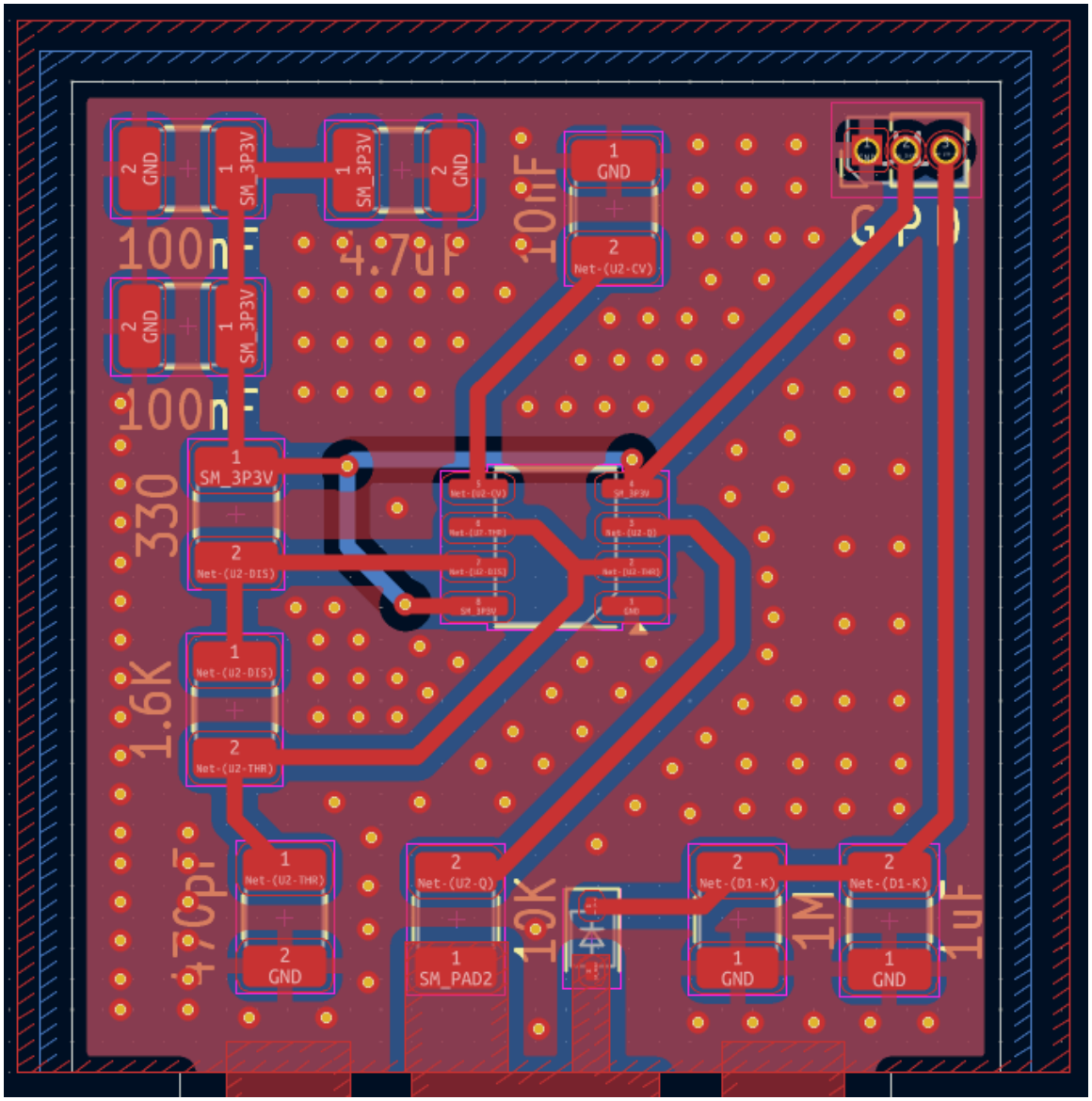


Figure 8.2.3.1 Soil Moisture Sensor PCB Design.

The PCB design in figure 8.2.3.1 is the design for the soil moisture sensor printed circuit board. The TLC555CDR IC was placed in the middle of the board in order to make soldering easier. The 3-pin through-hole jumper connector was placed in the top right to avoid the wires restraining the position of the sensor when placed into a pot. Only one signal was routed between layers using vias, the 3.3 VDC signal from the ESP32 GPIO pin. Vias were used to increase stability and thermal conduction due to the need for high precision and the fact that the sensor will likely be in line with the sun (increased heat). The decoupling capacitors for the 3.3 VDC power signal were placed near the top of the board due to close proximity with the pins of the TLC555CDR IC. The components for the diode rectification and RC circuit near the output of the sensor were placed near the bottom of the sensor for close proximity with the capacitive pads. The capacitive pads were designed using GND or signal planes with dimensions approximately equal to the

pad dimensions of the commercial, off-the-shelf sensor. The trace width of 20 mil was selected after receiving feedback from Professor Weeks.

8.2.4 MCU / ESP32

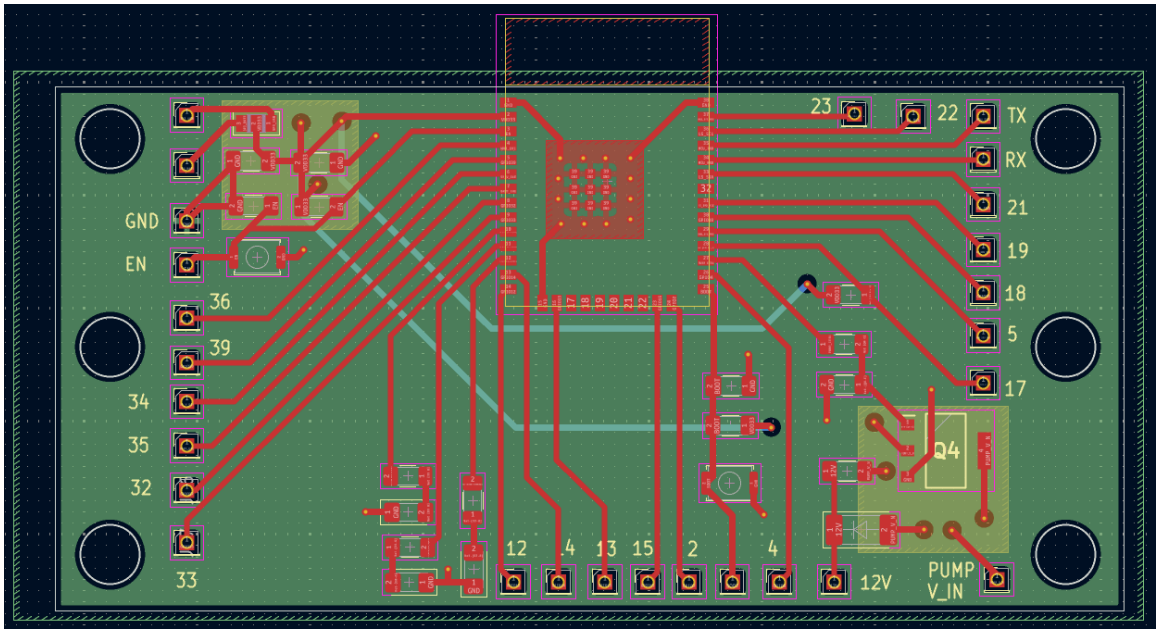


Figure 8.2.4.1 MCU PCB Design.

Figure 8.2.3.1 displays the PCB design for the ESP32 microcontroller and water pump control circuit. The remaining peripherals will be connected to their corresponding microcontroller pins via wires soldered to through-hole jumper connections. This decision was made to increase modularity, decrease testing and debugging latency, and because most of the peripherals must face the sun or be closer to the plant.

The MCU PCB has four copper layers. The top layer consists of most of the signal and power traces. The second layer is a ground layer. The third layer is used to insert padded zones for the power signals. The bottom layer is used to route 3.3V signal traces for pull-up resistors.

The ESP32 footprint was placed such that the RF antenna module hangs off of the board. This is in accordance with the Espressif hardware design guidelines, which establishes a keep-out zone for the RF module to avoid interference during wireless communication. The center of the PCB was chosen to provide room on all other sides for other component footprints. The passive components for the power supply and reset pin were placed together on the top left of the PCB. in close proximity with the corresponding ESP32 pins and since the reset pin should be pulled-up. Similarly, the boot pin is also pulled-up, but this required routing the trace on the bottom layer. A GPIO pin (GPIO17) was pulled-up in the event that a button would be added to trigger a reconfiguration of the system. However, it was realized after ordering the PCBs that this GPIO pin was not an RTC GPIO pin, meaning that it can not be used to trigger a wakeup from deep sleep mode. Subsequently, this pull-up resistor was not placed. A subcircuit for a 32kHz crystal

oscillator was initially placed at the bottom left of the PCB, but this was removed after feedback from Dr. Weeks. Three LEDs with series resistors were placed at the bottom of the PCB. In the bottom right of the PCB, the water pump control subcircuit was placed. Padded copper zones were placed on the third layer for the 3.3V and PUMP_V_IN (12V) power signals. These zones improve thermal conduction and reduce the voltage drop of the signals, which increases stability.

8.2.4 PSU

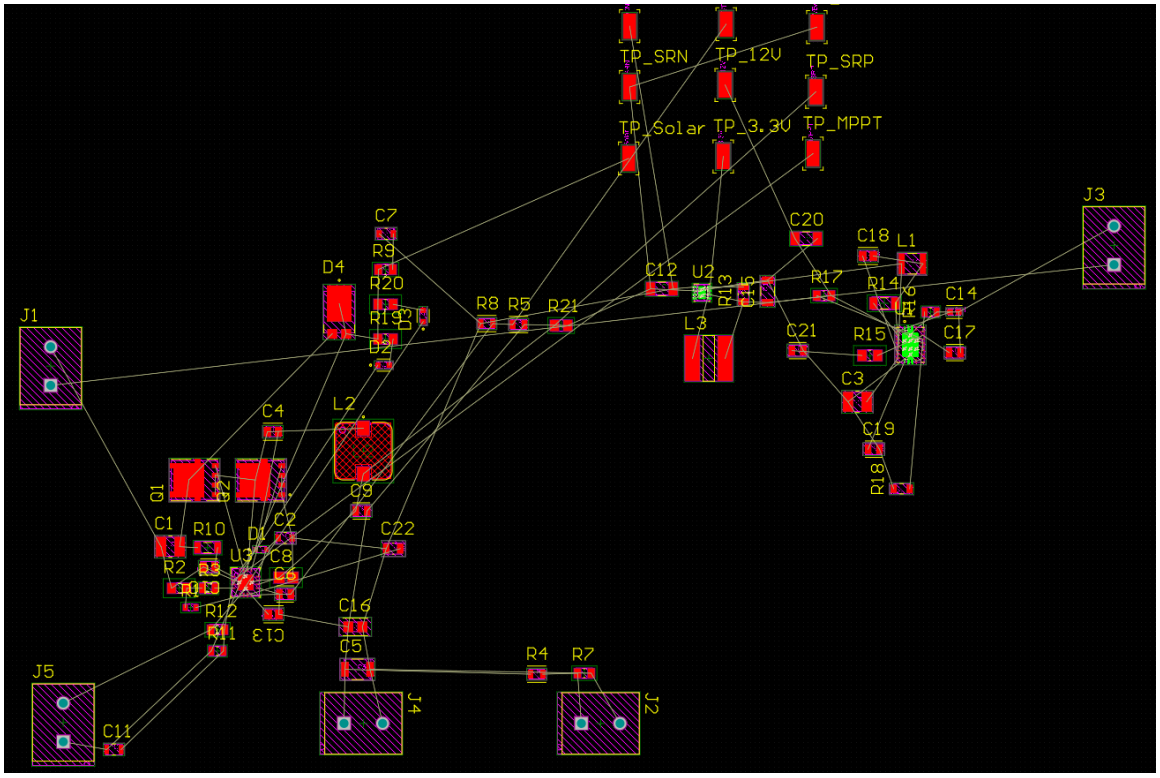


Figure 8.2.5.1 MCU PSU.

Figure 8.2.4.1 shows the preliminary layout for the power supply section of the system. The design is centered around a 2S Li-ion battery, which is charged using a solar panel through a dedicated MPPT-enabled charge controller IC. This IC handles battery charging, regulates current based on solar input, and provides status outputs used for basic monitoring. It remains active independently of the rest of the system to ensure consistent charging as long as sunlight is available.

A boost converter IC is used to generate a stable 12 V rail for the peristaltic pump. This section is electrically separated from the microcontroller and sensor circuits to avoid cross-interference when the pump turns on or off under load. The converter draws directly from the battery and is configured to deliver sufficient current for motor operation.

A buck converter IC is used to generate a 3.3 V rail for the ESP32 and associated sensors. This rail remains active during normal operation and supports all logic-level components. Power is routed in a way that prioritizes noise isolation and voltage stability.

Together, these three regulators form a modular power supply unit that supports solar charging, motor control, and low-voltage digital logic. Final routing and layout refinement will be needed to optimize thermal management, reduce noise coupling, and ensure proper pad spacing across all board sections.

Chapter 9 - System Testing and Evaluation

9.1 Breadboard testing

The original solar charging configuration used the Voltaic Systems P108, a compact monocrystalline panel rated for 8.61 W with a maximum power voltage (Vmp) of 17.82 V and current of 483 mA. On paper, this panel appeared to be a suitable match for our MPPT charge controller, the CN3791, which supports input voltages between 6 V and 17 V and is designed for 2S lithium-ion battery charging.

Initial bench testing focused on the charging subsystem only. With no downstream loads connected, the panel delivered its expected open-circuit voltage of approximately 21.9 V. But once the CN3791 was connected to a partially discharged 2S lithium-ion pack, the panel's voltage sagged under load, sometimes dropping as low as 7 V. This placed it below the CN3791's minimum input requirement, causing the controller to repeatedly fail to initiate or sustain charging. The MPPT function would briefly activate, then shut down as the panel voltage collapsed.

To isolate whether the issue was with the controller or the panel, we swapped the CN3791 for a BQ24650EVM-639 from Texas Instruments. The BQ24650 offered improved MPPT tuning and a more robust power stage, but the behavior remained the same. The panel voltage continued to collapse under the modest charging current of a depleted battery, confirming that the panel itself was the limiting factor in low-light or transient load conditions—not the controller.

Despite this, we opted to continue using the P108 panel due to its compact size, durable ETFE coating, and mechanical integration with the enclosure. The charge controller was upgraded to the BQ24650, which provided more consistent MPPT tracking and better fault recovery. While the panel does exhibit voltage collapse under specific conditions, careful tuning of the MPPT resistor divider and real-world solar positioning have helped minimize these effects in the field. In most lighting scenarios with a partially charged battery, the P108 now performs well enough to maintain battery life and sustain system uptime across full daily cycles.

Downstream voltage regulation was tested using off-the-shelf regulators during early integration. We used an XL6009 boost converter to generate the 12 V rail and an LM2596 buck converter to drop 7.4–8.4 V battery input to 5 V. The LM2596's slightly higher ripple and slower transient response were not an issue in this configuration due to the

onboard LDO on the ESP32 DevKit, which smoothed out supply noise and helped prevent brownouts during Wi-Fi activity. The XL6009, although not as efficient as the TPS61088 intended for the final board, proved capable of handling pump startup loads and maintaining consistent 12 V output.

These test results confirmed that the P108 panel, while limited in raw current output, can still operate successfully when paired with the right MPPT tuning and usage profile. The use of the XL6009 and LM2596 in testing gave us a stable baseline for validating system performance prior to final PCB integration and has informed the sizing and protection requirements for the power path in our custom board design.

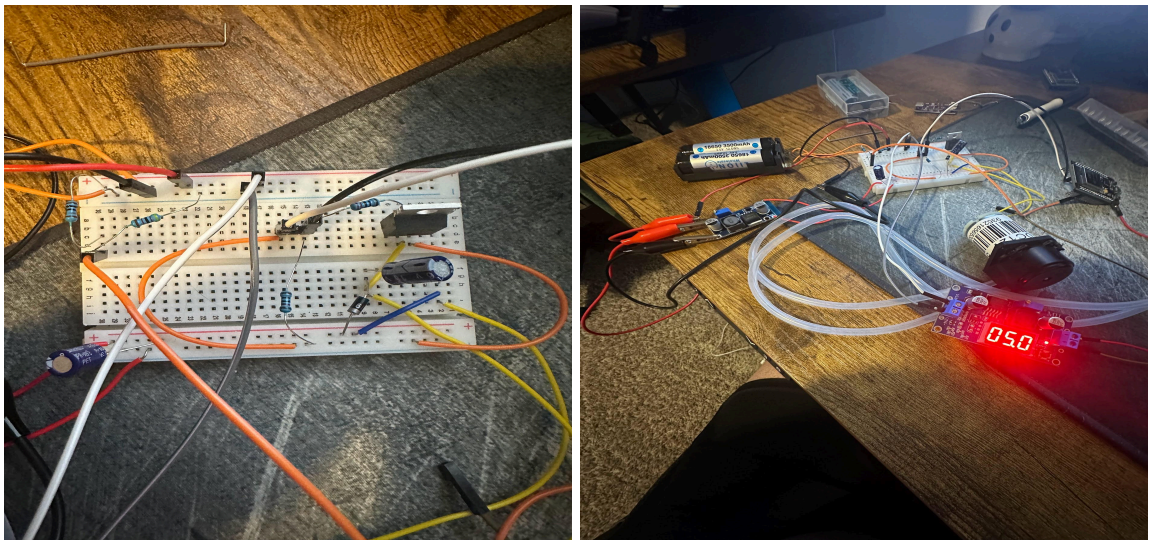


Figure 9.1.1 Breadboard Testing

9.1.1 Sensor testing

9.1.1.1 Light Sensor Testing

The BH1750 sensor was tested using the ESP32 microcontroller over the I2C protocol. Initial verification included ensuring proper wiring to SDA and SCL lines, address detection, and confirming communication via the serial monitor. The BH1750.h library was used to simplify code implementation. The sensor successfully initialized and produced real-time lux readings. Multiple lighting scenarios from direct sunlight to full darkness were simulated to test dynamic range and sensor responsiveness. The result is shown in the images below in order of total darkness, room light, to sunlight.

Table 9.1.1.1 Light Sensor Readings

Sensor Readings 1 (Zero Light)	Sensor Readings 2 (Room Light)	Sensor Readings 3 (Flashlight)
0 lux	45.83 lux	33538.33 lux
0 lux	44.17 lux	33423.33 lux

0 lux	44.17 lux	33281.66 lux
0 lux	44.17 lux	33220.83 lux
0 lux	43.33 lux	33286.66 lux
0 lux	41.67 lux	33295.00 lux
0 lux	45.00 lux	33101.66 lux
0 lux	45.00 lux	33152.50 lux

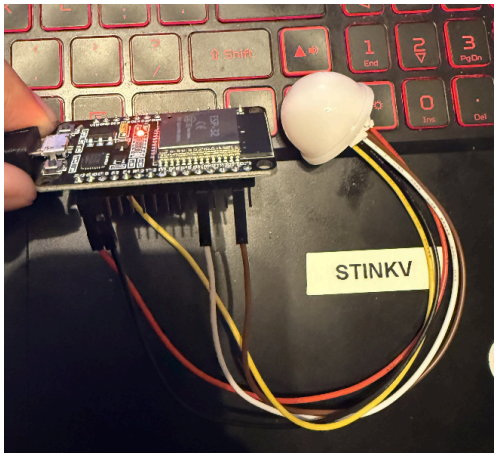


Figure 9.1.1.1 Light Sensor with ESP32

The BH1750 performed well under varying lighting conditions, consistently outputting lux values in line with expected environmental lighting levels lux in full sun. The sensor's high resolution and stable output made it a strong candidate for plant growth monitoring and light-based automation. The BH1750 sensor will be mounted on its own PCB and connected to the main board via header pins. The I2C lines will be shared with other sensors, and pull-up resistors were included in the design to ensure communication stability. The sensor operates independently in a scheduled loop, collecting data at hourly intervals and powering down between reads.

9.1.1.2 Soil Moisture Sensor Testing

The soil moisture sensors purchased from DFRobot were tested using Arduino and Python scripts. The sensors were placed in a styrofoam cup with a certain water level. The sensors were stationary while testing, and the sensors' copper pads were not in contact with the cup material. This process was repeated for various water levels and for both available sensors.

The Arduino test file initializes the sensor input and output pins. After five seconds, the sensor input (power supply) pin is written high. After five seconds, the program begins reading the sensor output value, writing the value to the serial monitor, and waiting for a

one second delay. After recording 200 soil moisture values, the sensor is powered down. These values were copied from the serial monitor and pasted into a text file.

The Python script was originally generated from the Google AI Overview response to a search result for “Python code for parsing file which contains floats separated by spaces” []. This code simply transforms the data in the text file into a Python array. Further lines of code were manually added. These lines casted the Python array into a NumPy array, then NumPy methods were used to calculate the mean and standard deviation values of each array. These values were recorded in Table 9.1.1.2.1. The standard deviation values were normalized into percentages by dividing the standard deviation by the range of the mean values for the respective sensor.

Table 9.1.1.2.1 Soil Moisture Sensor Data Test Results.

Water Level/Content	Average Value		Standard Deviation			
	S1	S2	S1		S2	
0% (sensor fully dry)	2838.925	2838.11	12.63	0.88%	18.98	1.31%
25% Wet	1744.565	1760.125	11.02	0.76%	20.46	1.42%
50% Wet	1485.65	1492.915	14.75	1.00%	16.59	1.15%
75% Wet	1432.93	1427.46	12.45	0.86%	16.61	1.15%
100% (sensor fully wet)	1395.9	1394.285	18.29	1.30%	13.46	0.93%

The mean values for both sensors were very similar at each water level. The range of mean values was approximately 450 for both sensors. The value of the sensor output decreased as the water level of the cup increased. This inversely proportional relationship between water level and sensor output is consistent with the design and theoretical analysis discussed in section 6.1.4.2 of this report. The standard deviation of the sensor output values was approximately 1% for each water level and sensor. There is no relationship between water level and standard deviation. This meets the component specifications for standard deviation listed in section 2.4 of this report. This indicates that the sensor design permits a high degree of precision.

Table 9.1.1.2.2 Soil Moisture Sensor Electrical Measurements.

Conditions	Measurement	Value (Sensor 1)	Value (Sensor 2)
100% Dry	Supply Voltage	3.146 V	3.147 V
	Supply Current	4.79 mA	4.76 mA

	Output Voltage	2.441 V	2.445 V
100% Wet	Supply Voltage	3.145 V	3.145 V
	Supply Current	4.84 mA	4.81 mA
	Output Voltage	1.285 V	1.288 V

Electrical measurements were also recorded for both sensors. The supply voltage to the sensor input was approximately 3.145V under wet and dry conditions for both sensors. The supply current was approximately 4.8mA under both conditions and for both sensors. These supply characteristics are consistent with design and theoretical analysis because the ESP32 GPIO (general-purpose input/output) pins provide an output voltage of around 3.3V for at most 40 mA across all GPIO pins. Lastly, the voltage of the sensor output was approximately 2.44V under dry conditions and 1.28V under wet conditions. This inversely proportional relationship between water level and output voltage is consistent with the design and theoretical analysis discussed in section 6.1.4.2 of this report.

9.1.1.3 Horizontal Float Sensor Testing

The float sensor is a mechanical switch triggered by raising or lowering water levels. It was tested first, out of water, to check that the switch itself was working. This was done by connecting an led to the ESP32, which made the light turn on when the float simulated a low water level. The opposite is also true as the led light turned off when the float was put into a raised position, simulating a full tank. Both images below show the led light turning off on and on when the float is in different positions.

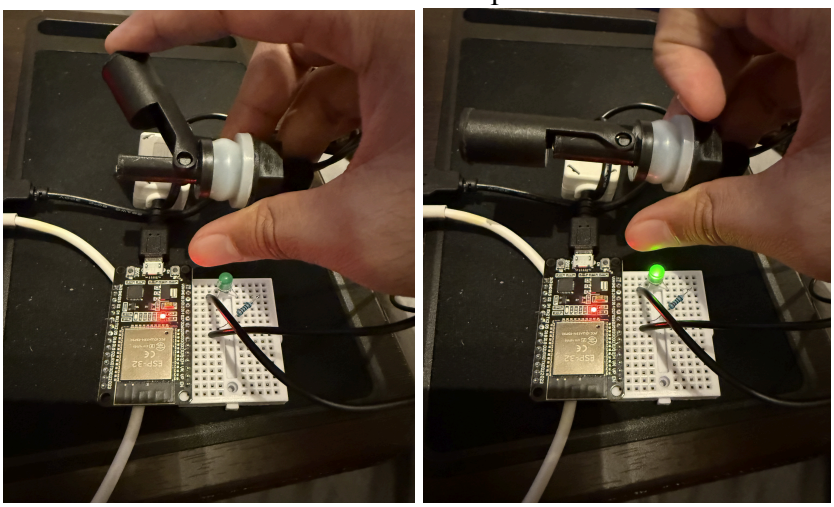


Figure 9.1.1.3.1 Horizontal Float Sensor Test

The float sensor responded quickly and consistently to water level changes. It operated as intended, operating as an off on switch, either connecting or disconnecting the circuit based on float position. The sensor maintained stable output over repeated trials, with no false triggers. Its simplicity, reliability, and waterproof design made it well suited for fixed measurement of the reservoir water levels The sensor itself will be mounted on a

dedicated PCB and wired to the main board using soldered male headers. It integrates as a straightforward digital input and fits cleanly into the system's control logic, where its HIGH or LOW signal determines whether the water level is sufficient for operation. Its modular design and ease of use allowed for flexible positioning within the enclosure and quick troubleshooting during assembly.

9.1.2 Pump Testing

To verify that the selected pump operated correctly and met basic performance expectations, we performed a series of informal bench tests using a breadboard setup. The pump was connected to a 12V DC power supply, with control handled via a MOSFET or relay circuit driven by a digital output pin on the ESP32. This setup simulated how the pump would function in the final system. We tested basic on/off control by toggling the GPIO pin in software, confirming that the pump activated reliably when triggered and shut off cleanly when deactivated. We also checked that the MOSFET or relay could handle the required current without overheating.

In addition to control functionality, we tested water flow by placing the pump tubing in a small container of water and measuring whether it could consistently deliver liquid through the other side. We observed factors like flow rate, priming behavior, and noise during operation. These basic tests confirmed that the pump was suitable for use in the system and could be effectively controlled by the ESP32 using our chosen driver circuit.

9.2 Software Testing

9.2.1 Embedded Software Testing

Embedded software testing was performed by compiling the Arduino file. If the file compiled without errors, the program was loaded onto the ESP32 development kit. Individual methods or functions were usually tested alone in a separate script before integration into the primary file. After integrating new code, the primary file was compiled and tested to ensure compatibility between the new and existing code. No software simulation tools were utilized in order to test the embedded software.

If the software functions under test did not require use of a peripheral component, the software was loaded onto the development kit, and the serial output was monitored. These functions generally required the establishment of an AP server by the ESP32 microcontroller or the connection of the ESP32 to an existing network, after which an HTML file was provided by the ESP32 software. In order to test these functions, we needed to connect to the same network that the ESP32 was connected to, enter the IP address corresponding to the server connection to access the HTML file, and then enter input and submit the form to trigger an HTTP request that sends data to the software. These functions usually corresponded to the setup mode discussed in section 7.2 of this report. When integrated to the primary file, the ESP32 wireless connection would switch between being its own access point and connecting to an existing access point. This would require the designer or tester to also switch between the respective networks.

Other software functions required the use of a peripheral component. The Arduino test scripts for these functions were created or modified from existing code. Documentation from the manufacturer or third-parties often provided a source for specific commands or

lines of code. These units under test were tested as described in section 9.1. Any sources used to create test code should be listed in the test file, which should be copied to the appendices of this document.

Testing of the wireless functionality of the ESP32 required testing the ability of the ESP32 to both send and receive data using HTTP requests. First, we were required to connect the microcontroller to a network access point in order for the microcontroller to be able to interact with devices on that network. To do this, we initially made us of hard-coded WiFi credentials to test the basic WiFi functionality of the device. After configuring the device to connect to a network access point with a given set of credentials, we tested to ensure that the WiFi was connected. Once we confirmed that the connection was working properly, we could configure the microcontroller to both respond to and make HTTP requests.

We first implemented the ability for the microcontroller to make HTTP requests. This was accomplished in the main loop body of the microcontroller test program. The program would attempt to reach out to a local machine and send a message to said machine. On success, the microcontroller would blink a debug LED and report in the arduino serial monitor that it had successfully uploaded data. On failure, the LED would stay lit instead of blinking, and the microcontroller would report the reason for failure to the serial console. Once we confirmed that our local machine was capable of being sent messages from the microcontroller, we tested sending data and confirmed that the microcontroller was able to successfully send data to the machine.

We then implemented the ability for the microcontroller to receive HTTP requests. This was accomplished by setting up the ability for the microcontroller to act as a web server, which required creating a server that would be hosted locally on the microcontroller on a given port. After configuring the server and port on the microcontroller, we then defined HTTP methods that the server could respond to. After defining these methods, we flashed the code to the microcontroller. Once the code was flashed and the microcontroller was connected to our network, we then utilized cURL on a local machine to send HTTP requests to the microcontroller, verifying that malformed or incorrect requests were properly rejected by the microcontroller, while correct requests returned an HTTP code of 200, signaling that the microcontroller had correctly processed our requests.

9.2.2 Web Server Testing

To test the functionality of sending data to and from the ESP32, we created a small webserver for the microcontroller to interact with. To create this webserver, we used ExpressJS to create a simple backend that could be used to send HTTP POST messages from the microcontroller to the server, and display that data on a client browser using an HTTP GET message.

To run this backend, we needed a stable development environment that could provide us with the packages and package versions that we needed. To accomplish this task, we created a Dockerfile based on Ubuntu 24.04 LTS, which we picked for its stability as an LTS release as well as its recency, which allowed us to utilize a version of NodeJS that

was new enough to work with ExpressJS. After creating the Dockerfile, we built the image and entered the resulting container, which placed us in a bash shell with NodeJS and npm already installed from the Ubuntu repositories, alongside their respective dependencies. We were then able to use npm to install necessary JavaScript libraries, ExpressJS in this case, which we could then use to run our server.

Since our server consisted of only a small file, named server.js, we simply ran the file using node. With the server running, we could go to our web browser and ensure that the server was properly running by visiting the IP 127.0.0.1:3000, where 127.0.0.1 refers to our host machine's loopback IP, where the machine can send network traffic back to itself, and 3000 refers to the port number that the server is hosted on. When accessing this IP address through our web browser, we were able to confirm that the server was functioning by viewing the web page. Since we had not yet configured our web page to interact with our microcontroller, for now, viewing the web page only produced our default result, which was that the page had not yet received any data from the microcontroller. However, for the purposes of testing the web server, confirming that the server was properly hosted was enough.

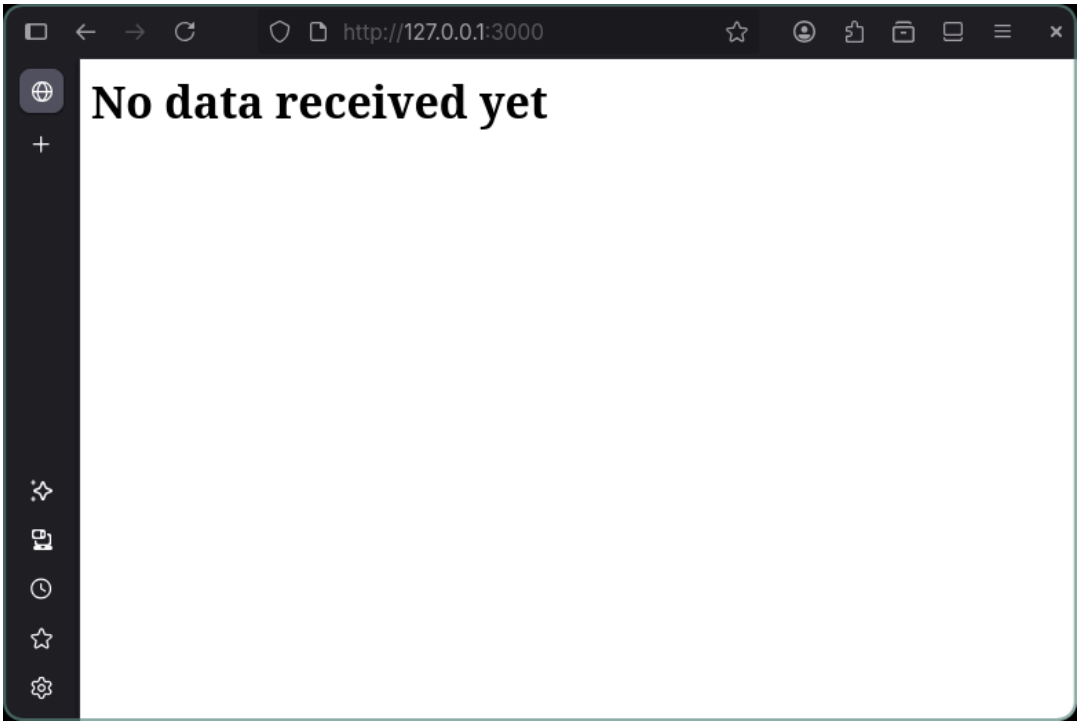


Figure 9.2.2.1 Web Page Test Results

Once we had tested that our server was accessible from our local machine, we then needed to test that it could be accessed by a microcontroller when it came time to integrate the microcontroller and the web server. To do this, we made use of cURL to send an HTTP POST request to the local machine running the server. In order to ensure that the server was correctly opened, this was done from a machine external to the machine running the server. After using cURL to send pseudo data mimicking the data we were expecting from the microcontroller, we were able to observe that the server had

correctly received our data and responded with an HTTP status code of 200, signaling a successful transfer of data.

9.3 overall integration

9.3.1 Web Server / Microcontroller Integration

The resulting web page from our web page testing showed that our server was running, as the page was available, but it also showed that we had no data available from the microcontroller. For this data to appear, we had to create code to send data from the microcontroller to the server. To do this, we modified our initial microcontroller program, making use of built-in WiFi and HTTP libraries from arduino to allow our microcontroller to send HTTP requests to our server running locally. At first, we tested sending a string to the server, which initially failed, with the microcontroller returning that the connection was refused by the server. This allowed us to realize that we had not opened port 3000 on our local testing machine, meaning that the machine's firewall would automatically reject the microcontroller's attempts to connect. After reconfiguring the firewall to allow TCP connections on port 3000, we were able to see results. After this, we modified the server slightly to accept and display JSON data. We then similarly modified the arduino code running on the microcontroller to send arbitrary JSON data. After this, we tested sending the arbitrary JSON data to the server and ensured that the data was received by checking in the browser and the server console.

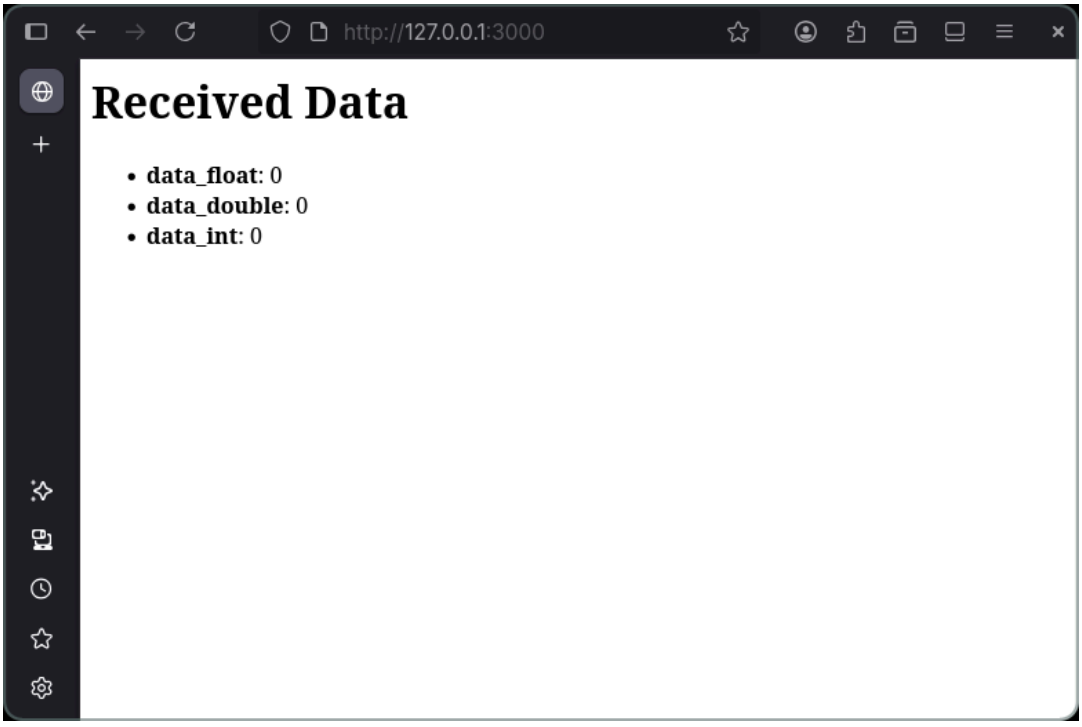


Figure 9.3.1.1 Web Page with received data

The next aspect of the server that we needed to test during integration was its ability to update data on the microcontroller. To do this, we modified our microcontroller code, making use of our microcontroller's ability to function as a webserver as well as libraries

to allow for asynchronous HTTP handling. After configuring the microcontroller to handle HTTP post requests, we then modified the server. In order to send data from the server to the microcontroller, we first needed to ensure that the server knew the IP address of the microcontroller. To do this, we had the server track the IP address of the microcontroller when it sent HTTP requests to the server. With the IP address of the microcontroller, we could then implement a method that would allow an end user to send a POST request to the server. This POST request would send data from the client to the server. The server would then make use of the JavaScript axios package to send that data to the microcontroller. If the data was sent successfully, the server would respond with an HTTP code of 200 and let the client know that the microcontroller data was updated. To test this, the cURL program was used to send a POST request to the server backend, in order to avoid overcomplicating the frontend logic needed for the testing server to support uploading data from a web browser.

Figure 9.3.1.2 Successful update of microcontroller data through server

After sending the POST request via cURL and receiving a response, we were then able to check both the serial monitor of the microcontroller as well as our web browser, which confirmed that the data was properly updated by the microcontroller and the new data was being sent to and received by the server.

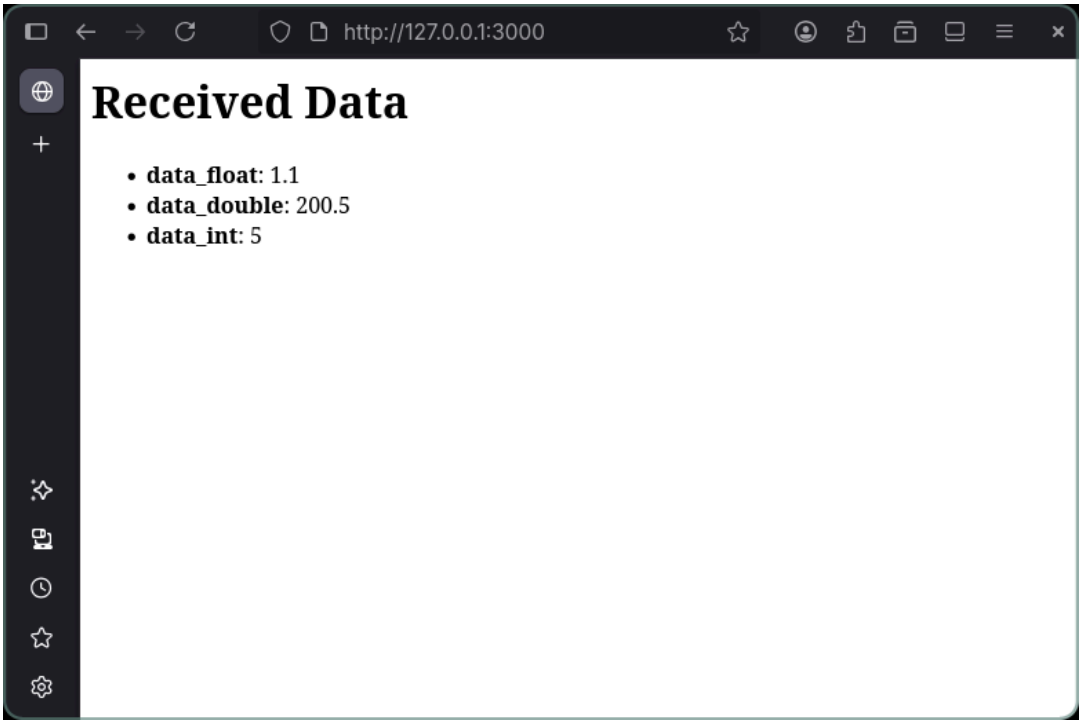


Figure 9.3.1.3 Updated received data seen by the server

The last feature we tested with regards to the web server was the ability to dynamically change the IP address that the microcontroller would send its data to. To accomplish this, we set up multiple instances of our web server on different machines. We then configured the microcontroller to process another HTTP POST request, which would contain the new IP address to use. When sending the new IP address to the microcontroller via cURL, we were able to observe that traffic from the microcontroller was properly redirected from the server instance on one machine to another machine by observing the logs on both machines as well as how the data received by the new machine updated the browser web page on the new machine. This was vital for allowing us to test the functionality of the microcontroller across different devices, as now we could utilize our own local machines instead of having to rely on one server for development.

Overall, integration between the microcontroller and the web server proved to be successful, and allowed for us to easily send requests to and from the microcontroller through the development server. It also allowed us to lay out our groundwork for our server API that could interact with and bridge the gap between the client machine and the microcontroller.

9.4 Plan for SD2

I. Power System Integration & Testing

- Assemble final custom PCB with:
    - BQ24650 charge controller (2S Li-ion)
    - TPS61088 boost converter (12 V rail)
    - TPS62162 buck converter (3.3 V rail)

- Solder all discrete components (capacitors, inductors, MOSFETs, diodes)
- Validate MPPT behavior under real sunlight
- Confirm current delivery for pump startup (>2 A) and 3.3 V logic loads
- Ensure shared ground is stable across battery, sensors, ESP32

II. Sensor & Actuator Integration

- Finalize wiring and calibration for:
    - Capacitive soil moisture sensor
    - Ultrasonic water level sensor
    - Light sensor (LDR or photodiode)
- Connect peristaltic pump through a low-side N-channel MOSFET driver
  Include flyback diode and capacitor to protect against inrush and back-EMF

III. Firmware Development (ESP32)

- Complete sensor sampling logic (every 15–20 min)
- Send HTTP POSTs to backend with sensor data
- Implement soil moisture threshold-based pump activation
- Enable deep sleep between cycles for energy efficiency
- Add battery voltage monitoring via voltage divider
- Implement hysteresis to avoid pump chatter
- Optional: Add OTA updates or remote debug support

IV. Web Application Integration

- Finalize full-stack web app to:
    - Display soil moisture, water level, light data
    - Allow user to configure plant moisture thresholds
    - Trigger pump manually from UI if needed
- Optional: Begin ESPHome/Home Assistant compatibility

V. 3D-Printed Enclosure Completion

- Finalize and print models for:
    - Outer housing
    - Pot with irrigation tubing groove
    - Water reservoir and drainage
    - Electronics enclosure
- Assemble and test internal fit of components
- Install tubing for 360° irrigation ring

VI. System Testing & Validation

- Electrical tests:
    - Confirm regulator output stability
    - Measure charge/discharge curves of battery
- Functional tests:
    - Simulate dry soil to trigger watering
    - Verify deep sleep and wake cycles
    - Test low water detection
- Long-term stress test (multi-day watering cycle)
- Collect graphs and photos for report

VII. Final Deliverables

- Update final report with:
    - Schematic and PCB images

    - Bill of materials
    - Testing results
    - System block diagrams
- Prepare demo video, technical poster, and presentation slides
- Submit documentation per course deadlines

Chapter 10 - Administrative Content

10.1 Budget

Table 10.1.1 Budget Estimate.

Sub System	**Budget**
Power delivery and solar	$80
PCB/ microcontroller	$30
Sensors	$30
PCB and other materials	$100
3d Prints	$25
Total	$265

Table 10.1.2 Bill of Materials.

Category	**Component**	**Link**	**Quantity**	**Cost**	**Total**
MCU	ESP32-WROOM-32E-N16	https://www.digikey.com/en/products/detail/espressif-systems/ESP32-WROOM-32E-N16/11613144	6	5.71	34.26
Battery	Panasonic/Sanyo NCR18650GA	https://liionwholesale.com/products/protected-panasonic-sanyo-ncr18650ga-button-top?variant=12534255236	2	11.99	23.95
Battery holder		https://www.digikey.com/en/products/detail/keystone-electronics/1048P/4499389?gad_source=1&gad_campaignid=20243136172&gbraid=0AAAADrbLlgmLn863JaFo9EzBeR3kEUUr&gclid=CjwKCAjwpMTCBhA-EiwA_-MsmUvwdo_EubkEjuadGt8X6pFbIveFhnYABu2i-86	1	10.77000	10.77

		W0tBMbnRjYoPtTRoCpEMQAvD_BwE&gclsrc=aw.ds			
Solar Panel	2600-P108-ND	https://www.digikey.com/en/products/detail/voltaic-systems/P108/12154978	1	49.00	49
Soil Sensor	Gravity: Analog Capacitive Corrosion Resistant Soil Moisture Sensor for Arduino / Raspberry Pi	https://www.dfrobot.com/product-1385.html	3	5.90	17.70
Light Sensor	BH1750	Amazon.com: ACEIRMC 3pcs BH1750 BH1750FVI Chip Light Intensity Light Module Light Ball for Arduino DIY KIT DC 5V : Electronics	3	4.00	12.00
Water Level Sensor	Horizontal Float Sensor	Amazon.com: Ximimark 3 pc Water Level Sensor Horizontal PP Side Mount Float Switch for Tank Pool Arduino : Industrial & Scientific	3	3.00	9.00
Pump	Adafruit 1150 peristaltic pump	https://www.digikey.com/en/products/detail/adafruit-industries-llc/1150/5638299?s=N4IgTCBcDaIIYBM4DMBOBXAlgFwAQEZ8BWABhAF0BfIA	1	24.95	24.95
Charge Controller	BQ24650RVAR	https://octopart.com/bq24650rvar-texas+instruments-18670764	1	2.631	2.631
Buck converter	TPS62162DSGR	https://octopart.com/tps62162dsgr-texas+instruments-21221928	1	1.38	1.38
Boost convertor	TPS61088RHLT	https://octopart.com/tps61088rhlt-texas+instruments-57155059	1	5.76	5.76
Total					191.25

10.2 Dates and Milestones

The milestones have been set with the information provided so far. We don't have much information about deadlines for SDII. The process for setting deadlines involves us targets 1-2 weeks ahead of the actual deadline to give us adequate time in case of unexpected events.

Table 10.2.1 Senior Design 1 Project Documentation Milestones.

Item	Start Date	TCD	Due Date
Bootcamp Assignment	05/13/25	05/23/25	05/23/25
Divide & Conquer Document and Committee Formation	05/13/25	05/30/25	05/30/25
DiscoveryKITForm	05/13/25	05/31/25	05/31/25
Divide & Conquer (D&C) Document Revision	06/02/25	06/05/25	06/06/25
Assignment on Standards	06/02/25	06/18/25	06/20/25
Midterm Milestone Report	05/13/25	06/21/25	07/07/25
Final Report - Draft	05/13/25	07/12/25	07/29/25
Final Report - Final	07/12/25	07/26/25	07/29/25
Mini Demo Video	07/12/25	07/26/25	07/29/25

Table 10.2.2 Senior Design 1 Project Design.

Item	Start Date	TCD	Due Date
Individual System Design	06/06/25	06/14/25	06/18/25
Individual System Testing	06/18/25	07/12/25	07/26/25
Breadboard Prototyping	06/18/25	07/12/25	07/26/25
PCB Design/Ordering	07/26/25	08/05/25	08/12/25
Component Selection	06/06/25	06/10/25	06/14/25

Table 10.2.3 Senior Design 1I Project Documentation Milestones.

Item	Start Date	TCD	Due Date
PCB Testing	TDB-SDII	TDB-SDII	TDB-SDII
System Integration/Testing	TDB-SDII	TDB-SDII	TDB-SDII
Practice Project Demo	TDB-SDII	TDB-SDII	TDB-SDII

Finalize Documentation	TDB-SDII	TDB-SDII	TDB-SDII
Practice Final Presentation	TDB-SDII	TDB-SDII	TDB-SDII
Final Presentation	TDB-SDII	TDB-SDII	TDB-SDII

10.3 Distribution of Work

The following will be the specific sub systems and other design aspects that each person will be responsible for.

Table 10.3.1 Distribution of roles.

Electrical Engineering	Responsibility
Brandon James	PCB Design and fabrication
	Solar and other power delivery
	3d printing pot design
	Regulator Design
Electrical Engineering	Responsibility
Adrian Vergara	PCB Design and fabrication
	Water pump system
	Solar, soil moisture and water level sensor
	3d printing pot design
Computer Engineering	Responsibility
Brian Ericson	MCU software design
	Sensor design support; MCU/Sensor interfacing
	Web app support
	Smart home integration
Computer Engineering	Responsibility

Elliott D'Amato	MCU software design
	Sensor design support; MCU/Sensor interfacing
	Web app support
	Smart home integration

Chapter 11- Conclusion

The C.H.A.N.O.S. system has come a long way from concept to a working design that is now ready for layout and testing. Our goal from the start was to build a reliable, solar-powered platform that could monitor plant conditions and automatically irrigate as needed. Every part of the system—power, sensing, control, and communication—has been designed to work together in a self-sustaining way.

The power system uses a solar panel to charge a 2-cell lithium-ion battery pack, which then powers the rest of the board. We've taken care to size everything to match the panel and battery so the system can run off-grid without overloading anything. The battery powers two main outputs: one for the irrigation pump and another for the ESP32 and sensors. This gives us the ability to run high and low power devices independently, while keeping everything efficient.

The ESP32 sits at the center of the system, making decisions based on inputs from moisture, light, and water level sensors. If the soil is too dry and there's enough water in the reservoir, the board triggers the pump to irrigate the plants. This behavior is controlled entirely in firmware, giving us flexibility to tweak logic or thresholds without changing hardware. The ESP32 also reads battery level through a voltage divider, so it knows how much power is left and can adjust behavior if needed.

In addition to local control, the ESP32 connects to Wi-Fi and pushes data to a web dashboard. This lets users see real-time stats like soil moisture and battery level, and eventually it can support remote control or notifications. We've also added status LEDs and development headers to help with debugging and future expansion.

At this stage, the entire schematic has been verified—both in terms of connections and part values. We've finalized the core structure of the system and made sure everything is built around practical, available components. What started as a breadboard proof of concept is now a full system design ready for PCB layout and final integration.

Overall, the work done so far sets up the C.H.A.N.O.S. system for successful testing and deployment. It's efficient, modular, and built to scale. The project stays true to the original vision: a smart, self-powered plant system that can make watering decisions

without human input and give users visibility into what's happening—even when they're not around.

Appendix A - references

[1] https://www.electronicsforu.com/technology-trends/learn-electronics/different-types-of-batteries

[2]https://aurorasolar.com/blog/solar-panel-types-guide/

[3]https://www.greenwood.io/news-posts/solar-panel-temperature-coefficients

[4]Battery University. "BU-205: Types of Lithium-ion." https://batteryuniversity.com/article/bu-205-types-of-lithium-ion

[5] Battery University. "BU-302: Series and Parallel Battery Configurations." https://batteryuniversity.com/article/bu-302-series-and-parallel-battery-configurations

[6] U.S. Department of Energy. “Nickel-Metal Hydride Battery Basics.” https://www.energy.gov/eere/vehicles/articles/nickel-metal-hydride-battery-basics

[7] Frontiers in Batteries and Electrochemistry. “Revitalizing Lead-Acid Battery Technology.”

[8] Victron Energy, “PWM or MPPT?” *Victron Energy Blog*, 2023. [Online]. Available: https://www.victronenergy.com/blog/2023/05/08/pwm-or-mppt/

[9] National Renewable Energy Laboratory, “Photovoltaic Charge Controllers,” *U.S. Dept. of Energy*, 2022. [Online]. Available: https://www.nrel.gov/docs/fy22osti/80367.pdf

[10] Renogy, “Types of Solar Charge Controllers,” *Renogy Knowledge Center*, 2023. [Online]. Available: https://www.renogy.com/blog/types-of-solar-charge-controllers/

[11] A. H. Etemadi and P. Das, “Charge controller topologies and control algorithms for photovoltaic systems: A review,” *Renewable and Sustainable Energy Reviews*, vol. 91, pp. 420–435, 2018.

[12] S. Jain and V. Agarwal, “A new algorithm for rapid tracking of maximum power point in photovoltaic systems,” *IEEE Power Electronics Letters*, vol. 2, no. 1, pp. 16–19, Mar. 2004.

[13] D. S. Kirschen and G. Strbac, *Fundamentals of Power System Economics*, 2nd ed., Wiley, 2018.

[14] M. Ali, R. Masood, “Comparative Study of Solar Charge Controllers,” *International Journal of Engineering Research & Technology (IJERT)*, vol. 7, no. 7, pp. 157–160, 2018.

[15] https://openai.com/research/chatgpt

[16] R. W. Erickson and D. Maksimović, *Fundamentals of Power Electronics*, 2nd ed., Springer, 2001.

[17] Texas Instruments, “Switching Regulator Overview,” [Online]. Available: https://www.ti.com/power-management/switching-regulators/overview.html

[18] Samsung SDI, “INR18650-30Q Data Sheet,” 2014. [Online]. Available: https://cdn.shopify.com/s/files/1/0693/6201/files/Samsung_30Q.pdf

[19] LG Chem, “INR18650-MJ1 Specification Sheet,” 2015. [Online]. Available: https://www.imrbatteries.com/content/lg%20mj1%203500mah.pdf

[20] Panasonic, “NCR18650B Lithium Ion Rechargeable Cell Data Sheet,” [Online]. Available: https://cdn.sparkfun.com/datasheets/Batteries/Panasonic_NCR18650B.pdf

[21] Sanyo/Panasonic, “NCR18650GA High Capacity 18650 Data Sheet,” [Online]. Available: https://www.batteryspace.com/prod-specs/NCR18650GA%203500mAh.pdf

[22] We have used Chatgpt for creating the image on the document as well as other proofreading.
[23] Moss, Justin Quetone, et al. “Smart Irrigation Technology: Controllers and Sensors.” OSU Extension, Oklahoma State University, 1 Feb. 2017, https://extension.okstate.edu/fact-sheets/smart-irrigation-technology-controllers-and-sensors.html. Accessed 29 May 2025.
[24] Dong, Younsuk, and Hunter Hansen. “Development and design of an affordable field scale weighing lysimeter using a microcontroller system.” Edited by Stephen Symons. *Smart Agricultural Technology*, 8 Dec. 2022, https://doi.org/10.1016/j.atech.2022.100147.
[25] Hillhouse, Grady. “Arduino Garden Controller - Automatic Watering and Data Logging.” *YouTube*, uploaded by Practical Engineering, 2 April 2015, https://www.youtube.com/watch?v=O_Q1WKCtWiA.
[26] Uhlmann, Martin. “ Flaura – Automatic, Smart Plant Pot (Self Watering, DIY Project, 3D Printed, Arduino) INTRODUCTION.” *YouTube*, uploaded by Flaura - Smart Plant Pot, 6 July 2021, https://www.youtube.com/watch?v=rHHFL17ncnc.
[27] U.S. Department of Energy. *Solar Photovoltaic Manufacturing Basics*. Office of Energy, Efficiency & Renewable Energy, https://www.energy.gov/eere/solar/solar-photovoltaic-manufacturing-basics.
[28] EnergySage. *Monocrystalline vs. Polycrystalline Solar Panels: What’s the Difference?*EnergySage,https://www.energysage.com/solar/monocrystalline-vs-polycrystalline-solar/.
[29]Solar Magazine. *PERC Solar Panels: What They Are and How They Work*. Solar Magazine, https://solarmagazine.com/solar-panels/perc-solar-panels/.
[30]Solar Magazine. *Thin-Film Solar Panels: Types, Technologies, and Applications*. Solar Magazine, https://solarmagazine.com/solar-panels/thin-film-solar-panels/
[31]Engineering with Rosie. *Monocrystalline vs Polycrystalline vs Thin Film Solar Panels: Which is Better?* YouTube, 5 Apr. 2021, https://www.youtube.com/watch?v=OGwUR_yYVKs.
[32]North Ridge Pumps. *What Are Positive Displacement Pumps?* North Ridge Pumps, https://www.northridgepumps.com/article-88_what-are-positive-displacement-pumps.
[33]Eddy Pump Corporation. *What to Know About Submersible Pumps*. Eddy Pump, https://eddypump.com/education/what-to-know-about-submersible-pumps/.
[34]Albin Pump. *How Peristaltic Pumps Work*. Albin Pump, https://www.albinpump.com/en-us/news/how-peristaltic-pumps-work.
[35]EEPower. *Photoresistor (Light-Dependent Resistor)*. EEPower, https://eepower.com/resistor-guide/resistor-types/photo-resistor/#.
[36]Electronics Hub. *Photodiode – Working, Characteristics, Applications & Advantages*.ElectronicsHub,22Feb.2021,
https://www.electronicshub.org/photodiode-working-characteristics-applications/.
[37]Digi-Key Electronics. *The Basics of Photodiodes and Phototransistors and How to ApplyThem*.Digi-Key,
https://www.digikey.com/en/articles/the-basics-of-photodiodes-and-phototransistors-and-how-to-apply-them.
[38]APG Sensors. *The Basics of Float Switches: Guide*. APG, https://apgsensors.com/the-basics-of-float-switches-guide/.

[39]AMS Industrial Automation. *How Does a Float Switch Work?* AMS-IAC, https://ams-iac.com/how-does-a-float-switch-work/.
[40]AtlasScientific.*TypesofFloatSwitches*.AtlasScientific,https://atlas-scientific.com/blog/types-of-float-switches/.
[41]Water Tech Online. *Ultrasonic Sensors for Water Level Measurement*. Water Tech Online,18Mar.2019,https://www.watertechonline.com/home/article/14171108/ultrasonic-sensors-for-water-level-measurement.
[42]vnet. *Pressure Sensors for Water Applications*. Avnet, https://my.avnet.com/abacus/solutions/technologies/sensors/pressure-sensors/media-types/water/.
[43]Reventec.*WhatAreCapacitiveLiquidLevelSensors?*Reventec, https://www.reventec.com/what-are-capacitive-liquid-level-sensors/.
[44]Reventec. *What Are Capacitive Liquid Level Sensors?* Reventec, https://www.reventec.com/what-are-capacitive-liquid-level-sensors/.
[45]Whale.*SubmersibleElectricGalleyPump*.Amazon, https://www.amazon.com/Whale-Submersible-Electric-Galley-Pump/dp/B0014VR37M/.
[46]SEAFLO. *Submersible and Inline Pump 12V/24V*. SEAFLO, https://www.seaflo.com/index.php?m=home&c=View&a=index&aid=547.
[47]Adafruit. *TSL2561 Digital Luminosity/Lux/Light Sensor Breakout*. Adafruit Industries, https://www.adafruit.com/product/1150#technical-details.
[48]Conquerall Electrical. *Conquerall Electrical Website*. https://myconquerall.com/.
[49]Adafruit. *Adafruit BH1750 Ambient Light Sensor*. Adafruit Learning System, https://learn.adafruit.com/adafruit-bh1750-ambient-light-sensor/overview.
[50]AMS. *TSL2561 Light-to-Digital Converter Datasheet*. Mouser Electronics, https://www.mouser.com/datasheet/2/737/tsl2561-932888.pdf.
[51]JESSINIE. *JSN-SR04T Ultrasonic Waterproof Distance Sensor (V3.0)*. Amazon, https://www.amazon.com/JESSINIE-JSN-SR04T-Ultrasonic-JSN-SR04T-V3-0-Waterproof/dp/B0B7MGYM13/
[52]Ximimark. *Water Level Sensor Horizontal Float Switch for Arduino*. Amazon, https://www.amazon.com/Ximimark-Sensor-Horizontal-Switch-Arduino/dp/B0B7WPBPBC/.
[53] Schneider, Josh. “Microcontroller vs Microprocessor: What’s the Difference?” *IBM*, 13 June 2024, www.ibm.com/think/topics/microcontroller-vs-microprocessor.
[54] Gupta, Piyush. “FPGA vs Processor: Understanding the Differences.” *FPGA Insights*, 9 Feb. 2024, fpgainsights.com/fpga/fpga-vs-processor/.
[55] Schneider, Josh. “When to Use FPGA vs Microcontroller.” *IBM*, 3 June 2024, www.ibm.com/think/topics/fpga-vs-microcontroller.
[56] Tirado-Andrés, Francisco, and Alvaro Araujo. “Performance of clock sources and their influence on time synchronization in wireless sensor networks.” *International Journal of Distributed Sensor Networks*, vol. 15, no. 9, 28 Sept. 2019, p. 155014771987937, https://doi.org/10.1177/1550147719879372.
[57] “Timekeeping Accuracy, Automatic and Affordable.” *Analog Devices*, Analog Devices, 17 Aug. 2005, www.analog.com/en/resources/technical-articles/timekeeping-accuracy-automatic-and-affordable.html.
[58] “16 MHz Ceramic Resonator / Oscillator.” *Adafruit*, Adafruit Industries,

www.adafruit.com/product/1873. Accessed 6 July 2025.
[59] Analog Devices. *LTC6906 Micropower, 10kHz to 1MHz Resistor Set Oscillator in SOT-23*. LTC6906. Analog Devices. Accessed 6 June. 2025.
[60] Corpeño, Eduardo. “The Good and the Bad of MCU Internal Oscillators.” *All About Circuits*, 11 Dec. 2019, www.allaboutcircuits.com/technical-articles/the-good-and-the-bad-of-mcu-internal-oscillators/.
[61] "Microcontroller Clock Selection Options." *Analog Device*s, 10 Sept. 2003, www.analog.com/en/resources/design-notes/microcontroller-clock-selection-options.html.
[62] "Soil Moisture." *Open Science Catalog*, 24 Sept. 2012, opensciencedata.esa.int/variables/soil-moisture/catalog.
[63] Rohde & Schwartz. "Introduction into Time Domain Reflectometry." *YouTube*, 17 May 2021, youtu.be/cabqRlQdNSY?si=2Xi4fAn6tyIMsEim. Accessed 14 Jun. 2025.
[64] "Time Domain Reflectometry." *Soil Lab Modules*, labmodules.soilweb.ca/time-domain-reflectometry/.
[65] Evett, Steven R. "Soil Water Measurement by Time Domain Reflectometry." *Wayback Machine*, 2003, web.archive.org/web/20130418081955/http://www.cprl.ars.usda.gov/pdfs/DekkerEvettTDR.pdf.
[66] LiveWire Innovation. "Types of Reflectometry: FDR Vs SSTDR." *YouTube*, 21 Dec. 2011, www.youtube.com/watch?v=yJWlSOXj3ZU.
[67] "How Soil Moisture Sensors Work in Home Gardening and Agricultural Irrigation." *DFRobot*, 12 Jan. 2025, www.dfrobot.com/blog-17283.html.
[68] "Grove - Soil Moisture Sensor." *Seeed Studio*, www.seeedstudio.com/Grove-Moisture-Sensor.html?gQT=1.
[69] "SparkFun Soil Moisture Sensor." *Sparkfun Electronics*, www.sparkfun.com/sparkfun-soil-moisture-sensor.html?gQT=1.
[70] "Gravity: Analog Capacitive Corrosion Resistant Soil Moisture Sensor for Arduino / Raspberry Pi." *DFRobot*, www.dfrobot.com/product-1385.html.
[71] Espressif Systems (Shanghai) Co., Ltd. "Introduction to Low Power Mode in Wi-Fi Scenarios." *API Guides*, docs.espressif.com/projects/esp-idf/en/stable/esp32/api-guides/low-power-mode/low-power-mode-wifi.html.
[72] "Insight Into ESP32 Sleep Modes & Their Power Consumption." *Last Minute Engineers*, lastminuteengineers.com/esp32-sleep-modes-power-consumption/.
[73] Espressif. *ESP32 Series Datasheet.* Ver. 4.9. Espressif, https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.
[74] Espressif. *ESP32-C6 Series Datasheet.* Ver. 1.3. Espressif,https://www.espressif.com/sites/default/files/documentation/esp32-c6_datasheet_en.pdf.
[75] Espressif. *ESP32-S3 Series Datasheet.* Ver. 2.0. Espressif, https://www.espressif.com/sites/default/files/documentation/esp32-s3_datasheet_en.pdf.
[76] Espressif. *ESP32-PICO Series Datasheet.* Ver. 1.1. Espressif, https://www.espressif.com/sites/default/files/documentation/esp32-pico_series_datasheet_en.pdf.

[77] "Esp32-c6." *Mouser Electronics*, www.mouser.com/c/?q=esp32-c6.
[78] "Espressif Systems ESP32-PICO-KIT-1." *Mouser Electronics*, www.mouser.com/ProductDetail/Espressif-Systems/ESP32-PICO-KIT-1?qs=HoCaDK9Nz5ccXDqscvfzVw%3D%3D.
[79] "esp32-s3-devkit." *Mouser Electronics*, https://www.mouser.com/c/?q=esp32-s3-devkit&sort=pricing.
[80] "esp32-d0wd RF System on a Chip - SoC." *Mouser Electronics*, https://www.mouser.com/c/semiconductors/wireless-rf-integrated-circuits/rf-system-on-a-chip-soc/?q=esp32-d0wd&sort=pricing.
[81] "Esp32-pico." *Mouser Electronics*, www.mouser.com/c/?q=esp32-pico.
[82] "Esp32-s3 Pricing." *Mouser Electronics*, www.mouser.com/c/?q=esp32-s3&sort=pricing.
[83] "NRF52840." *Nordic Semiconductor*, www.nordicsemi.com/Products/nRF52840.
[84] "NRF91." *Mouser Electronics*, www.mouser.com/c/?q=nRF91.
[85] "Texas Instruments CC3200R1M2RGCR." *Mouser Electronics*, www.mouser.com/ProductDetail/Texas-Instruments/CC3200R1M2RGCR?qs=bT6MOr62zwJ3zXJW6FHJfg%3D%3D.
[86] "Texas Instruments CC3235SM2RGKR." *Mouser Electronics*, www.mouser.com/ProductDetail/Texas-Instruments/CC3235SM2RGKR?qs=gZXFycFWdAN24yNQJI4ehw%3D%3D.
[87] "Texas Instruments CC3220RM2ARGKR." *Mouser Electronics*, www.mouser.com/ProductDetail/Texas-Instruments/CC3220RM2ARGKR?qs=Mv7BduZupUjRiUUTMz7MyQ%3D%3D.
[88] Texas Instruments. *CC3200 SimpleLink™ Wi-Fi® and Internet-of-Things Solution, a Single-Chip Wireless MCU.* SWAS032F. Texas Instruments, https://www.ti.com/lit/ds/symlink/cc3200.pdf.
[89] Texas Instruments. *CC3200 SimpleLink™ Wi-Fi® and Internet-of-Things Solution, a Single-Chip Wireless MCU.* SWAS035C. Texas Instruments, https://www.ti.com/lit/ds/symlink/cc3220s.pdf.
[90] Texas Instruments. *CC3235S and CC3235SF SimpleLink™ Wi-Fi®, Dual-Band, Single-Chip Solution.* SWRS215E. Texas Instruments, https://www.ti.com/lit/ds/symlink/cc3235s.pdf.
[91] Texas Instruments. *CC3220 SimpleLink™ Wi-Fi® and Internet of Things Technical Reference Manual.* SWRU465. Texas Instruments, https://www.ti.com/lit/ug/swru465/swru465.pdf
[92] "Microchip Technology PIC32MZ1025W104132-I/NX." *Mouser Electronics*, www.mouser.com/ProductDetail/Microchip-Technology/PIC32MZ1025W104132-I-NX?qs=W%2FMpXkg%252BdQ78n1M1zD7i9g%3D%3D.
[93] "Microchip Technology WFI32E01PE-I." *Mouser Electronics*, www.mouser.com/ProductDetail/Microchip-Technology/WFI32E01PE-I?qs=W%2FMpXkg%252BdQ6FpmXU%2Fd7PKw%3D%3D.
[94] "Microchip Technology ATSAMW25H18-MR210PB1961." *Mouser Electronics*, www.mouser.com/ProductDetail/Microchip-Technology/ATSAMW25H18-MR210PB1961?qs=sGAEpiMZZMu3sxpa5v1qrs7aFKzpKeg13ZaX5Sz%252B6zo%3D.
[95] Microchip. *PIC32MZ W1 MCU and WFI32 Module with Wi-Fi® and*

Hardware-Based Security Accelerator Data Sheet. DS70005425N, https://ww1.microchip.com/downloads/aemDocuments/documents/WSG/ProductDocuments/DataSheets/PIC32MZ-W1-and-WFI32E01-Family-Data-Sheet-DS70005425.pdf.
[96] Atmel. *ATSAMW25-MR210PB IEEE 802.11 b/g/n SmartConnect Wi-Fi Module DATASHEET*. Microchip, https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42618-SmartConnect-ATSAMW25-MR210PB_Datasheet.pdf.
[97] "Texas Instruments CC3220S-LAUNCHXL." *Mouser Electronics*, www.mouser.com/ProductDetail/Texas-Instruments/CC3220S-LAUNCHXL?qs=KuuZdrM3jLwCt5Hz59KiTg%3D%3D.
[98] "Microchip Technology EV67T15A." *DigiKey*, www.digikey.com/en/products/detail/microchip-technology/EV67T15A/24617041.
[99] "3 Pack ESP32 ESP-WROOM-32 Development Board WiFi + Bluetooth CP2102 Dual Core Microcontroller Compatible with Arduino." *Amazon*, www.amazon.com/Hosyond-ESP-WROOM-32-Development-Microcontroller-Compatible/dp/B0C7C2HQ7P?pd_rd_w=rend9&content-id=amzn1.sym.cd152278-debd-42b9-91b9-6f271389fda7&pf_rd_p=cd152278-debd-42b9-91b9-6f271389fda7&pf_rd_r=E2K942V3Y1H9ZFQ2Q7MQ&pd_rd_wg=MKLQH&pd_rd_r=834753e5-2355-4264-bcf4-52ff5e29fe8e&pd_rd_i=B0C7C2HQ7P.
[100] "Adafruit STEMMA Soil Sensor - I2C Capacitive Moisture Sensor." *Adafruit*, learn.adafruit.com/adafruit-stemma-soil-sensor-i2c-capacitive-moisture-sensor.
[101] "SparkFun Soil Moisture Sensor." *Sparkfun Electronics*, www.sparkfun.com/sparkfun-soil-moisture-sensor.html?gQT=1.
[102] "Gravity: Analog Capacitive Corrosion Resistant Soil Moisture Sensor for Arduino / Raspberry Pi." *DFRobot*, www.dfrobot.com/product-1385.html.
[103] "Grove - Soil Moisture Sensor." *Seeed Studio*, www.seeedstudio.com/Grove-Moisture-Sensor.html?gQT=1.
[104] "802.11-2007 - IEEE Standard for Information Technology - Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks - Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications." *IEEEXplore*, ieeexplore.ieee.org/document/4248378.
[105] "RFC 9112 HTTP/1.1." *RFC Editor*, www.rfc-editor.org/rfc/rfc9112.pdf.
[106] "RFC 9110 HTTP Semantics." *RFC Editor*, www.rfc-editor.org/rfc/rfc9110.pdf.
[107] "What Kind of HTTP Version Does Current ESP32 Support? #7530." *Espressif / Arduino-esp32*, github.com/espressif/arduino-esp32/discussions/7530.
[108] "IEC 61191-1:2018." *International Electrotechnical Commission*, webstore.iec.ch/en/publication/30733.
[109] "IEC 61086-3-1:2004." *International Electrotechnical Commission*, webstore.iec.ch/en/publication/4481.
[110] "IPC (Electronics)." *Wikipedia*, en.wikipedia.org/wiki/IPC_(electronics).
[111] "IEC TR 61191-8:2021." *International Electrotechnical Commission*, webstore.iec.ch/en/publication/67433.
[112] "ESP32 Hardware Design Guidelines Version 2.1." *Espressif Systems*, public.robotical.io/Datasheets/ESP32%20Hardware%20Design%20Guidelines.pdf.

[113] “Days of Sunshine per Year in Florida.” *Annual Days of Sunshine in Florida - Current Results*, www.currentresults.com/Weather/Florida/annual-days-of-sunshine.php. Accessed 7 July 2025.
[114] “Sun Hours Map: How Many Sun Hours Do You Get?” *Unbound Solar*, 9 Mar. 2021, unboundsolar.com/solar-information/sun-hours-us-map.
[115] *Factsheet*, www.ustravel.org/sites/default/files/media_root/document/NPVD19_Fact_Sheet.pdf. Accessed 7 July 2025.
[116] “P108 Drawing.” *Voltaic*, voltaicsystems.com/content/P108_dwg.pdf. Accessed 7 July 2025.
[117] “Protection Norms and Standards.” *World Health Organization*, World Health Organization, www.who.int/teams/environment-climate-change-and-health/radiation-and-health/protection-norms. Accessed 7 July 2025.
[118] *Icnirpstatement*, www.icnirp.org/cms/upload/publications/ICNIRPNewTech.pdf. Accessed 7 July 2025.
[119] Espressif. *ESP-IDF Programming Guide*. Ver. 5.4.2. Espressif, https://docs.espressif.com/projects/esp-idf/en/v5.4.2/esp32/index.html
[120] "BH1750 Ambient Light Sensor IC Series: Video, Datasheet, Features and Applications." *Utmel*, 13 July 2022, www.utmel.com/components/bh1750-ambient-light-sensor-ic-series-video-datasheet-features-and-applications?id=1863.
[121] “esp32-wroom-32 - RF Transceiver Modules and Modems.” *DigiKey*, https://www.digikey.com/en/products/filter/rf-transceiver-modules-and-modems/872?s=N4IgjCBcoLQExVAYygFwE4FcCmAaEA9lANogCsIAugL7X4KSnYDOADgMxwwDu6BBAWxicq%2BAGyIQASwAmUEDDAAGCPlap5IfKgCerbPICGzFLSA.
[122] Moshiri, Sam. "USB to UART (RS232/Serial) Converter, MCP2200." *Autodesk Instructables*. https://www.instructables.com/USB-to-UART-RS232Serial-Converter-MCP2200/
[123] "USB-UART CH340 Converter Board." Soldered, soldered.com/product/usb-uart-ch340-converter-board/.
[124] "ESP32 Thing Hookup Guide." *Sparkfun Electronics*, learn.sparkfun.com/tutorials/esp32-thing-hookup-guide.
[125] Lindblom, Jim. "Esp32-thing." *Sparkfun*, 7 Oct. 2016, cdn.sparkfun.com/assets/learn_tutorials/5/0/7/esp32-thing-schematic.pdf.
[126] "What Languages Can I Use with the Esp32?" *Reddit*, www.reddit.com/r/esp32/comments/x035er/what_languages_can_i_use_with_the_esp32/.
[127] "ESP32 Auto Program Fix." *HydraRaptor*, 7 Aug. 2021, hydraraptor.blogspot.com/2021/08/esp32-auto-program-fix.html.
[128] "Tips and Quirks." Espressif, docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/jtag-debugging/tips-and-quirks.html#jtag-debugging-tip-openocd-configure-target.
[129] Chen, Bernie. “Schematic Prints of SEN0193 Capacitive Soil Moisture SensorV1.0.SchDoc.” DFRobot, 27 Aug. 2015, https://dfimg.dfrobot.com/nobody/wiki/25ce0211fa0bf9fb5653d67dafcf2aba.pdf
[130] "TLC555 LinCMOSTM Technology Timer." *Texas Instruments*, 1 Nov. 2023, www.ti.com/lit/ds/symlink/tlc555.pdf?HQS=dis-dk-null-digikeymode-dsf-pf-null-wwe&ts=1752264163972&ref_url=https%253A%252F%252Fwww.ti.com%252Fgeneral%252Fdocs%252Fsup

pproductinfo.tsp%253FdistId%253D10%2526gotoUrl%253Dhttps%253A%252F%252Fwww.ti.com%252Flit%252Fgpn%252Ftlc555.
[131] "XC6206 Series." *Torex*, www.mouser.com/datasheet/2/760/xc6206-3371512.pdf.
[132] "Small Signal Diodes 1N4148WS, 1N4448WS, 1N914BWS." Onsemi, www.onsemi.com/download/data-sheet/pdf/1n4148ws-d.pdf.
[133] Lindblom, Jim. “Esp32-Thing-Schematic.” SparkFun, 7 Oct. 2016, https://cdn.sparkfun.com/assets/learn_tutorials/5/0/7/esp32-thing-schematic.pdf.
[134] Santos, Sara, and Rui Santos. “Input Data on HTML Form ESP32/ESP8266 Web Server Arduino IDE.” *Random Nerd Tutorials*, 29 Aug. 2019, randomnerdtutorials.com/esp32-esp8266-input-data-html-form/.
[135] “ESP32_DevKitc_V4.” *Espressif*, 6 Dec. 2017, https://dl.espressif.com/dl/schematics/esp32_devkitc_v4-sch.pdf.

Appendix B - copyright permission

1.

Hello,

I am a Computer Engineering student at the University of Central Florida. I am currently working on a senior capstone project for an automatic houseplant watering and nourishment system. I read your 2017 OSU article on Smart Irrigation Technology and it contains some information on smart irrigation controllers and sensors that I would like to cite in my capstone report. We are required to obtain written permission from the authors/creators for the content we use and cite in our report. I have CC'ed the other members of my capstone group for transparency.

Would we be permitted to use and cite your article in our report?

Thank you,
Brian Ericson

2.

Moss, Justin Quetone<mossjq@okstate.edu>

To: Brian Ericson; malarie.gotcher@okc.gov; saleh.taghvaeian@unl.edu; **+1 other** Sat 5/24/2025 5:36 PM

Cc: Brandon James; Adrian Vergara; Elliott D'Amato

Retention: UCF Delete after 10 Years (10 years) Expires: Tue 5/22/2035 5:36 PM

Yes, please proceed.

Justin Quetone Moss
405.744.5415

https://doi.org/10.1016/j.atech.2022.100147 ↗

Get rights and content ↗

3. Under a Creative Commons license ↗

Open access

You are free to:

Share — copy and redistribute the material in any medium or format

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit , provide a link to the license, and indicate if changes were made . You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial — You may not use the material for commercial purposes .

NoDerivatives — If you remix, transform, or build upon the material, you may not distribute the modified material.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation .

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

Hello,

I am a Computer Engineering student at the University of Central Florida. I am currently working on a senior capstone project for an automatic houseplant watering and nourishment system. I watched your YouTube video on your Arduino Garden Controller and it contains some information on automatic watering sensors and systems that I would like to cite in my capstone report. We are required to obtain written permission from the authors/creators for the content we use and cite in our report. I have CC'ed the other members of my capstone group for transparency.

Would we be permitted to use and cite your video in our report?

Thank you,
Brian Ericson

4.

Grady Hillhouse<grady@practical.engineering>
To: Brian Ericson
Cc: Brandon James; Adrian Vergara; Elliott D'Amato
Tue 5/27/2025 10:46 AM

Retention: UCF Delete after 10 Years (10 years) Expires: Fri 5/25/2035 10:46 AM

Brian,

Glad that video is still being seen! Feel free to use it in your report. As long as the video isn't downloaded and rehosted elsewhere, it's not a problem. Best of luck on your project.

Grady

5. This is a detailed introductory video to Flaura, my Do-It-Yourself, Open-source project.

6.

发件人: DFRobot <store@dfrobot.com>
发送时间: 2025年7月15日 19:01
收件人: brian.ericson@ucf.edu <brian.ericson@ucf.edu>; Tech Support <techsupport@dfrobot.com>
主题: Re: Product Use in Academic Project

Dear Brian Ericson,

Thank you for reaching out to us.

Regarding your request, we are pleased to grant permission. However, we kindly ask that you indicate that the design in the schematic is derived from DFRobot.

For the remaining inquiry, @Tech Supportplease check below email and help on this, thanks.

Also, would you be able to tell me what type of coating(s) was used for this sensor?

Best Regards,
DFRobot Team

DFROBOT
DRIVE THE FUTURE

DFRobot Team | Customer Service
Sales | **DFRobot**
A: Room 603, 2 Boyun Road, Pudong, Shanghai | 201203 P.R.China
P: +8621-61620183-635
W: www.dfrobot.com

a. Permission to use schematic / design of soil moisture sensor from DFRobot (Designer/Manufacturer).

Contact RNT

RS

Rui Santos<rffsantos16@gmail.com>

To: Brian Ericson

Mon 6/23/2025 10:56 AM

Some content in this message has been blocked because the sender isn't in your Safe senders list.

Trust sender | Show blocked content

Retention: UCF Delete after 10 Years (10 years) Expires: Thu 6/21/2035 10:56 AM

Hello Brian

We're glad that our resources are helpful for your own projects.

Please note that we don't allow anyone to redistribute our work or to re-publish it online if the project matches like 95% of the original project.

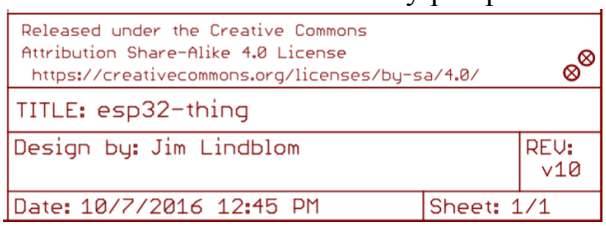
With this said...
You can definitely use code in your project, by giving us the appropriate credit (link back to the original work).

Regards.
Rui

7.

a. Permission to use/modify/paraphrase select lines of code in software

8.



You are free to:

Share — copy and redistribute the material in any medium or format for any purpose, even commercially.

Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit , provide a link to the license, and indicate if changes were made . You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation .

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

a. permission to use USB-UART circuit design.

Appendix C - Data Sheet

Appendix D - Software code

Soil Moisture Sensor Python Test Script

```
import numpy as np


def op(arr):
    average = arr.mean()
    std_dev = arr.std()
    print(average)
    print(std_dev)


data = []
lines = open("C:\\Users\\minec\\classes\\sd1\\testing\\sm_sensor_test_script\\test_val.txt",
'r')
for line in lines:
     string_arr = line.split()
     for str in string_arr:
          num = float(str)
          data.append(num)


#print(data)
numpy_array = np.array(data)
print(numpy_array)
op(numpy_array)
'''low_cond = numpy_array > 10
numpy_array = numpy_array[low_cond]
print(numpy_array)
op(numpy_array)
numpy_array = np.delete(numpy_array, 0)
print(numpy_array)
op(numpy_array)'''
```

Primary Arduino File

```
/*********
  Parts of this code were selected/inspired from Rui Santos & Sara Santos - Random Nerd
Tutorials
  https://RandomNerdTutorials.com/esp32-esp8266-input-data-html-form/
*********/


////////////////////////////////////////////////////////////////////////////////////////////////////////////
/* ARDUINO AND ESP LIBRARIES */
////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
#include <Arduino.h>
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include <WiFiAP.h>
#include "esp_sleep.h"
#include <Wire.h>
#include <BH1750.h>

////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/* GLOBAL VARIABLES */
////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// async_web_server used when ESP32 is in AP mode, creating its own WiFi network
AsyncWebServer async_web_server(80);

// main_server used when ESP32 is in station mode, connected to user's WiFi network
AsyncWebServer main_server(80);

// SSID & password when ESP32 is in AP mode
// user connects to ESP32 AP to send SSID & password for personal WiFi
const char* esp_ap_ssid = "vDIg4qYsY2HZ";
const char* esp_ap_pw = "I6,mqr!Cs4ui";
IPAddress pot_ip_ap = "";

// SSID & password for user's personal WiFi
// ESP will connect to this AP as station for length of use
//String client_ap_ssid = "";
//String client_ap_pw = "";
String client_ap_ssid = "";
String client_ap_pw = "";
IPAddress pot_ip_sta = "";

// state variables
bool end_ap_state = false;
bool end_station_init_state = false;
bool enter_sleep_state = false;
bool moisture_sensor_init = false;
bool sensor_val_init = false;
bool initial_sensor_check_done = false;
bool operation_server_init = false;
bool leave_display = false;

bool get_dry_SMV_flag = false;
bool get_wet_SMV_flag = false;
```

```
// "Valid values should be positive values less than RTC slow clock period * (2 ^ RTC timer bitwidth)."
int sleep_duration_us = 0;
float current_SMV = -1;
float dry_SMV = -1;
float wet_SMV = -1;
float SMV_frac = 0;
float SMV_threshold = -1;

unsigned int watering_duration_us = 0;
float water_lvl = -1;
float light_val_lux = -1;
unsigned int total_sunlight_cnt = 0;
unsigned int maximum_sunlight = 0;
float battery_lvl = 0;
bool water_level_low = false;

// variables to hold display values for sensor_init_page
String SMV_perc_disp = "";
String max_sun_disp = "";
String sampling_period_disp = "";
String watering_duration_disp = "";

const int water_lvl_pin = 36; //GPIO36 - pin 4
const int sms_data_pin = 34; //GPIO34 - pin 6
const int battery_lvl_pin = 35; //GPIO35 - pin 7
const int sms_power_pin = 16; //GPIO16 - pin 27
const int light_sensor_pwr_pin = 17; //GPIO17 - pin 28
const int pump_ctrl_pin = 18; //GPIO18 - pin 30
const int wls_power_pin = 23; //GPIO23 - pin 37
const int ledPin     = 2;    // Onboard blue LED

// use addr 0x23 if addr pin voltage is < 0.7*Vcc
// use addr 0x5C if addr pin voltage is > 0.7*Vcc
BH1750 light_sensor(0x23);

// ADC and calibration
const float vRef = 3.3;         // ADC reference voltage (ESP32)
const float correctionFactor = 1.11; // Tune this to match real voltmeter
const float R1 = 100.0; // kΩ (top)
const float R2 = 47.0;  // kΩ (bottom)


////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/* HTML PAGES/FILES */
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
// html page; interfaces between ESP AP & user
// user defines SSID and password of personal WiFi network
char front_page[] = R"(
<!DOCTYPE HTML> <html>
<head> <title>front_page</title> </head>
<body>
<form action=/get target="_self">
<label for="ssid">SSID:</label>
<input type="text" name="ssid"><br>
Current ssid = %ssid%<br><br>
<label for="pswd">Password:</label>
<input type="text" name="pswd"><br>
Current password = %pswd%<br>
<input type="submit" value="Enter">
</form>
</body> </html>)";



char dry_SMV_sensor_page[] = R"rawliteral(
<!DOCTYPE HTML> <html>
<head> <title>dry_SMV_sensor_page</title> </head>
<body>
<p>The moisture sensor must be held in the air to initialize value of dry soil (0% humidity).<br>
Press OK when moisture sensor is held in the air.</p>
<form action="/get" target="_self">
<input type="submit" name="submit_button" value="OK">
</form>
</body> </html>)rawliteral";

char wet_SMV_sensor_page[] = R"rawliteral(
<!DOCTYPE HTML> <html>
<head> <title>wet_SMV_sensor_page</title> </head>
<body>
<p>The moisture sensor must be held in water to initialize value of wet soil (100% humidity).<br>
Press OK when moisture sensor is held in the water.</p>
<form action="/get" target="_self">
<input type="submit" name="submit_button" value="OK">
</form>
</body> </html>)rawliteral";

char sensor_init_page[] = R"(
<!DOCTYPE HTML> <html>
```

```
<head> <title>sensor_init_page</title> </head>
<body>
<form action=/get target="_self">
<label for="SMV_perc">minimum soil moisture threshold (percent):</label>
<input type="text" name="SMV_perc"><br>
Current min SMV threshold = %SMV_perc%<br><br>
<label for="max_sun">maximum time of sunlight (minutes):</label>
<input type="text" name="max_sun"><br>
Current maximum time of sunlight (minutes) = %max_sun%<br><br>
<label for="sampling_period">sampling period (minutes):</label>
<input type="text" name="sampling_period"><br>
Current sampling period (minutes) = %sampling_period%<br><br>
<label for="watering_duration">watering_duration (seconds):</label>
<input type="text" name="watering_duration"><br>
Current watering duration (seconds) = %watering_duration%<br><br>
<input type="submit" value="Enter">
</form>
</body> </html>)";

char sensor_disp_page[] = R"(
<!DOCTYPE HTML> <html>
<head> <title>sensor_disp_page</title> </head>
<body>
<form action="/get" target="_self">
Current battery level = %battery_lvl%<br><br>
Dry soil moisture value = %dry_SMV%<br><br>
Wet soil moisture value = %wet_SMV%<br><br>
Current soil moisture value = %current_SMV%<br><br>
Current light value = %light_val_lux%<br><br>
Current total_sunlight = %total_sunlight_cnt%<br><br>
Water level is low = %water_level_low%<br><br>
<input type="submit" name="submit_button" value="OK">
</form>
</body> </html>)";


int serial_output = 255;

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////
void setup() {
  Serial.begin(9600);

  Serial.println("Initializing pin modes.");
  pinMode(pump_ctrl_pin,OUTPUT); digitalWrite(pump_ctrl_pin,LOW);
  pinMode(wls_power_pin,INPUT_PULLUP);
```

```
  pinMode(ledPin, OUTPUT); digitalWrite(ledPin, LOW);
  //digitalWrite(wls_power_pin,LOW);
  pinMode(sms_power_pin,OUTPUT); digitalWrite(sms_power_pin,LOW);
  pinMode(light_sensor_pwr_pin,OUTPUT); digitalWrite(light_sensor_pwr_pin,LOW);
  //pinMode(sms_data_pin,INPUT);
  //pinMode(battery_lvl_pin,INPUT);
  pinMode(water_lvl_pin,INPUT);
  Wire.begin();
  //light_sensor.begin();
  Serial.println("Initialized pin modes.");

   while (!end_ap_state || !end_station_init_state || !moisture_sensor_init || !sensor_val_init
|| !operation_server_init) {
    if(!end_ap_state) {
      ap_init_state();
      delay(500);
    }
    else if(!end_station_init_state) {
      async_web_server.end();
      station_init_state();
      delay(500);
    }

    else if(!moisture_sensor_init) {
      init_moisture_sensor();
      delay(500);
    }
    else if(!sensor_val_init) {
      init_sensor_val();
      delay(500);
    }
    else if(!operation_server_init) {
      /*Serial.println("Disconnecting from WiFi.");
      main_server.end();
      WiFi.setAutoReconnect(false);
      WiFi.disconnect();

      init_operation_server();*/
      operation_server_init = true;

      delay(500);
    }
  }
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////
// replaces html variable names with value of corresponding variable in ESP memory
String varRepl(const String& var){
  if(var == "ssid") {
    return client_ap_ssid;
  }
  if(var == "pswd") {
    return client_ap_pw;
  }
  if(var == "SMV_perc") {
    return SMV_perc_disp;
  }
  if(var == "max_sun") {
    return max_sun_disp;
  }
  if(var == "sampling_period") {
    return sampling_period_disp;
  }
  if(var == "watering_duration") {
    return watering_duration_disp;
  }
  if(var == "battery_lvl") {
    return String(battery_lvl);
  }
  if(var == "light_val_lux") {
    return String(light_val_lux);
  }
  if(var == "total_sunlight_cnt") {
    return String(total_sunlight_cnt);
  }
  if(var == "water_level_low") {
    return String(water_level_low);
  }
  if(var == "dry_SMV") {
    return String(dry_SMV);
  }
  if(var == "wet_SMV") {
    return String(wet_SMV);
  }
  if(var == "current_SMV") {
    return String(current_SMV);
  }
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////
void ap_init_state () {
  WiFi.mode(WIFI_AP);
  if(WiFi.softAP(esp_ap_ssid, esp_ap_pw)){
    pot_ip_ap = WiFi.softAPIP();
  }
  else {
    Serial.println("Error creating AP.");
    while (true);
  }
  if(serial_output & 1 != 0) {
    Serial.print("ESP AP IP Address: ");
    Serial.println(pot_ip_ap);
    serial_output = serial_output & 254;
  }

  digitalWrite(ledPin, HIGH);
  async_web_server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(200,"text/html",front_page, varRepl); } );

  async_web_server.on("/get", HTTP_GET, [](AsyncWebServerRequest *request){
    if(request->hasParam("ssid")) {
      client_ap_ssid = request->getParam("ssid")->value();
    }
    if(request->hasParam("pswd")) {
      client_ap_pw = request->getParam("pswd")->value();
    }
    end_ap_state = (client_ap_ssid != "" && client_ap_pw != "");
    request->send(200, "text/html", front_page, varRepl);
  });

  async_web_server.begin();

  while (!end_ap_state) {
    //Serial.print("ESP AP IP Address: "); Serial.println(pot_ip_ap);
    delay(200);
  }
  digitalWrite(ledPin, LOW);
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////
void station_init_state() {
  int counter = 0;
  if(end_ap_state) {
```

```
  Serial.println("Disconnecting ESP as AP.");
   if(WiFi.softAPdisconnect()){
     Serial.println("Initializing ESP32 in station mode.");
     WiFi.mode(WIFI_STA);
     if(client_ap_ssid != "" && client_ap_pw != "") {
       WiFi.begin(client_ap_ssid, client_ap_pw);
       while(counter < 20) {
         if(WiFi.isConnected()) {
           Serial.println("ESP connected to WiFi in station mode.");
           pot_ip_sta = WiFi.localIP();
           Serial.println("IP address: "); Serial.println(pot_ip_sta);
           WiFi.setAutoReconnect(true);
           end_station_init_state = true;
           return;
         }
         else {
           delay(500); Serial.print(" ... "); counter++;
         }
       }
       Serial.println("Error connecting to WiFi. Please reenter WiFi info.");
       end_ap_state = false;
       return;
     }
   }
  }
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////
float get_wet_SMV() {
  digitalWrite(sms_power_pin,HIGH);
  delay(2500);
  float SMV_temp = 0; int N = 10;
  float wet_SMV = analogRead(sms_data_pin);

  Serial.println("\nSMV during get_wet_SMV function:");
  for (int i = 0; i < N; i++) {
    delay(250);
    SMV_temp = analogRead(sms_data_pin);
    Serial.print(SMV_temp); Serial.print(" ");
    if (wet_SMV > SMV_temp)
      wet_SMV = SMV_temp;
  }
  Serial.println();

  digitalWrite(sms_power_pin,LOW);
```

```
  Serial.print("wet (minimum) SMV: "); Serial.println(wet_SMV);
  return wet_SMV;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////
float get_dry_SMV() {
  digitalWrite(sms_power_pin,HIGH);
  delay(2500);
  float SMV_temp = 0; int N = 10;
  float dry_SMV = analogRead(sms_data_pin);

  Serial.println("\nSMV during get_dry_SMV function:");
  for (int i = 0; i < N; i++) {
    delay(100);
    SMV_temp = analogRead(sms_data_pin);
    Serial.print(SMV_temp); Serial.print(" ");
    if (dry_SMV < SMV_temp)
      dry_SMV = SMV_temp;
  }
  Serial.println();

  digitalWrite(sms_power_pin,LOW);
  Serial.print("dry (maximum) SMV: "); Serial.println(dry_SMV);
  return dry_SMV;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////
void init_moisture_sensor() {
  Serial.println("IP address: "); Serial.println(pot_ip_sta);
  Serial.println("Starting initialization for dry SMV");

  digitalWrite(ledPin, HIGH);
  while(dry_SMV < 0) {
    get_dry_SMV_flag = false;
    main_server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    (*request).send(200,"text/html",dry_SMV_sensor_page); } );
    main_server.on("/get", HTTP_GET, [](AsyncWebServerRequest *request){
      if(request->hasParam("submit_button")) {
        //dry_SMV = get_dry_SMV();
        //dry_SMV = 5000;
        get_dry_SMV_flag = true;
        Serial.println(dry_SMV);
      }
      request->send(200, "text/html", dry_SMV_sensor_page);
```

```
  });

  main_server.begin();

  while (!get_dry_SMV_flag) {
    delay(200);
  }
  dry_SMV = get_dry_SMV();
}
digitalWrite(ledPin, LOW);

  Serial.println("Finished initialization for dry SMV");
  //main_server.end();

main_server.reset();

digitalWrite(ledPin, HIGH);
Serial.println("Starting initialization for wet SMV");
while(wet_SMV < 0) {
  main_server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    (*request).send(200,"text/html",wet_SMV_sensor_page); } );
  main_server.on("/get", HTTP_GET, [](AsyncWebServerRequest *request){
    if(request->hasParam("submit_button")) {
      //wet_SMV = 1400;
      get_wet_SMV_flag = true;
      Serial.println(wet_SMV);
    }
    request->send(200, "text/html", wet_SMV_sensor_page);
  });

  main_server.begin();

  while (!get_wet_SMV_flag) {
    delay(200);
  }

  wet_SMV = get_wet_SMV();
}


Serial.println("Finished initialization for wet SMV");
digitalWrite(ledPin, LOW);

moisture_sensor_init = !(dry_SMV < 0 || wet_SMV < 0);
Serial.print("moisture_sensor_init: "); Serial.println(moisture_sensor_init);
//Serial.println(dry_SMV);
```

```
  //Serial.println(wet_SMV);
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////
void init_sensor_val() {
  Serial.println("IP address: "); Serial.println(pot_ip_sta);
  Serial.println("Sending sensor initialization html page");

  main_server.reset();

  digitalWrite(ledPin, HIGH);
  main_server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(200,"text/html",sensor_init_page, varRepl); } );

  main_server.on("/get", HTTP_GET, [](AsyncWebServerRequest *request){
    if(request->hasParam("SMV_perc")) {
      SMV_perc_disp = request->getParam("SMV_perc")->value();
      SMV_frac = SMV_perc_disp.toFloat() / 100.0;
    }
    if(request->hasParam("max_sun")) {
      max_sun_disp = request->getParam("max_sun")->value();
      maximum_sunlight = max_sun_disp.toInt();
    }
    if(request->hasParam("sampling_period")) {
      sampling_period_disp = request->getParam("sampling_period")->value();
      sleep_duration_us = sampling_period_disp.toFloat() * 60 * 1000000;
    }
    if(request->hasParam("watering_duration")) {
      watering_duration_disp = request->getParam("watering_duration")->value();
      watering_duration_us = watering_duration_disp.toInt() * 1000000;
    }
    sensor_val_init = (SMV_frac > 0 && maximum_sunlight > 0 && sleep_duration_us >
0 && watering_duration_us > 0);
    request->send(200, "text/html", sensor_init_page, varRepl);
  });

  main_server.begin();

  while (!sensor_val_init) {
    //Serial.print("ESP AP IP Address: "); Serial.println(pot_ip_ap);
    delay(200);
  }
  digitalWrite(ledPin, LOW);
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////
void init_operation_server() {
  ////////////////////////////////////////////////////// Setup Section
  /*WiFiManager wm;
  bool res = wm.autoConnect("ESPDemo", "s3curep4ssw0rd");
  if(!res) {
    Serial.println("Failed to connect");
  }
  else {
    Serial.println("connected");
  }
  Serial.println("\nWiFi connected. IP: " + WiFi.localIP().toString());
  // Define webserver to handle incoming data in JSON format at /update
    server.on("/update", HTTP_POST, [](AsyncWebServerRequest *request) {}, NULL,
UpdateSensorData);
  // Debug method for changing server IP for local development
     server.on("/set-server-ip", HTTP_POST, [](AsyncWebServerRequest *request) {},
NULL, UpdateServerIP);
  server.begin();*/
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////
// SMV = soil moisture value
float get_SMV() {
  Serial.print("starting get_SMV function");
  digitalWrite(sms_power_pin,HIGH); Serial.println("513");
  delay(2500); Serial.println("514");
  float SMV = 0; int N=10; Serial.println("515");
  float SMV_temp = 0; Serial.println("516");
  for (int i = 0; i < N; i++) {
    SMV_temp = analogRead(sms_data_pin); Serial.println("518");
    SMV += SMV_temp; Serial.println("519");
    if (wet_SMV > SMV_temp){
      wet_SMV = SMV_temp; Serial.println("521");
    }
    if (dry_SMV < SMV_temp){
      dry_SMV = SMV_temp; Serial.println("524");
    }
    delay(500);
  }

  digitalWrite(sms_power_pin,LOW); Serial.println("529");
  current_SMV = SMV / N; Serial.println("530");
  Serial.print("SMV Value: "); Serial.println(SMV);
```

```
  return current_SMV;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////
void water_plant() {
  Serial.println("Enabling power to plant");
  unsigned long current_time = micros();
  digitalWrite(pump_ctrl_pin, HIGH);
  // (max limit - buffer) - current_time
  //buffer needed because
  unsigned long dist_from_limit = (4294967295 - 100000) - current_time;
  if(dist_from_limit > watering_duration_us) { //if watering duration will not exceed limit
    while((micros() - current_time) < watering_duration_us) {}
  }
  else{ // watering duration will exceed limit
    unsigned long stop_counter = watering_duration_us - dist_from_limit;
    while(micros() < stop_counter) {}
  }
  digitalWrite(pump_ctrl_pin, LOW);
  Serial.println("Disabling power to plant");
  Serial.print("start time: "); Serial.println(current_time);
  Serial.print("finish time: "); Serial.println(micros());
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////
float get_light_val () {
  // supply 3.3V to light sensor Vcc
  digitalWrite(light_sensor_pwr_pin,HIGH);
  delay(100);
  // configure light_sensor into continuous high resolution mode
  light_sensor.configure(BH1750::CONTINUOUS_HIGH_RES_MODE);
  delay(50);
  // light_val - return variable; stores total then divide by N to get average
  // N - number of measurements
  // temp - each individual measurement
  // fail_cnt - number of times measurementReady() fails
  float light_val = 0; int N=5;
  float temp = 0; int fail_cnt = 0;

  for(int i = 0; i < N; i++){
    if(i == N-1) {
      light_sensor.configure(BH1750::ONE_TIME_HIGH_RES_MODE);
    }
```

```
    while (!light_sensor.measurementReady()) {
      // waits at most 2.5s before returning 0 & outputting failure message to serial monitor
      delay(50); fail_cnt++;
      if(fail_cnt >= 50) {
        Serial.println("Measurement Ready failed for light sensor. Returning 0.");
        return 0;
      }
    }

    temp = light_sensor.readLightLevel(); //measurement

    // readLightLevel returns -1 or -2 if failure occurs
    if(temp > 0) {
      light_val += temp;
    }
    else { // failure
      // retry another measurement
      i = i-1;
    }
    delay(50);
  }

  light_val = light_val / N;
  Serial.print("Light Value: "); Serial.println(light_val);
  // power down light sensor
  //light_sensor.configure(BH1750::UNCONFIGURED);
  delay(50);
  // disable 3.3V power to light sensor
  digitalWrite(light_sensor_pwr_pin,LOW);
  return light_val;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////
float read_light_sensor () {
  int sunlight_threshold = 10000;
  float new_light_val = get_light_val();
  unsigned int sunlight_interval = sleep_duration_us/100;
    if ((new_light_val <= sunlight_threshold && light_val_lux >= sunlight_threshold) ||
(new_light_val >= sunlight_threshold && light_val_lux <= sunlight_threshold)) {
    total_sunlight_cnt += sunlight_interval / 2;
  }
  else if (new_light_val > sunlight_threshold && light_val_lux > sunlight_threshold) {
    total_sunlight_cnt += sunlight_interval;
  }
  light_val_lux = new_light_val;
```

```
  return light_val_lux;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////
void read_battery_level() {
  int raw = analogRead(battery_lvl_pin);
  Serial.print("Raw value at battery level pin: "); Serial.println(raw);
  float v_adc = raw * vRef / 4095.0;
  battery_lvl = v_adc * ((R1 + R2) / R2) * correctionFactor;
        Serial.print("Adjusted battery level value: "); Serial.print(battery_lvl);
Serial.println("V");
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////
void read_water_level() {
  Serial.println("Reading Water Level");
  //digitalWrite(wls_power_pin,HIGH);
  //delay(3000);
  int N = 9; int temp = 0;
  for(int i = 0; i < N; i++) {
    if(digitalRead(wls_power_pin) == HIGH) {
      temp++;
    }
    else {
      temp--;
    }
    delay(500);
  }
  water_level_low = (temp < 0);
  //digitalWrite(wls_power_pin,LOW);
  Serial.print("Water Level is Low: "); Serial.println(water_level_low);
}

void sensor_check () {
  read_battery_level();
      if(battery_lvl > 0.5){
        read_water_level();
        Serial.println("line 657");
        get_SMV();
        Serial.print("measured SMV: "); Serial.println(current_SMV);
        SMV_threshold = dry_SMV - (dry_SMV - wet_SMV)*SMV_frac;
        if(!water_level_low) {
          Serial.println("Water level not low");
          if(current_SMV > SMV_threshold) {
```

```
          water_plant();
        }
       }
       else {
        // send message to server/client to refill water tank
        Serial.println("Water level low");
       }
       read_light_sensor();
       if(total_sunlight_cnt >= maximum_sunlight) {
        // send message to server/client to cover plant
        Serial.println("Cover plant");
       }
      }
      else {
       // send message to server/client that battery is low
       Serial.println("Battery low");
      }
}

////////////////////////////////////////////////////////////////////////////////
void display_val() {
 Serial.println("IP address: "); Serial.println(pot_ip_sta);
 //Serial.println("Starting initialization for dry SMV");

 main_server.reset();

 digitalWrite(ledPin, HIGH);
 main_server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
  (*request).send(200,"text/html",sensor_disp_page, varRepl); } );
 main_server.on("/get", HTTP_GET, [](AsyncWebServerRequest *request){
  if(request->hasParam("submit_button")) {
   //dry_SMV = get_dry_SMV();
   //dry_SMV = 5000;
   leave_display = true;
   //Serial.println(dry_SMV);
  }
  request->send(200, "text/html", sensor_disp_page, varRepl);
 });

 main_server.begin();

 while (!leave_display) {
  delay(200);
 }
 digitalWrite(ledPin, LOW);
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////
void loop() {
  if(!initial_sensor_check_done) {
    Serial.println("Initial Sensor Check.");
    sensor_check();
    initial_sensor_check_done = true;
    leave_display = false;
    display_val();
  }

  Serial.println("Entering light-sleep mode.");
  Serial.println("Disabling all wakeup sources.");
  esp_sleep_disable_wakeup_source(ESP_SLEEP_WAKEUP_ALL);

  Serial.println("Enabling GPIO and timer wakeup sources.");
  esp_sleep_enable_timer_wakeup(sleep_duration_us);
  delay(20000);
  esp_light_sleep_start();

  esp_sleep_wakeup_cause_t wakeup_cause = esp_sleep_get_wakeup_cause();

  switch (wakeup_cause) {
    case ESP_SLEEP_WAKEUP_TIMER:
      Serial.println("\n\nwoke up from timer");
      sensor_check();
      break;
    //case ESP_SLEEP_WAKEUP_WIFI:
    //  Serial.println("woke up from wifi");
    //  break;
    default:
      break;
  }
  Serial.print("dry_SMV\t"); Serial.println(dry_SMV);
  Serial.print("wet_SMV\t"); Serial.println(wet_SMV);
  Serial.print("SMV_frac\t"); Serial.println(SMV_frac);
  Serial.print("maximum_sunlight\t"); Serial.println(maximum_sunlight);
  Serial.print("sleep_duration_us\t"); Serial.println(sleep_duration_us);
  Serial.print("watering_duration_us\t"); Serial.println(watering_duration_us);
  Serial.println("\n");
  delay(10000);

  leave_display = false;
  display_val();
```

```
  //sensor_val_init = false;
  //init_sensor_val();
}
```

Express Server

```
const express = require('express'); // Express for backend framework, handle incoming HTTP requests
const axios = require('axios'); // send outgoing requests to ESP32
const app = express();
const PORT = 3000;

let latest_data = null;
let esp_ip = null;

app.use(express.json());

// POST method for getting data from ESP32
app.post('/submit', (req, res) => {
  latest_data = req.body;
  // Keep IP of ESP32 so we can send data to it
  esp_ip = req.ip.replace(/^.*:/, '');
  console.log('Received data from ESP32 at IP:', esp_ip);
  console.log('Received JSON:', latest_data);
  res.sendStatus(200);
});

// GET method for showing recieved data in browser
app.get('/', (req, res) => {
  if (!latest_data) {
    return res.send('<h1>No data received yet</h1>');
  }

  let html = `<h1>Received Data</h1><ul>`;
  for (const [key, value] of Object.entries(latest_data)) {
    html += `<li><strong>${key}</strong>: ${value}</li>`;
  }
  html += `</ul>`;

  res.send(html);
});

// POST method to send data to ESP32
app.post('/send-to-esp', async (req, res) => {
  if (!esp_ip) {
    return res.status(500).json({ error: 'ESP32 IP unknown. No data received yet.' });
  }
```

```
  const payload = req.body;

  const cleanIP = esp_ip.trim();
  const url = `http://${cleanIP}:80/update`;
  console.log("Posting to:", url);

  const esp_result = await axios.post(url, payload)
  .then(resp => {
    console.log('ESP32 response:', resp.data);
  })
  .catch(err => {
    console.error('Error contacting ESP32:', err.message);
    return res.status(502).json({ error: 'Failed to contact ESP32' });
  });

  return res.status(200).json({ status: "sent", esp_result: "updated" });
});

// Start Server on given port
app.listen(PORT, '0.0.0.0', () => {
  console.log(`Server running at http://localhost:${PORT}`);
});
```

Microcontroller Integration Source File

```
#include <WiFi.h>
#include <WiFiManager.h>
#include <HTTPClient.h>
#include <ArduinoJson.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "connection_test.h"

// TODO: Cleaner setup code; less spaghetti
void setup() {
  WiFi.mode(WIFI_STA);
  Serial.begin(SERIAL_BAUD);
  pinMode(LED, OUTPUT);

  WiFiManager wm;

  bool res = wm.autoConnect("ESPDemo", "s3curep4ssw0rd");

  if(!res) {
    Serial.println("Failed to connect");
  }
```

```
  else {
    Serial.println("connected");
  }

  Serial.println("\nWiFi connected. IP: " + WiFi.localIP().toString());

  // Define webserver to handle incoming data in JSON format at /update
    server.on("/update", HTTP_POST, [](AsyncWebServerRequest *request) {}, NULL,
UpdateSensorData);

  // Debug method for changing server IP for local development
     server.on("/set-server-ip", HTTP_POST, [](AsyncWebServerRequest *request) {},
NULL, UpdateServerIP);

  server.begin();
}

// loop will continuously send latest available data to server
void loop() {
  // Using JSON allows us to send data originally in native structure / class easily
  // No need for pesky byte-level encode / decode functions :)
  StaticJsonDocument<HTTP_OK> doc;
  doc["data_float"] = data.data_float;
  doc["data_double"] = data.data_double;
  doc["data_int"] = data.data_int;

  String json;
  serializeJson(doc, json);
  Serial.println("Sending JSON: " + json);

  if (WiFi.status() == WL_CONNECTED) {
    HTTPClient http;
    http.begin(server_url);
    http.addHeader("Content-Type", "application/json");

    int httpResponseCode = http.POST(json);

    if (httpResponseCode > 0) {
      // blink if we send so I can see it
      Serial.printf("POST success: %d\n", httpResponseCode);
      blinkLED(LED);
    } else {
      // reverse blink if failed to send + write to console
      Serial.printf("POST failed: %s\n", http.errorToString(httpResponseCode).c_str());
      doubleBlinkLED(LED);
    }
```

```
    http.end();
  } else {
    // No wifi == continuous LED ON
    Serial.println("WiFi not connected.");
    digitalWrite(LED, HIGH);
  }

  // wait between POST reqs
  delay(TEN_S);
}

// Blink whej send is successfull
void blinkLED(int pin) {
  digitalWrite(pin, HIGH);
  delay(THREE_HUNDRED_MS);
  digitalWrite(pin, LOW);
}

// Backwards blink for send error
void doubleBlinkLED(int pin) {
  digitalWrite(pin, HIGH);
  delay(THREE_HUNDRED_MS);
  digitalWrite(pin, LOW);
  delay(THREE_HUNDRED_MS);
  digitalWrite(pin, HIGH);
}
```

Microcontroller Integration Header File

```
#include <WiFi.h>
#include <WiFiManager.h>
#include <HTTPClient.h>
#include <ArduinoJson.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include <functional>

// Keep track of server url in global cope
String server_url = "";

typedef enum {
  HTTP_OK = 200,
  HTTP_BAD_REQUEST = 400
} HTTPCodes_t;
```

```
// Pins
#define LED 2

// Locally Hosted server on ESP
#define SERVER_PORT 80
AsyncWebServer server(SERVER_PORT);

// Time delays
#define TEN_S 10000
#define THREE_HUNDRED_MS 300
#define FIVE_HUNDRED_MS 500

// Serial baud rate
#define SERIAL_BAUD 115200

// Generic Sensor Data struct
struct SensorData {
  float data_float;
  double data_double;
  int data_int;
} data = {0, 0, 0};

// Definition of function for handling HTTP request to update sensor data
std::function<void(AsyncWebServerRequest*, uint8_t*, size_t, size_t, size_t)>
UpdateSensorData =
  [](AsyncWebServerRequest *request, uint8_t *data_recv, size_t len, size_t index, size_t
total)
  {
    StaticJsonDocument<HTTP_OK> doc;
    DeserializationError err = deserializeJson(doc, data_recv);
    if (err) {
          request->send(HTTP_BAD_REQUEST,  "application/json",  "{\"error\":\"Invalid
JSON\"}");
      return;
    }

     // TODO: More Scalable way of defining data? So we don't need to look in 3 places
across
    // 2 differen projects (server & esp) to change one field
    if (doc.containsKey("data_float")) data.data_float = doc["data_float"];
    if (doc.containsKey("data_double")) data.data_double = doc["data_double"];
    if (doc.containsKey("data_int")) data.data_int = doc["data_int"];

    Serial.println("Received data via POST:");
    serializeJsonPretty(doc, Serial);
    Serial.println();
```

```
  request->send(HTTP_OK, "application/json", "{\"status\":\"updated\"}");
 };

// Definition of function for handling HTTP request to update server IP address
std::function<void(AsyncWebServerRequest*, uint8_t*, size_t, size_t, size_t)>
UpdateServerIP =
 [](AsyncWebServerRequest *request, uint8_t *data_recv, size_t len, size_t index, size_t
total)
 {
  StaticJsonDocument<HTTP_OK> doc;
  DeserializationError err = deserializeJson(doc, data_recv);
  if (err) {
        request->send(HTTP_BAD_REQUEST, "application/json", "{\"error\":\"Invalid
JSON\"}");
    return;
  }

  if (!doc.containsKey("server_ip")) {
       request->send(HTTP_BAD_REQUEST, "application/json", "{\"error\":\"Missing
'server_ip'\"}");
    return;
  }

  const char* ip = doc["server_ip"];
  IPAddress parsed_ip;
  if (!parsed_ip.fromString(ip)) {
     request->send(HTTP_BAD_REQUEST, "application/json", "{\"error\":\"Invalid IP
format\"}");
    return;
  }

  String raw_server_url = String(ip);
  Serial.print("Server IP set to: ");
  Serial.println(raw_server_url);
  server_url = "http://" + raw_server_url + ":3000/submit";
  Serial.println(server_url);

  request->send(HTTP_OK, "application/json", "{\"status\":\"server_url updated\"}");
 };
```

Appendix E - ChatGPT prompts and outcomes

1. "Write Arduino-compatible code for an ESP32 that reads soil moisture data every 15 minutes and sends it via HTTP to a custom web dashboard."

```
#include <WiFi.h>
#include <HTTPClient.h>

const char* ssid = "YOUR_SSID";
const char* password = "YOUR_PASSWORD";

const int sensorPin = 34; // Analog pin for soil moisture sensor
const char* serverURL = "http://your-dashboard-endpoint.com/upload";

unsigned long lastReadTime = 0;
const unsigned long interval = 15 * 60 * 1000; // 15 minutes

void setup() {
  Serial.begin(115200);
  WiFi.begin(ssid, password);

  Serial.print("Connecting to WiFi");
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("\nWiFi connected");
}

void loop() {
  unsigned long currentTime = millis();

  if (currentTime - lastReadTime >= interval || lastReadTime == 0) {
    int moistureValue = analogRead(sensorPin);
    Serial.println("Soil Moisture: " + String(moistureValue));

    if (WiFi.status() == WL_CONNECTED) {
      HTTPClient http;
      http.begin(serverURL);
      http.addHeader("Content-Type", "application/json");

      String payload = "{\"moisture\":" + String(moistureValue) + "}";
      int httpResponseCode = http.POST(payload);

      Serial.print("HTTP Response Code: ");
      Serial.println(httpResponseCode);

      http.end();
    } else {
      Serial.println("WiFi not connected");
```

```
    }

    lastReadTime = currentTime;
  }

  delay(1000); // Small delay to avoid flooding loop
}
```

Gemini Response

```
#include <WiFi.h>
#include <HTTPClient.h>
#include <ArduinoJson.h> // For easily creating JSON payloads

// --- WiFi Credentials ---
// Replace with your WiFi network SSID
const char* WIFI_SSID = "YOUR_WIFI_SSID";
// Replace with your WiFi network password
const char* WIFI_PASSWORD = "YOUR_WIFI_PASSWORD";

// --- Web Dashboard Server Details ---
// Replace with your web dashboard server's IP address or hostname
const char* WEB_DASHBOARD_HOST = "your.web.dashboard.com";
// Replace with the port your web server is listening on (e.g., 80 for HTTP, 443 for HTTPS)
const int WEB_DASHBOARD_PORT = 80;
// Replace with the specific path on your server where you want to send the data
const char* WEB_DASHBOARD_PATH = "/api/soil_data";

// --- Soil Moisture Sensor Pin ---
// Connect your analog soil moisture sensor's data output to this GPIO pin on your ESP32.
// Common analog pins on ESP32 are GPIO 32, 33, 34, 35, 36, 39.
// GPIO 34 is a good choice as it's an ADC1 pin and typically available.
const int SOIL_MOISTURE_PIN = 34; // Example: GPIO34 for analog read

// --- Deep Sleep Configuration ---
// Factor to convert microseconds to seconds for deep sleep
#define uS_TO_S_FACTOR 1000000ULL
// Time ESP32 will sleep (in seconds) - 15 minutes * 60 seconds/minute
#define TIME_TO_SLEEP  (15 * 60)

// --- Device Identifier ---
// A unique ID for this specific ESP32 sensor. Useful if you have multiple sensors.
const char* DEVICE_ID = "esp32_soil_sensor_001";

// Function to connect to WiFi
void connectToWiFi() {
```

```
  Serial.print("Connecting to WiFi: ");
  Serial.println(WIFI_SSID);

  WiFi.begin(WIFI_SSID, WIFI_PASSWORD);

  // Wait for connection
  int attempts = 0;
  while (WiFi.status() != WL_CONNECTED && attempts < 20) {
    delay(500);
    Serial.print(".");
    attempts++;
  }

  if (WiFi.status() == WL_CONNECTED) {
    Serial.println("\nWiFi connected!");
    Serial.print("IP Address: ");
    Serial.println(WiFi.localIP());
  } else {
    Serial.println("\nFailed to connect to WiFi. Retrying after deep sleep.");
    // In a real application, you might want to handle this more robustly
    // For this example, it will just try again after waking from deep sleep.
  }
}

// Function to read soil moisture data
int readSoilMoisture() {
  // Read the analog value from the sensor pin
  // ESP32 ADC resolution is 12-bit by default (0-4095)
  // The actual moisture range depends on your specific sensor calibration.
  int moistureValue = analogRead(SOIL_MOISTURE_PIN);
  Serial.print("Raw Soil Moisture Value: ");
  Serial.println(moistureValue);
  return moistureValue;
}

// Function to send data via HTTP POST
void sendDataToDashboard(int moisture) {
  if (WiFi.status() == WL_CONNECTED) {
    HTTPClient http;

    // Construct the full URL
    String serverPath = "http://"; // Use "https://" for HTTPS
    serverPath += WEB_DASHBOARD_HOST;
    serverPath += ":";
    serverPath += WEB_DASHBOARD_PORT;
    serverPath += WEB_DASHBOARD_PATH;
```

```
    Serial.print("Sending data to: ");
    Serial.println(serverPath);

    http.begin(serverPath); // Specify the URL

    // Set HTTP header for content type
    http.addHeader("Content-Type", "application/json");

    // Create JSON payload using ArduinoJson library
    StaticJsonDocument<200> doc; // Allocate a JSON document
    doc["device_id"] = DEVICE_ID;
    doc["moisture"] = moisture;

    String jsonPayload;
    serializeJson(doc, jsonPayload); // Serialize JSON document to string

    Serial.print("JSON Payload: ");
    Serial.println(jsonPayload);

    // Send HTTP POST request
    int httpResponseCode = http.POST(jsonPayload);

    if (httpResponseCode > 0) {
      Serial.print("HTTP Response code: ");
      Serial.println(httpResponseCode);
      String response = http.getString();
      Serial.println("Server Response: ");
      Serial.println(response);
    } else {
      Serial.print("Error code: ");
      Serial.println(httpResponseCode);
      Serial.println(http.errorToString(httpResponseCode));
    }
    http.end(); // Free resources
  } else {
    Serial.println("WiFi not connected. Cannot send data.");
  }
}

void setup() {
  Serial.begin(115200); // Initialize serial communication for debugging
  pinMode(SOIL_MOISTURE_PIN, INPUT); // Set the sensor pin as input

  // Configure deep sleep
  // This will set the ESP32 to wake up after TIME_TO_SLEEP seconds
```

```
  esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * uS_TO_S_FACTOR);

  Serial.println("Wake up from deep sleep. Starting measurement cycle...");

  // 1. Connect to WiFi
  connectToWiFi();

  // 2. Read soil moisture
  int moistureData = readSoilMoisture();

  // 3. Send data to dashboard
  sendDataToDashboard(moistureData);

  // 4. Go to deep sleep
  Serial.print("Entering deep sleep for ");
  Serial.print(TIME_TO_SLEEP);
  Serial.println(" seconds...");
  esp_deep_sleep_start(); // Enter deep sleep
}

void loop() {
  // This loop function will typically not be reached if deep sleep is used,
  // as the ESP32 restarts from setup() after waking up.
  // However, it's good practice to keep it for standard Arduino sketches.
  // If you were not using deep sleep, you would place your timing logic here.
}
```

How to Use This Code:

1. **Install ESP32 Board Support:** If you haven't already, open your Arduino IDE, go to File > Preferences, and add https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json to the "Additional Board Manager URLs." Then, go to Tools > Board > Board Manager, search for "esp32", and install the "esp32 by Espressif Systems" package.
2. Install Libraries:
    - HTTPClient: This library is usually included with the ESP32 core.
    - ArduinoJson: Go to Sketch > Include Library > Manage Libraries..., search for "ArduinoJson", and install the latest version (by Benoit Blanchon).
3. Hardware Connections:
    - Connect the VCC pin of your soil moisture sensor to a 3.3V or 5V pin on the ESP32 (check your sensor's voltage requirements).
    - Connect the GND pin of your soil moisture sensor to a GND pin on the ESP32.
    - Connect the Analog Output (A0 or DO/AO) pin of your soil moisture sensor to GPIO 34 on the ESP32. If your sensor has both analog (AO) and digital (DO) outputs, use the analog one.

4. Configuration:
    - WiFi Credentials: Replace "YOUR_WIFI_SSID" and "YOUR_WIFI_PASSWORD" with your actual Wi-Fi network name and password.
    - Web Dashboard Details:
        - WEB_DASHBOARD_HOST: Enter the IP address or domain name of your server (e.g., "192.168.1.100" or "my.dashboard.com").
        - WEB_DASHBOARD_PORT: Set the port your web server listens on (e.g., 80 for standard HTTP).
        - WEB_DASHBOARD_PATH: Specify the API endpoint on your server where the data should be sent (e.g., "/data_receiver").
    - SOIL_MOISTURE_PIN: Ensure this matches the GPIO pin you connected the sensor to.
    - DEVICE_ID: Assign a unique identifier for your sensor if you plan to deploy multiple.
5. Upload: Select your ESP32 board (Tools > Board > ESP32 Arduino > ESP32 Dev Module or similar) and the correct COM port, then click "Upload."

How the Code Works:

- Deep Sleep for Power Saving: The ESP32 is configured to wake up every 15 minutes using a built-in timer. This is highly efficient for battery-powered applications.
- WiFi Connection: Upon waking, the ESP32 attempts to connect to your specified Wi-Fi network.
- Soil Moisture Reading: It then reads the analog value from the SOIL_MOISTURE_PIN. The raw value will be between 0 (dry, depending on sensor) and 4095 (wet, depending on sensor). You might need to calibrate this range based on your sensor and environment.
- HTTP POST: The collected data (device ID and moisture value) is formatted as a JSON string using the ArduinoJson library. This JSON payload is then sent via an HTTP POST request to your web dashboard.
- Server Response: The ESP32 prints the HTTP response code and any message received from your server for debugging.
- Sleep Again: After sending the data, the ESP32 immediately enters deep sleep again for another 15 minutes.

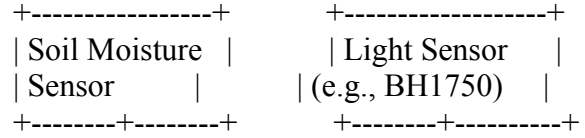
This setup provides a robust and power-efficient way to monitor soil moisture and send data to your custom dashboard.

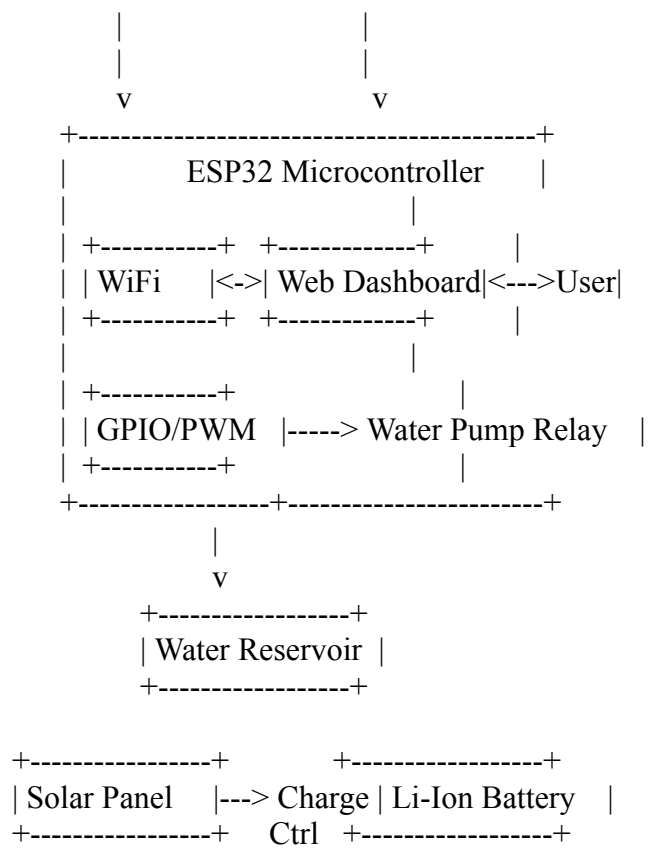
“Make a chart comparing the two outputs”

Feature / Focus	GPT-4o	Gemini 1.5 Pro
Basic Functionality	Moisture reading, timing, and HTTP POST handled cleanly	Same core features, handled correctly
Power Management	Not included	Uses deep sleep for low-power operation
JSON Handling	Manual string formatting	Uses ArduinoJson for clean, structured payload
Code Structure	Flat script, no functions	Modular functions with clear purpose
Comments & Readability	Simple and clean, not overly detailed	Well-commented with context for each step
Setup Instructions	None	Includes notes on GPIO pins, server paths, and payload
Error Handling / Debugging	Basic output only	Includes Wi-Fi retry logic and HTTP response handling
Deployment Readiness	Good for prototyping	Closer to plug-and-play for real deployment
Flexibility for Iteration	Responds well to follow-ups	Responds well and adds helpful suggestions
Best Fit For	Quick testing and getting started	Deployment on a solar-powered field unit like CHANOS

2. "Give me a block diagram and explanation of how the sensors, ESP32, solar power, battery, and web dashboard work together in a smart irrigation system."

Here’s a block diagram and accompanying explanation of how your smart irrigation system components—**sensors**, **ESP32**, **solar power system**, **battery**, and **web dashboard**—work together.





Component	Role
Soil Moisture Sensor	Detects current soil water level; sends analog/digital signal to ESP32.
Light Sensor (optional)	Monitors sunlight exposure to help optimize watering based on time of day.
ESP32	Central controller. Reads sensor values, controls pump, connects to web dashboard via WiFi, sends/receives data.
Relay Module	Acts as a switch. When triggered by ESP32, it powers the water pump.
Water Pump	Delivers water to plants when soil moisture is below a set threshold.
Solar Panel	Provides renewable energy. Connected to charge controller to manage battery charging safely.
Battery (Li-Ion)	Stores energy from the solar panel to power the system at night or in cloudy conditions.

Web Dashboard	User interface to monitor sensor data in real time, view logs, and optionally adjust watering thresholds remotely.

Operational Flow

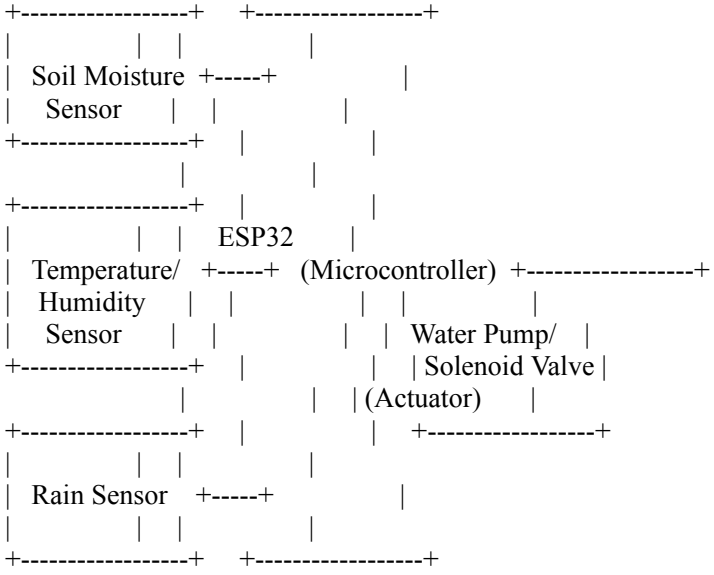
1. **Power Supply**: Solar panel charges the battery via a charge controller.
2. **Sensor Readings**: ESP32 periodically reads values from soil moisture and light sensors.
3. **Decision Making**: If soil is dry (below threshold), the ESP32 triggers the relay to run the pump.
4. **Data Logging**: ESP32 sends sensor readings to the web dashboard via WiFi.
5. **Remote Monitoring**: User accesses dashboard to view plant status or change system behavior (e.g., watering interval or thresholds).
6. **Energy Efficiency**: The ESP32 can enter deep sleep between readings to conserve power.

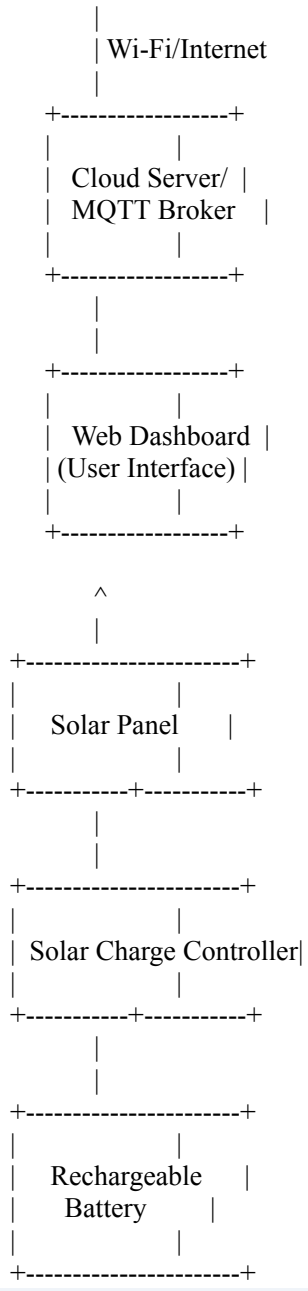
Response from Gemini

Block Diagram

A smart irrigation system using an ESP32, solar power, and a web dashboard integrates various components to automate and optimize watering. Here's a block diagram and an explanation of how these elements work together:

Block Diagram





Explanation of Components and Their Interaction:

1. Sensors:

- **Soil Moisture Sensor:** This is a crucial sensor that measures the volumetric water content in the soil.[1] It typically outputs an analog value that the ESP32 converts into a readable moisture percentage.
    - **How it works:** When the soil is dry, the sensor's resistance is high, resulting in a low conductivity and a high analog reading. As the soil becomes wet, resistance decreases, leading to higher conductivity and a lower analog reading.

- **Temperature/Humidity Sensor (e.g., DHT11/DHT22):** Measures ambient air temperature and humidity. This data can be used to calculate evapotranspiration rates, further optimizing watering schedules.
    - **How it works:** These sensors use thermistors and capacitive humidity sensors to provide digital temperature and humidity readings.
- **Rain Sensor:** Detects the presence of rainfall. This is vital for preventing overwatering and conserving water.
    - **How it works:** Many rain sensors work by detecting water bridging conductive traces, changing their resistance or capacitance, which the ESP32 can then interpret.

2. ESP32 (Microcontroller):

The ESP32 is the brain of the system, acting as the central processing unit and communication hub.

- **Data Acquisition:** It receives data from all connected sensors (soil moisture, temperature/humidity, rain). It has built-in Analog-to-Digital Converters (ADCs) to read analog sensor outputs and digital pins to interface with digital sensors.
- **Decision Making (Logic):** Based on the collected sensor data and pre-programmed thresholds or algorithms, the ESP32 makes decisions about when and how much to irrigate. For example:
    - If soil moisture drops below a certain threshold AND it's not raining, activate watering.
    - Consider temperature and humidity to adjust watering duration (e.g., water more during hot, dry periods).
- **Actuator Control:** It sends signals to control the water pump or solenoid valve, turning the water flow ON or OFF.
- **Connectivity (Wi-Fi):** The ESP32's integrated Wi-Fi module allows it to connect to your home network or a mobile hotspot. This is crucial for sending sensor data to the cloud and receiving commands from the web dashboard.
- **Power Management:** The ESP32 is designed to be low-power, which is essential for solar-powered applications. It can be put into deep sleep modes to conserve energy when not actively performing tasks.

3. Solar Power System:

This system provides an independent and sustainable power source for the entire irrigation setup.

- **Solar Panel:** Converts sunlight into electrical energy (DC current). The size and wattage of the panel depend on the power requirements of the system and local sunlight conditions.
- **Solar Charge Controller:** This is an essential component that regulates the voltage and current from the solar panel to the battery.
    - **Functions:** Prevents overcharging and deep discharging of the battery, extends battery life, and ensures a stable power supply to the ESP32 and other components.
- **Rechargeable Battery:** Stores the electrical energy generated by the solar panel. This allows the system to operate continuously, even during the night or on cloudy days when the solar panel is not generating power.

    - **Types:** Lithium-ion (Li-ion) or Lithium Polymer (LiPo) batteries are common choices due to their energy density and relatively long lifespan.[2]
    -

4. Water Pump / Solenoid Valve (Actuator):

- This is the device that physically controls the water flow to the plants.[3]
-
- **Water Pump:** Used for drawing water from a reservoir or well and delivering it to the irrigation system.[4]
-
- **Solenoid Valve:** An electrically controlled valve that opens or closes a water line.[5] Commonly used to control water from a main supply or a pressurized system.[6]
-
- **How it works:** The ESP32 sends a signal (typically through a relay module for higher current devices) to activate or deactivate the pump/valve, starting or stopping irrigation.[7]
-

5. Cloud Server / MQTT Broker:

- **Data Transmission:** The ESP32 sends sensor data (soil moisture, temperature, humidity, rain status, pump status) to a cloud server or an MQTT (Message Queuing Telemetry Transport) broker over Wi-Fi. MQTT is a lightweight messaging protocol well-suited for IoT devices.
- **Data Storage:** The cloud server stores this historical data, allowing for analysis and long-term monitoring.
- **Command Reception:** The cloud server/MQTT broker can also receive commands from the web dashboard (e.g., manually turn on/off the pump, change watering schedules) and relay them to the ESP32.

6. Web Dashboard (User Interface):

- **Data Visualization:** A web-based application (accessible from any web browser on a computer or smartphone) that displays real-time and historical sensor data in an easy-to-understand format (graphs, charts, numerical values).
- **Remote Control:** Allows the user to remotely monitor and control the irrigation system. This includes:
    - Manually turning the water pump ON/OFF.
    - Setting or modifying watering schedules and thresholds (e.g., desired soil moisture percentage).
    - Viewing system status (battery level, last watering time).
    - Receiving alerts or notifications (e.g., low battery, system error).
- **User Interaction:** Provides an intuitive interface for users to interact with their smart irrigation system without needing to be physically present.

Overall Workflow:

1. **Sensing:** The sensors continuously monitor environmental conditions (soil moisture, temperature, humidity, rain).
2. **Data Transmission to ESP32:** Sensor data is sent to the ESP32.
3. **Local Processing and Decision:** The ESP32 processes the sensor data and, based on pre-programmed logic and user-defined settings, decides whether irrigation is needed.
4. **Cloud Communication:** The ESP32 sends the sensor data to the cloud server/MQTT broker via Wi-Fi.
5. **Web Dashboard Interaction:** The user accesses the web dashboard to view the data, monitor system status, and configure settings or manually control the system.
6. **Command Execution:** If the ESP32 decides to irrigate (or receives a command from the web dashboard), it activates the water pump/solenoid valve.
7. **Power Management:** The solar power system charges the battery during the day, providing continuous power to the ESP32 and sensors. The ESP32 utilizes low-power modes to conserve energy.
8. **Feedback Loop:** The system continuously monitors the soil moisture after watering, ensuring that the desired moisture level is achieved and stopping irrigation once it's reached.

This integrated system creates an autonomous, efficient, and user-friendly smart irrigation solution that conserves water and promotes healthier plant growth.

3. **“For a solar-powered smart irrigation system like CHANOS, evaluate the different types of small pumps that could be used. Consider options such as submersible pumps, peristaltic pumps, diaphragm pumps, and solenoid valves. Compare them based on power consumption, flow control, ability to run dry, ease of integration with an ESP32, and reliability in unattended operation. The pump must be battery-compatible, capable of moving water from a small reservoir, and operate briefly every few hours. Which type would be most appropriate, and why?”**

Understood. Here's the natural, context-free response as if you had just posted the prompt:

For a solar-powered smart irrigation system, each pump type brings different tradeoffs depending on the design goals. Since the system is battery-powered, draws water from a small reservoir, and only operates briefly every few hours, efficiency, reliability, and simplicity are key.

Submersible pumps are common in DIY irrigation systems. They're inexpensive, compact, and easy to integrate using a basic transistor or relay. These pumps sit directly in the water and typically operate at 5V or 6V. They can move a decent volume of water quickly, but they usually don’t tolerate dry running well—running them dry can burn out the motor. There’s also little to no flow control unless additional valves or sensors are added.

Peristaltic pumps work by squeezing fluid through tubing using rollers. Because the fluid only contacts the tubing, they’re clean and can run dry without damage. They offer slower, more precise flow control, which is useful if dosing exact amounts of water.

However, they tend to be noisier and may be bulkier depending on the model. Power draw is typically low, and they're also simple to switch on and off with an ESP32 using a MOSFET.
Diaphragm pumps use a pulsing diaphragm to create suction and pressure. They tend to be self-priming and more powerful than submersibles or peristaltics. However, they draw more current and are less efficient for small, low-volume tasks. Unless pressure is a requirement, they're likely overkill for this type of system.
Solenoid valves are useful if the system already has a pressurized water supply, like a household hose or gravity-fed tank. They're fast-switching and simple to control, but they don't move water themselves—they just allow flow. They also draw significant current when actuated and usually aren't ideal for battery-based setups unless paired with a latching valve, which complicates the design.
Based on these tradeoffs, a peristaltic pump is often the best fit for this kind of setup. It can run dry safely, is easy to control, uses relatively little power, and delivers consistent flow in small amounts. Submersibles can also work if low cost is the priority and the reservoir is never allowed to run dry.

Response from Gemini

Given the design goals of efficiency, reliability, and simplicity for a battery-powered system drawing from a small reservoir and operating briefly every few hours, your assessment is spot on.

Peristaltic Pumps: The Ideal Choice

You're right that peristaltic pumps often emerge as the best fit for this specific application. Here's why they align so well with your stated goals:

- Reliability (Dry Run Safety): The ability to run dry without damage is a massive advantage. In a small reservoir, it's easy for water levels to drop, and a burnt-out submersible pump would significantly impact the system's reliability and maintenance.
- Efficiency & Control: Their design offers slower, more precise flow control. This means less wasted water and more accurate delivery, which is ideal for "brief operation every few hours" and potentially dosing exact amounts. Their low power draw also directly supports the battery-powered nature of the system.
- Simplicity of Integration: You correctly noted they're simple to switch on and off with an ESP32 using a MOSFET, maintaining the desired simplicity in the control electronics.

Submersible Pumps: A Viable, Cost-Effective Alternative

While peristaltic pumps offer superior features for this scenario, submersible pumps are indeed a strong contender if low cost is the priority. As you pointed out, their primary drawback is the risk of dry running, which requires careful management of the reservoir level or the addition of a water level sensor to protect the pump.

Diaphragm Pumps & Solenoid Valves: Less Suitable for this Design

Your analysis of diaphragm pumps as "overkill" due to their higher current draw and the unsuitability of solenoid valves for battery-based setups without an existing pressurized

supply is accurate. They introduce complexities and power demands that go against the core design principles for this specific system.

Conclusion

For a solar-powered, battery-operated smart irrigation system focusing on efficiency, reliability, and simplicity with brief, intermittent operation, the peristaltic pump stands out as the most suitable choice. Its ability to run dry, precise flow control, and low power consumption directly address the key requirements. However, if budget is extremely tight, a submersible pump can be used, provided adequate measures are in place to prevent dry running.

This detailed evaluation provides clear guidance for anyone designing such a system!

4. **"Design a simple and low-power water level sensor circuit that can detect when a reservoir is low, and interface it directly with an ESP32. Recommend components, explain the circuit, and describe how the ESP32 should read the signal."**

A simple and low-power water level detection circuit can be built using two exposed conductive probes (such as stainless steel wires) placed at the minimum water level inside the reservoir. When water is present, it completes a circuit between the two probes due to its conductivity. When the water drops below the probes, the circuit breaks, indicating a low water level.

Components:

- 2 stainless steel probe wires or screws
- 10kΩ pull-up resistor
- NPN transistor (e.g., 2N2222 or S8050)
- 1kΩ base resistor
- GPIO pin on the ESP32

Circuit Description:

1. The two probes are placed vertically in the reservoir so that both are submerged when water is at a safe level.
2. One probe connects to 3.3V through a 10kΩ resistor (pull-up).
3. The other probe connects to the base of the NPN transistor through a 1kΩ resistor.
4. The emitter of the transistor goes to ground.
5. The collector connects to a GPIO pin on the ESP32, with an internal pull-up enabled.

Operation:

- When water is present between the probes, a small current flows into the base of the transistor, turning it on and pulling the GPIO pin low.

- When water drops below the probes, the circuit is open, the transistor stays off, and the GPIO pin remains high due to the pull-up.

ESP32 Logic:

- Configure the GPIO pin as an input with pull-up enabled.

- Read the pin periodically: LOW means water is present; HIGH means low water level.

- Optionally, debounce or add a delay to avoid false triggers from small ripples or splashes.

This setup uses very little power, has no moving parts, and can be polled infrequently to reduce energy consumption further.

Gemini's reponse

That's an excellent and indeed **simple, low-power, and effective** design for a water level detection circuit for your smart irrigation system! It perfectly addresses the need to protect a submersible pump from dry running without adding significant complexity or power consumption.

Let's break down why this design is well-suited for your application and recap its benefits:

Analysis of the Proposed Circuit

1. Simplicity and Low Cost:

- Uses common, inexpensive components: a couple of wires/screws, resistors, and a standard NPN transistor.
- No moving parts, which means greater reliability and less wear and tear compared to float switches.

2. Low Power Consumption:

- The circuit only draws a small current when water is present (to turn on the transistor). When dry, the current draw is practically negligible.
- As you noted, the ESP32 can poll the GPIO pin infrequently (e.g., once every few minutes, or even just before attempting to run the pump), further minimizing overall power consumption. This is crucial for a solar-powered, battery-dependent system.

3. Reliability:

- The use of conductive probes is robust for water detection. Stainless steel is a good choice for corrosion resistance.
- The transistor acts as a switch, providing a clean digital signal to the ESP32, isolating the sensitive microcontroller from potential issues with direct probe connections.

4. Ease of Integration with ESP32:

- Connects directly to a standard GPIO pin.

- The logic (LOW for water, HIGH for low water) is straightforward for the ESP32 firmware to interpret.

How it Addresses the Submersible Pump Limitation

This circuit directly solves the primary drawback of using a submersible pump:

- **Dry-Run Protection:** By placing the probes at the minimum safe water level, the ESP32 can detect when the water level drops below this point.
- **Preventing Damage:** When a low water level is detected (GPIO pin HIGH), the ESP32 can be programmed to:
    - Prevent the pump from starting.
    - Immediately shut off the pump if it's currently running.
    - Send an alert to the web dashboard indicating a low reservoir.

Minor Considerations / Enhancements (Optional):

- **Probe Placement:** Ensure the probes are positioned to avoid false readings from splashes or minor surface ripples. Placing them slightly apart and ensuring they are below the normal operating water level but above the critical "dry-run" level is key.
- **Corrosion/Fouling:** Over very long periods, mineral deposits or algae might build up on the probes, potentially affecting conductivity. Stainless steel is good, but periodic cleaning might be necessary in some water conditions.
- **Debouncing in Software:** As you mentioned, debouncing the GPIO reading in the ESP32 code is a good practice. This can be done by taking multiple readings over a short period (e.g., 50-100ms) and confirming the state change before reacting. This prevents spurious triggers from small water movements.

Conclusion

This **conductive probe circuit with an NPN transistor** is an excellent and highly suitable addition to your solar-powered smart irrigation system, especially if you opt for an inexpensive submersible pump. It provides reliable, low-power water level detection, effectively protecting your pump and enhancing the overall robustness of the system.
Well thought out!

Block Diagram (Text Format)

```
[Sensors]
   ↓
[ESP32 Microcontroller] ↔ [Wi-Fi Module]
   ↓                          ↓
[Battery Management System]   → [Web Dashboard]
   ↑
[Solar Panel] → [Charge Controller] → [Battery]
```

Explanation:

1. **Sensors** (e.g., soil moisture, light level, water level) collect data from the environment.
2. The **ESP32 microcontroller** reads the sensor values, processes them, and determines if any action is needed (e.g., triggering a water pump).
3. The **ESP32** is powered by a **battery**, which is kept charged by a **solar panel** connected through a **charge controller**. This setup ensures continuous operation without reliance on external power.
4. The ESP32 connects to the internet via its built-in **Wi-Fi module**, allowing it to transmit data to a **web dashboard**.
5. The **web dashboard** displays real-time sensor values and system status, enabling remote monitoring and control.