

Department of Design Engineering and Mathematics

Middlesex University



IoT System for Environmental Data Collection

Patrick J Nkwocha

Supervisor: Quoc-Tuan Vien

PDE3400 Major Project

BEng Robotic Engineering

July 2023

Abstract

The overall idea of this project is to explore the various ways in which sensor networks can be used to monitor environmental conditions, and henceforth support precision agriculture practices. The goal of the project is to develop a sensor network which monitors environmental parameters such as, humidity, barometric pressure, temperature, and windspeed, with the measurements elicited being relayed back to the user via the IoT cloud for mobile access, and an LCD screen for indoor access. After deriving the sensor readings for each day, the intention is to store key characteristics into a dataset and then predict the corresponding readings for the next day, therefore serving as a forecast.

Table of contents

1.	Introduction	4
1.1.	Developing a Use-case	4
1.2.	Objectives and Requirements specification	5
1.3.	Initial SWOT Analysis	6
2.	Background Research	7
2.1.	Existing Systems	7
2.2.	Issues and Challenges	8
3.	Product Design Specification	12
4.	Component Selection and Technology Research	16
4.1.	Initial Hardware Setup	16
4.2.	Component Research and Selection	17
4.3.	Programming Approach and Methodology	23
5.	Design Documentation	24
5.1.	Hardware Setup (2nd Iteration)	24
5.2.	Functionality Checklists (Pass Tests)	25
5.3.	System Flowcharts	26
5.4.	GUI Design	29
5.5.	Casings CAD Design	32
5.6.	Circuit Design: Schematics	34
6.	Implementation and Testing	28
6.1.	Hardware Implementation and Testing	39
6.2.	Software Implementation and Testing	39

7. Further Developments	67
8. Conclusions	68
Bibliography	69
Appendix	71

1. Introduction

It is important to note that this project has not been devised with the intention of creating a system ready for commercial use. However, with the existence of a problem this project aims to somewhat mitigate that problem by exploring how sensors can be used to help abstract environmental data in a way which is legible for the user. Also, this project involves an investigation in how data science can be applied to make predictions relevant to this use case of precision agriculture. Overall, this project will involve an amalgamation of electronic sensors, hardware, machine learning, and networking.

1.1. Developing a Use-case

Initially, as advised by supervisor I decided to come up with a use case scenario to help specify my requirements and deliverables. The underlining objective was to design an IoT system used for monitoring environmental characteristics and extracting environmental data. With just the title, at this point the project appeared to be a data collection system (data stream). To expand on this, the next questions to ask would be, why is one collecting the data? And what is the data going to be used for? The following use-case scenario allowed me to identify my user, the problem at hand, and specify primary goals.

“John is a gardener growing various salads and vegetables in his garden. However, with other obligations taking up his time he has encountered several issues when going to reap his crops. John is consistent with tending to his crops, watering them consistently, weeding and reaping but he finds that often the cycle he uses to tend to his crops does not correspond with recent weather (environmental) patterns. For example, when watering his plants, he finds that sometimes he waters his plants unknowing that rainfall is to occur the same day resulting in overwatering, which leads to poor root health. He also finds that when the weather has been particularly warm and dry (no rainfall) for long periods of time the soil moisture is adversely affected which diminishes yields.”

The intervals between when the user tends to their garden is when the problem seemed to manifest. Without knowing the current and future weather the user cannot cater their actions to their garden appropriately. To combat this problem, development of a system to monitor environmental factors (weather) and predict weather would be useful. From this use-case scenario I could establish an overview of project objectives and specify my requirements.

1.2. Objectives and Requirements specification

Following observations of my use-case scenario I was able to break down the solutions to the problem into bullet points.

To solve the problem at hand, our user required a system that was successful at:

- ***Monitoring environmental (weather) characteristics.***
- ***Predicting weather (relevant characteristics).***
- ***Relaying information to the user.***

From these bullet points I could establish a detailed product specification where I was able to expand on each one further.

1.2.1. *Objective 1: Monitoring the environment:*

The system has a means of extracting environmental data, meaning the present characteristics of the place it finds itself operating in e.g., John's Garden.

The expected outcomes of achieving this objective include:

- Improved understanding of the current environmental conditions in John's garden.
- Real-time monitoring of wind speed, humidity, soil moisture, and temperature.
- Enhanced ability to make informed decisions regarding watering, weeding, and other gardening activities based on accurate environmental data.

1.2.2. *Objective 2: Predicting the weather:*

The system will have a means of predicting the weather. The expected outcomes of achieving this objective include:

- Accurate weather predictions specific to John's garden, considering the environmental characteristics and historical data collected.
- Improved ability to plan gardening activities based on reliable weather forecasts.
- Mitigation of overwatering or insufficient watering issues by aligning watering schedules with predicted rainfall.

1.2.3. Objective 3: Relaying information to the user:

The system will have means of displaying the relevant collected data - see points 1 and 2, in a legible manner. The expected outcomes of achieving this objective include:

- Clear presentation of environmental data, weather predictions, and other relevant information to John.
- User-friendly interfaces, such as an LCD display or IoT API, to easily access and interpret the data.
- Empowered decision-making through visual representations and actionable insights derived from the data.

1.3. Initial SWOT Analysis

As a preliminary caution, having specified the requirements and project expectations I decided to carry out a SWOT analysis on myself. I felt this would help me identify the areas where I may need to dedicate extra time, before formulating my Gantt chart. This was done to give me a competitive advantage in my approach to achieving requirements and managing weaknesses.

Table 1 - SWOT Analysis Table

Strengths	Weaknesses
<ul style="list-style-type: none"> • Proficiency in programming (Python, JavaScript, and Swift). • Decent understanding of electrical components and electrical theory. • Proficiency in CAD design (SolidWorks). • Familiarity with implementing machine learning algorithms. • Familiarity with system architectures. 	<ul style="list-style-type: none"> • Little experience in PCB and circuit design. • Still new to implementing machine learning models. • Little experience with Arduino programming language. • Still new to programming microcontrollers.
Opportunities	Threats
<ul style="list-style-type: none"> • Research existing similar systems. • Improving skills regarding PCB or circuit design. • Demonstrate programming proficiency. • Demonstrate CAD design skills. • Learn more about machine learning. • Learn more about programming microcontrollers. • Learn more about electronics. • Learn more about sensors and actuators. • Determine sensors, microcontrollers, and actuators suitable for meeting requirements. 	<ul style="list-style-type: none"> • Poor time management. • Unrealistic goals (too ambitious). • Unforeseen circumstances. • Lack of sufficient learning material. • Overcomplicating things (method of implementation etc). • Not documenting/keeping track of progress. • Not saving work.

2. Background Research

To begin this project, it became apparent that I will need to do extensive research on methods of implementation for similar existing systems. This was done with the intention of gaining clarity on my method of approach. Also, I hoped to identify areas in which existing systems are currently facing problems, as well as their potential solutions or how these problems are mitigated.

2.1. Existing Systems

IoT applications in agriculture can be broadly categorized into two main groups: precision agriculture for outdoor environments and greenhouse monitoring for indoor settings. In the context of this project, it aligns with the principles of precision agriculture. Precision agriculture refers to the adoption of agricultural techniques that aim to enhance farmers' and crop growers' livelihoods by automating and optimizing various factors relevant to agricultural practices. By utilizing IoT sensors, farmers can effectively measure crucial variables such as soil quality, weather conditions, and moisture levels. Subsequently, these parameters can be optimized to maximize crop yield and overall productivity [1].

IoT precision agriculture consist of four components:

- ***Weather monitoring.***
- ***Soil conditions monitoring.***
- ***Plant disease monitoring.***
- ***Irrigation monitoring***

Below is some information detailing how these four components are being revolutionized via IoT technology.

Enhancing Agricultural Growth through Weather Monitoring and Analysis

Agricultural growth is greatly influenced by various crucial weather factors such as the ones I intended to collect namely, temperature, humidity, wind, and air pressure. To monitor these parameters effectively, advanced sensor technology is employed, both in wired and wireless forms, allowing the collection of accurate data. These collected weather data points are then transmitted to cloud servers for further analysis.

Once in the cloud, the data is combined with climate conditions and subjected to various analytic tools. Through the utilization of these tools, actionable insights can be derived to guide future steps and enhance agricultural growth. By examining the gathered information in relation to the climate patterns, informed decisions can be made to optimize farming practices and increase productivity.

The Importance of Soil Content Monitoring

Monitoring the contents of soil has emerged as a highly essential practice within the realm of agriculture and has been a staple point of agriculture since its advent in human history. Key factors that significantly impact agricultural cultivation involve the analysis of soil patterns, such as soil humidity, pH level, moisture content, and temperature, which IoT technologies have made easier. Soil content monitoring provides valuable insights and benefits, including:

- **Optimal Nutrient Management:** Soil content monitoring allows precise application of fertilizers, ensuring crops receive the right amount of nutrients for healthy growth and maximum yield.
- **pH Regulation:** Monitoring soil pH levels helps farmers adjust soil acidity or alkalinity, creating an optimal pH range for specific crops and promoting efficient nutrient absorption.

Disease Monitoring Revolutionized

The advent of IoT technology has ushered in a new era in agriculture, transforming crucial aspects like disease monitoring and identification. Farmers and gardeners now benefit from digitized applications that enable them to make well-informed decisions with remarkable speed. This remarkable progress is achieved through the integration of image processing and machine learning techniques, empowering farmers to accurately assess the health of their crops [2].

Revolutionizing Irrigation with Real-Time IoT Monitoring

Traditional irrigation systems have undergone also remarkable transformation through the integration of Internet of Things (IoT) technology, enabling a more innovative approach that considers the present weather and soil conditions in real time. This advanced monitoring system ensures that irrigation is implemented exclusively based on these parameters [3-4].

My project objectives; of weather and soil data collection fall within the bounds of both weather monitoring, and soil condition monitoring. From observing these components (weather monitoring and soil contents monitoring) I was sure that similarly I would require a means of monitoring/sensing the described parameters e.g., air pressure, humidity, soil moisture, etc. However, my exact choices of sensors along with my reasoning is discussed later in the report, see Component research.

2.2. Issues and Challenges

Smart agriculture consists of a combination of technologies. Through research I was able to quickly learn that amongst the various technologies that contribute to achievement of its objectives, comes a myriad of issues it faces spanning across each of them. The usual issues faced involve:

- **Physical components and equipment – (WSNs & Components)**
- **Communication and connectivity – (Networking)**
- **Underlying systems and framework – (Infrastructure)**
- **Disturbance in transmission signals – (Signal and Transmission Handling)**
- **Protection of information and privacy – (Security & Data Protection)**

2.2.1. Wireless Communication Protocol Selection for Precision Agriculture Systems.

Within precision agriculture, wireless devices tend to be vital. This is due to the scope of its coverage area (monitoring field) greatly varying. Also, within industry solutions to reduce power consumption or extend battery life are necessary as the communication protocols utilized in precision agriculture directly affect the systems power consumption. These solutions usually come in the form of energy harvesting such as solar cells, low power consumption sensors and communication technologies, or via the incorporation of intelligent power efficient management algorithms [1].

I was able to identify the five key stages which make up successful precision agriculture systems. Which are, data collection, diagnosis, data analysis, precision field operation, and evaluation. To do so, wireless sensor networks (WSNs) are implemented as part of the communication protocol. Despite the requirements of my systems falling short of what may be expected from a commercial product, I found to get the best out of my system the best wireless communication protocol should be selected [1]. Therefore, I carried out a study of each wireless communication protocol, to identify those suitable [see appendix 1]. My findings for suitable communication protocols were as follows:

Table 2 – Wireless Communication protocol study

Name	About	Suitable (Y/N)
ZigBee Wireless Protocol	<ul style="list-style-type: none"> • Considered the best candidate for farming domains. • Often is used for irrigation supervision, water quality management and pesticide control. • Low power consumption, low cost, and self-forming characteristics. • Reduceable communication range for indoor (greenhouse) monitoring. 	Y
Bluetooth (BT) Wireless Protocol	<ul style="list-style-type: none"> • The BT standard is used to establish a communication link between portable devices over a short distance. • Often used in collecting data on weather information, soil moisture, temperature, and sprinkler position. This is done using Global Positioning System (GPS) and BT Technologies. • Low energy consumption, wide availability, and ease of use. 	Y
Wi-Fi Wireless	<ul style="list-style-type: none"> • Currently the most utilized wireless technology available in portable devices (ubiquitous). 	Y

Protocol	<ul style="list-style-type: none"> • Suitable communication distance for indoor and outdoor environments. • Offers the ability to connect several types of devices via an ad hoc network. • Requires much power, long communication time, and huge data payload. • Wi-Fi nodes listen all the time, so power consumption will increase exponentially unless a form of control is implemented (via switches etc). 	
GPRS/3G/4G Technology	<ul style="list-style-type: none"> • General Packet Radio Service (GPRS) is a packet data service for GSM-based cellular phones. • GPRS depends on the volume of consumers that share the same communication channels and resources. 	Y
Long Range Radio (LoRa) Protocol	<ul style="list-style-type: none"> • A protocol for low power and wide area Internet of Things (IoT) communication technologies that are associated with indoor transmission. 	Y
SigFox Protocol	<ul style="list-style-type: none"> • SigFox is an ultra-narrowband wireless cellular network with low data rate applications. • Its technology is appropriate for IoT and machine-type communications systems. • Has been used in different applications, including telephone, security, mobile, broadband, and television. 	Y

LoRa and ZigBee are currently the most used protocols amongst industry agriculture systems. However, my use-case favoured both GPRS or Wi-Fi as my intended system is more localised and is only for garden usage. In the end I would opt for Wi-Fi as my choice of communication protocol due to the ease of compatibility with the existing use-case. For example, as the system is operating in a place where our user (John) will have Wi-Fi (John's House) I could easily connect the system to his existing Wi-Fi router, which would in turn assist with the transmission of data.

2.2.2. Hardware Components

For existing systems, it is quite possible that the hardware and other equipment could be exposed to fluctuating environmental conditions, as parts of the system are potentially situated outside. These conditions could adversely affect the equipment, causing damage. The design of the IoT devices is therefore very important. To support this, within industry, compliance with IP67 standard is mandatory especially for devices expected to operate outdoors [7][8].

The IP67 rating is a standardized measure used to assess the level of protection provided by a product against the intrusion of solid and liquid substances, particularly dust and water. The first digit, "6," indicates that the device is safeguarded against the entry of objects larger than 1mm in diameter, such as wires or small tools. On the other hand, the second digit, "7," signifies that the device can withstand temporary submersion in water under pressure between 15cm and 1m. When combined as "IP67," the device is expected to meet both protective standards [9]. This rating has motivated me to consider

incorporating some form of encapsulation or casing for each outdoor device in my system, ensuring that it remains impermeable to water to a significant extent.

2.2.3. Networking and Communications

Precision agriculture systems extensively utilize wireless communication due to the exorbitant costs associated with wiring and the challenges posed by wiring maintenance, as previously stated. However, communication signals between IoT devices and transceivers tend to weaken over time as they struggle to pass through the many physical obstacles in the agricultural field. In my case the usual physical obstacles were limited as the communication signals are only expected to travel between John (Inside his house) and his garden. Nonetheless, the most reliable network technology should be used for my system [10]. Parameters such as bandwidth speed would be taken into consideration as internet connectivity may not always be always reliable.

2.2.4. Security and Data Protection

Generally, most devices used in IoT applications are not designed with security and privacy in mind. This leads to security and privacy breaches where data integrity and access take a hit. In my system only, non-personal data was to be transmitted however, this still required some form of security protocol to ensure data is not manipulated adversely [11]. Separately, I found just like in existing systems, a mechanism that guarantees data security throughout IoT network could be implemented if need be. However, this was dependent on the wireless communication protocol I would decide to utilize [12].

3. Product Design Specification

After gaining some understanding on how similar IoT precision agriculture systems work and some of the problems/ issues they face, as well as the solutions utilized to get around them. The following questions arose:

- ***What wireless communication protocol was I going to use and why?***
- ***What microcontroller would I use and why?***
- ***What sensors would I use and why?***
- ***How would my sensors communicate?***
- ***What other components would I use and why?***

By answering these questions (some of which I had answered – namely the first one) I would be able to clarify my method of approach to fulfilling my requirements. However, beforehand I felt a blueprint describing the requirements for the project was needed. This was achieved via a simple product design specification document and would describe the product with precise and detailed information in several areas, forming as a checklist for deliverables.

Below is a depiction of the product design specification document in a table.

Table 3 – Product Design Specification Table

Project Overview		
<p>Name: IoT System for Environmental Data Collection (Precision Agriculture)</p> <p>Objective: Investigate the use of sensors and data science techniques to abstract environmental data and make predictions relevant to precision agriculture.</p> <p>User: John (Home Gardener)</p>		
Demand/Wishes (D/W)	Functional Requirements	Importance 1-10 (10 being most important)
Sensor Integration		

D	Integration - Integrate various electronic sensors to measure environmental parameters such as temperature, humidity, soil moisture etc.	9
D	Compatibility - Ensure compatibility of sensors with the chosen hardware platform.	8
Data Abstraction		
D	Algorithms - Develop software algorithms to process raw sensor data and abstract meaningful information related to the agricultural environment.	9
D	Legible Insights - Transform sensor data into legible and actionable insights for the user.	8
Predictive Analysis		
D	Data Science Techniques - Apply data science techniques, including machine learning algorithms, to analyse historical and real-time sensor data.	9
W	Predictive Models - Develop models to make predictions related to weather.	9
Networking		
W	Wireless Communication - Enable wireless communication between sensors and the central system.	7
D	Network Infrastructure - Establish a reliable network infrastructure to transmit sensor data.	8
User Interface		
D	GUI Design - Design a graphical user interface (GUI) to present the	9

	processed data and predictions in a clear and intuitive manner.	
D	Visualizations - Provide visualizations, and charts to help users interpret and act upon the information.	8
Data Storage		
D	Database System - Implement a database system to store and manage the collected sensor data for historical analysis and reference.	9
W	Security and Privacy - Ensure data security and privacy by adhering to best practices and relevant regulations.	6
Non-Functional Requirements		
Scalability		
W	Design the system to accommodate a scalable number of sensors and handle a growing volume of data.	7
Reliability		
D	Ensure high system reliability to prevent data loss and maintain continuous operation.	9
Performance		
D	Optimize data processing and analysis algorithms for efficient and timely execution.	8
Compatibility		
W	Ensure compatibility with different sensor models and hardware platforms commonly used in precision agriculture.	7

Usability		
W	Implement robust security measures to protect the system from unauthorized access or data breaches.	6
Portability		
W	Consider portability and compatibility with different operating systems and devices for user convenience.	6

4. Component Selection and Technology Research

4.1. Initial Hardware Setup

With the deliverables obtained from the product specification document, I brainstormed a design for an initial hardware setup for my system. The accompanying image illustrated my original plan, accompanied by an explanation of its functionality in meeting the specified requirements. It is important to acknowledge that this design underwent numerous revisions throughout the project, which will be elaborated on in subsequent sections of this report, along with the rationale behind these modifications.

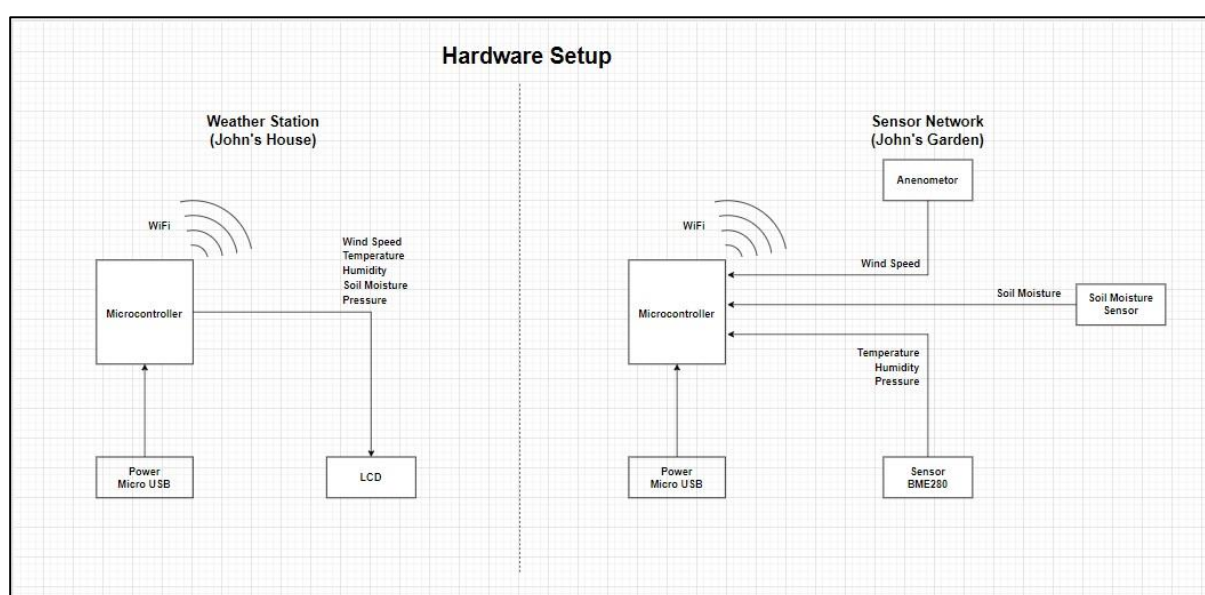


Figure 1 – Image depicting initial hardware setup design.

The image above illustrates a system that operates through communication between two microcontrollers, utilizing Wi-Fi as the means of communication. The sensors within this system detect environmental data and transmit it to the microcontroller integrated into the sensor network within John's Garden. Subsequently, this microcontroller forwards the data to the other microcontroller located in the weather station at John's House. The received data is then presented on an LCD screen. Throughout the communication process between the two microcontrollers, a logical mechanism for weather prediction is expected to be implemented. It was intended for this prediction logic to be executed once the sensor data reaches the microcontroller within the house. This framework provided a fundamental understanding of the necessary components to realize my concept, serving as a reference for the overall design.

4.2. Component Research and Selection

To select my components, it was necessary to conduct a thorough investigation into the available components and implementation methods. This involved analysing their advantages and disadvantages, enabling selection of the most suitable approach, which in turn would ultimately address the relevant initial questions posed.

4.2.1. *What microcontroller would I use and why?*

When selecting a microcontroller, I started with a list of existing microcontrollers and would rank them in favourability based on their cost, capabilities, software (programming language) compatibility and how they may affect my hardware setup (mainly the sensor network). This list consisted of the following:

- **ESP8266**
- **ESP32**
- **Arduino Uno R3 + (ESP8266 or ESP32)**
- **Arduino Uno Wi-Fi Rev2**

ESP8266

The ESP8266 module, developed by Espressif Systems, is a robust 32-bit microcontroller that packs a punch. It boasts an array of features, including built-in TCP/IP networking software and seamless integration with Wi-Fi networks. Operating on the 2.4GHz band, it provides reliable connectivity. Notably, the ESP8266 module gained popularity as an affordable solution for do-it-yourself IoT and home automation endeavours [16].

ESP32

The ESP32 module, produced also by Espressif Systems, is a robust 32-bit microcontroller that sets itself apart from the Arduino Uno. It boasts integrated Wi-Fi, offering seamless internet connectivity through a comprehensive TCP/IP protocol stack, as well as Bluetooth 4.2 capabilities. This module's architecture centres around a 2.4GHz Wi-Fi-and-Bluetooth combo chip, meticulously engineered to deliver exceptional power efficiency and optimal RF performance [16].

Arduino Uno R3 + (ESP8266 or ESP32)

This combination consists of a basic Arduino USB board and either the ESP266 or ESP32 module to enable Wi-Fi connection. The Uno's variety of shields allows its capabilities to be extended beyond just adding Wi-Fi via the Wi-Fi modules [13, 14]. It's design also makes connecting inputs and outputs

easier, which was something to bear in mind considering my SWOT analysis. There also was a lot of learning material regarding this microcontroller available online.

Arduino Uno Wi-Fi Rev2

The Arduino Uno Wi-Fi Rev2 is an Arduino Uno with an integrated Wi-Fi module. Unlike the Arduino Uno + (ESP8266 or ESP32) pairing described above the board is integrated with a self-contained module with TCP/IP protocol stack that can give access to a Wi-Fi network (or where the device can act as an access point) [15]. This would potentially allow me to restructure my hardware setup topology into more of a star-like structure. It can be powered via a USB connection or via an external power supply (Battery or AC-to-DC adapter).

Microcontroller Comparison Report

After familiarizing myself with the various options for microcontrollers, I decided that the most effective approach to selection would be to create a comparative report in the form of a table. In this report, I would evaluate each available option by considering relevant parameters and comparing them against one another.

Table 4 - Table detailing how well each potential microcontroller performs regarding the necessary parameters [17].

Parameters	ESP8266 NodeMCU V2	NodeMCU V3	ESP32 NodeMCU	ESP8266 WeMos D1 Mini	Arduino UNO R3	Arduino UNO Wi-Fi R2
Microcontroller	ESP8266	ESP8266	ESP32	ESP8266	ATmega328p	ATmega4809
Operating Voltage	3.3V	3.3V	3.3V	3.3V	5V	5V
Power Supply	7V-12V	7V-12V	7V-12V	4V-6V	7V-12V	7V-12V
Current consumption	15 μ A – 400 mA	15 μ A – 400 mA	20 mA – 240 mA	N/A	45 mA – 80 mA	50 mA – 150 mA
Current consumption (Deep Sleep)	0.5 μ A	0.5 μ A	5 μ A	N/A	35 mA	35 mA
Digital I/O Pins	11 or 13	16	36	11	14	14

Digital I/O Pins with PWM	11 or 13	16	36	11	6	5
Flash Memory	4 MB	4 MB	4 MB	4 MB	32 KB	48 KB
SRAM	-	64 KB	520 KB	-	2 KB	6 KB
EEPROM	512 bytes	512 bytes	-	-	1024 bytes	256 bytes
Analog Input Pins	1	1	15	1	6	6
Clock Speed	52 MHz	80 MHz	80 MHz / 160 MHz	80 MHz / 160 MHz	16 MHz	16 MHz
Wi-Fi	✓	✓	✓	✓	✗	✓
Bluetooth	✗	✗	✓	✗	✗	✗
USB connection	✓	✓	✓	✓	✓	✓
Power jack	✗	✗	✗	✗	✓	✓
CAN	✗	✗	✓	✗	✗	✗
Price	£5.99	£6.99	£8.99	£4.99	£21.28	£41.41

After careful consideration, I ultimately chose to utilize the ESP32 microcontroller (specifically the ESP32 NodeMCU) due to its affordability, Wi-Fi capabilities, and Bluetooth functionalities, all of which align with the communication protocol I had previously selected. The ESP32's provision of 15 analog input pins is essential for both the LCD component of my system and for obtaining readings from the anemometer. Furthermore, the notable difference in power consumption between its normal state and deep sleep mode allows for effective power management when necessary.

Additionally, I discovered that ESP microcontrollers could communicate through a wireless protocol called ESPNow, which offers a communication equal to that of Wi-Fi (400m). This protocol, developed by Espressif, facilitates direct, speedy, and low-power control of smart devices without relying on a

router [18]. I believed this feature could prove invaluable in case of internet connection failures, as it ensures uninterrupted data transmission.

4.2.2. *What sensors would I use and why?*

To answer this question, I had to revisit the fundamental objectives of this project. As part of the fundamental objectives one of my main goals is to monitor the environment with a key focus on the following environmental parameters:

- ***Humidity***
- ***Temperature***
- ***Soil Moisture***
- ***Wind speed***

To do so I chose to make use of sensors that detect changes in the forementioned environmental parameters. I also included pressure amongst the list of important environmental parameters to focus on due to its relevance when predicting future forecasts. The sensors I ended up choosing were:

- ***BME280 sensor***
- ***Adafruit Anemometer***
- ***Capacitive Soil Moisture sensor***

Below are descriptions of my chosen sensors, along with the rationale behind selecting them.

BME280

The BME280 is a low-power, high-functional sensor module. Most available environmental sensors tend to collect data regarding one or two parameters, for example, the DHT11 (temperature and humidity) and the LM35 (temperature). However, the BME280 sensor module was responsible for reading three parameters: temperature, barometric pressure, and humidity. This provided users with a comprehensive and holistic measurement of the environment. When it came to the temperature range, the BME280 could measure a range between -40 to 85°C, which was considerably higher than its competitors, with the BME180 only clocking a range between 0 to 65°C. In addition, the BME280 could use either of its two communication protocols (I2C or SPI), meanwhile, competitors such as the BMP180 could only use one. This can be seen in the table below which depicts how the BME280 fared in comparison with its competitors.

Table 5 - A detailed comparison between each of the BME280's competitors can be seen below.

Sensor	DHT11	DHT22	LM35	DS18B20	BME280	BMP180
Measures	Temperature Humidity	Temperature Humidity	Temperature	Temperature	Temperature Humidity Pressure	Temperature Pressure
Communication Protocol	One-wire	One-wire	Analog	One-wire	12C SPI	12C
Supply Voltage	3 to 5.5V DC	3 to 6V DC	4 to 30V DC	3 to 5.5V DC	1.7 to 3.6V (for the chip) 3.3 to 5V for the board	1.8 to 3.6V (for the chip) 3.3 to 5V for the board
Temperature Range	0 to 50° C	-40 to 80° C	-55 to 150° C	-55 to 125° C	-40 to 85° C	0 to 65° C
Accuracy	+/- 2° C (at 0 to 50° C)	+/- 0.5° C (at -40 to 80° C)	+/- 0.5° C (at 25° C)	+/- 0.5° C (at -10 to 85° C)	+/- 0.5° C (at 25° C)	+/- 0.5° C (at 25° C)
Support (Arduino IDE)	Adafruit DHT Library Adafruit Unified Sensor Library	Adafruit DHT Library Adafruit Unified Sensor Library	analogRead()	DallasTemperature OneWire	Adafruit BME280 Library Adafruit Unified Sensor Library	Adafruit BME085 Adafruit Unified Sensor Library
Support (MicroPython)	dht module (included in MicroPython firmware)	dht module (included in MicroPython firmware)	from machine import ADC ADC().read	ds18b20 module (included in MicroPython firmware)	BME280 Adafruit Library	BMP180 module

All in all, the BME280 sensor was more expensive but offered more functionality. For example, users could build a weather station project with only this sensor, which was ideal and fell in line with the objective for this project. Another reason for opting to use this sensor was that, according to research experiments, the BME280 gave more stable temperature readings with smaller oscillations occurring between readings.

Adafruit Anemometer

An anemometer serves the purpose of measuring wind speed and is commonly utilized as a weather station instrument. The anemometer I selected is specifically designed to be placed outdoors, enabling effortless measurement of wind speed. However, it is important to note that it lacks an IP rating, which meant it is not fully waterproof and could benefit from additional protective measures. I selected this component due to its affordability and its suitability for my specific use-case.

Table 6 - Table depicting key the specifications for the Adafruit anemometer [19].

Input Voltage	Output Voltage	Testing Range	Start Wind Speed	Resolution	Max Error	Max Wind Speed
7-24VDC	0.4-2.0V	0.5-5.0 m/s	0.2 m/s	0.1 m/s	1 m/s	70 m/s

Due to the high input voltage required for the anemometer I noticed a voltage booster would be required to boost the normal micro-USB voltage of 5V to the expected minimum of 7.5V.

Capacitive Soil Moisture sensor

In my search for a soil moisture sensor, I ultimately chose a capacitive soil moisture sensor. This type of sensor detects changes in capacitance to measure the water content of the soil. The reason behind my decision was the sensor's construction using corrosion-resistant materials, which made it ideal for handling unpredictable environmental conditions and ensured a longer lifespan compared to its competitors, specifically the resistive soil moisture sensor. Moreover, capacitive soil moisture sensors provided more precise readings of soil moisture content compared to resistance measurements, making them a preferred choice [20].

4.2.3. What signal processing components would I use and why?

To address this question, I needed to revisit my system's signal processing requirements. In this case, I aimed to present the readings obtained from each sensor in textual format on a display screen, catering to the user's viewing needs. To achieve this, I opted for the following method:

- ***Nextion 3.5-inch HMI Display***

Additionally, I thought to include in implementation for presenting readings to the user the ***ThingSpeaks IoT Cloud Platform*** for analytics. This would help with fulfilling the IoT aspect of the project.

Nextion 3.5-inch HMI Display

The NEXTION HMI (human machine interface) has gained significant popularity within the Arduino community due to its affordability, seamless Serial connectivity, and the inclusion of a user-friendly GUI editor. With the help of the Nextion Editor software, developers can swiftly create interactive graphical user interfaces (GUIs) by simply dragging and dropping components such as graphics, text, buttons, sliders, and more. To define the behaviour of these components on the display, ASCII text-based instructions can be utilized for coding purposes. Additionally, the Nextion HMI supports touchscreen functionality, providing users with even greater capabilities and versatility. Despite having

no prior experience with NEXTION HMIs before its ease of usability was the deciding factor in opting for this method of signal processing.

ThingSpeak

ThingSpeak was the ideal choice as an approach to IoT analytics. As a cloud based IoT analytics platform, ThingSpeak offers a wide range of services that align perfectly with the project requirements. What stood out was the fact it offered effortless aggregation, visualisation of data, while analysing real-time data streams, making it ideal for monitoring the environmental conditions crucial to precision agriculture practices. With seamless data transmission from my devices, I could generate instant visualizations, receive timely alerts, and access the platform from anywhere via mobile devices. Additionally, ThingSpeak's forecasting capabilities enable you to predict future readings based on historical data, empowering you to make well-informed decisions. However, I did not intend to use it for its forecasting within this project but saw this a potential avenue for future developments.

4.3. Programming Approach and Methodology

After finalizing my component selection as seen in the bill of materials [see appendix 3], and IoT technologies, I turned my attention to the programming aspect of implementation. Having opted for the ESP32 microcontroller as my preferred choice of microcontroller, which could be programmed using both Arduino code and MicroPython. Initially, I embarked on programming in Arduino to test ideas out, intending to handle the machine learning component of my project by establishing communication with the Python environment through PySerial, which is a library that facilitates serial connections ("RS-232") across various devices. However, this approach necessitated a constant connection between my system and the computer hosting the Python code.

Upon further exploration, I discovered an alternative approach: creating a Python client connection to receive daily weather parameter data from the ESP32. Subsequently, this data could be processed by Python code to update a CSV file and handle the machine learning aspect for prediction purposes. After conducting thorough research, I opted to shift my programming approach from Arduino code using the Arduino IDE to programming the microcontrollers in MicroPython utilizing the Thonny IDE. This decision was based on my proficiency in Python programming, as highlighted in the SWOT analysis, as opposed to my limited experience with Arduino programming and the Arduino IDE. Although MicroPython closely resembles standard Python, I soon realised it does have certain limitations [24]. In changing my choice of programming language, I would also have to change my approach in displaying data on the Nextion HMI and would have to investigate MicroPython Bit-Serial ports to achieve this.

5. Design Documentation

5.1. Hardware Setup (2nd Iteration)

Having identified and selected the components, the software programming approach and the IoT technology I would use for my system, I began referring to my 3 main requirements (mentioned in section 1.2.) along with the product design specification document. This accompanied with the extensive research I had carried out before my selections were made, gave me a broader idea of component capabilities, and the possibilities for this project. At this point I felt modifications to my hardware setup would be necessary. I decided to alter my initial setup by using four microcontrollers instead of two, where communication between the weather station and the sensor station(s) was to be achieved via the ESPNow protocol, and sensor readings would be published to the ThingSpeak cloud platform via the sensor station(s). Additionally, I incorporated a Python client running on a laptop for handling the predictive analysis aspect of my project. Here, I aimed to establish a TCP server-client connection with the weather station, for transmitting daily aggregated data and adding it amongst the historical data (csv file) to then make a prediction. Once a prediction was made, I would then send the prediction back to the weather station to be displayed on the LCD (Nextion HMI). The network topology between each aspect of this system follows a hybrid form.

Below is an image depicting the renewed hardware setup after implementing these changes.

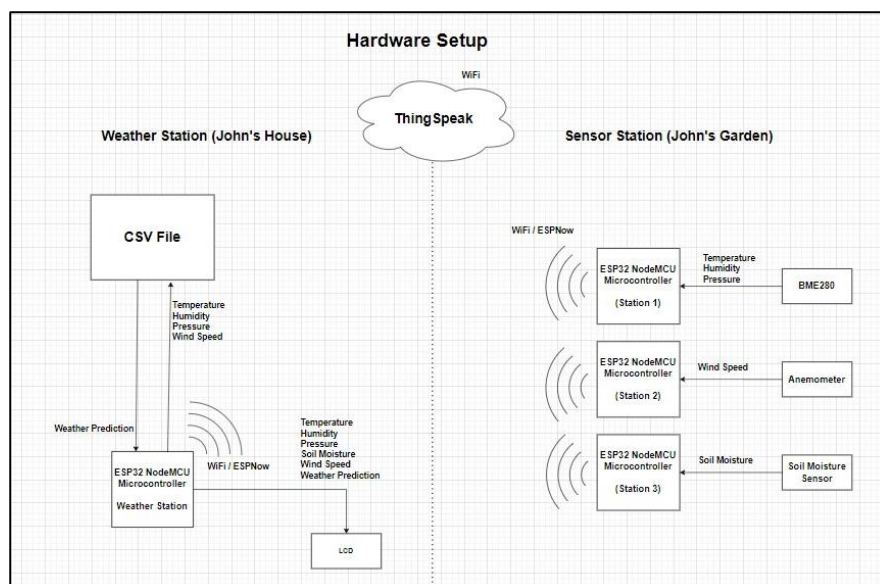


Figure 2 - Image depicting the systems hardware setup.

Despite these changes, my system did undergo a final iteration, which is explained further, along with my rationale behind such change in the "Implementation" chapter of this report.

5.2. Functionality Checklists (Pass Tests)

Once I established the new hardware configuration, a concept started to form regarding the functional specifications I wanted for my system. As a result, the following questions emerged:

- ***What are the functional requirements of my system?***
- ***How does my system work?***
- ***What is the relevant data to display?***
- ***How should the user interface be organized?***

Answering these questions would not be difficult, however answering them in a manner that was coherent, and maintaining a sense of direction throughout this project to ensure the requirements were met in line with my PDS could be difficult, as deviation could occur. To ensure this would not be the case I made use of specific design documentation.

Regarding assessing functionality, I developed a pass test checklist for each of my sensor stations, the weather station, and the Python client. These checklists ensured that each component of the system met specific criteria and requirements. Below is a depiction of the pass test checklists for each component:

Table 7 - Depicting Pass test checklist for sensor stations.

Sensor Station Pass Tests	
Connect to Wi-Fi network	✓
Retrieve readings from sensor(s)	✓
Retrieve readings from sensor(s) for targeted timestamps (e.g., hourly, or 9am and 3pm)	✓
Send readings to Weather Station	✓
Publish sensor readings as (topic) to ThingSpeaks cloud platform	✓

Table 8 - Depicting Pass test checklist for Weather Station (Nextion HMI Circuit)

Weather Station (Nextion HMI Circuit)	
Connect to Wi-Fi	✓

Retrieve data sent from each Sensor Station	✓
Display relevant data on Nextion HMI	✓
Group timestamp data appropriately as (Daily data)	✓
Create MicroPython Server	✓
Establish connection with Python Client	✓
Send daily data to Python Client	✓
Retrieve forecast from Python Client	✓

Table 9 - Depicting Pass test checklist for Python Client

Python Client	
Establish connection with MicroPython server	✓
Retrieve data sent from MicroPython server	✓
Update dataset (CSV file) with newly acquired data	✓
Calculate predictions using machine learning model	✓
Send forecast back to MicroPython server	✓

Not only did these checklists assist in answering the first question posed “**What are the functional requirements of my system?**”, but by using these pass test checklists, I could verify the proper functioning and integration of components, including microcontrollers, the Python client, ThingSpeak IoT cloud platform, and Wi-Fi communication.

5.3. System Flowcharts

The next step involved developing a conceptual understanding of the system architecture, subsequently answering the second question posed “**How does my system work?**”. To address the "weakness" of inexperience in programming microcontrollers, I decided to develop flowcharts for the sensor stations, weather station, and python client. These flowcharts helped me visualize the system flow and clarified communication mediums. They assisted in highlighting data transmission timestamps and showcasing microcontroller integration, simplifying the programming process by

providing a clear roadmap for code implementation and reducing errors and inconsistencies. The flowcharts also served as documentation for future reference and facilitated easier maintenance and updates.

Below are depictions of my initial flow charts for the sensor station(s), weather station, and python client. A better iteration of which can be accessed on the project blog page [appendix 1].

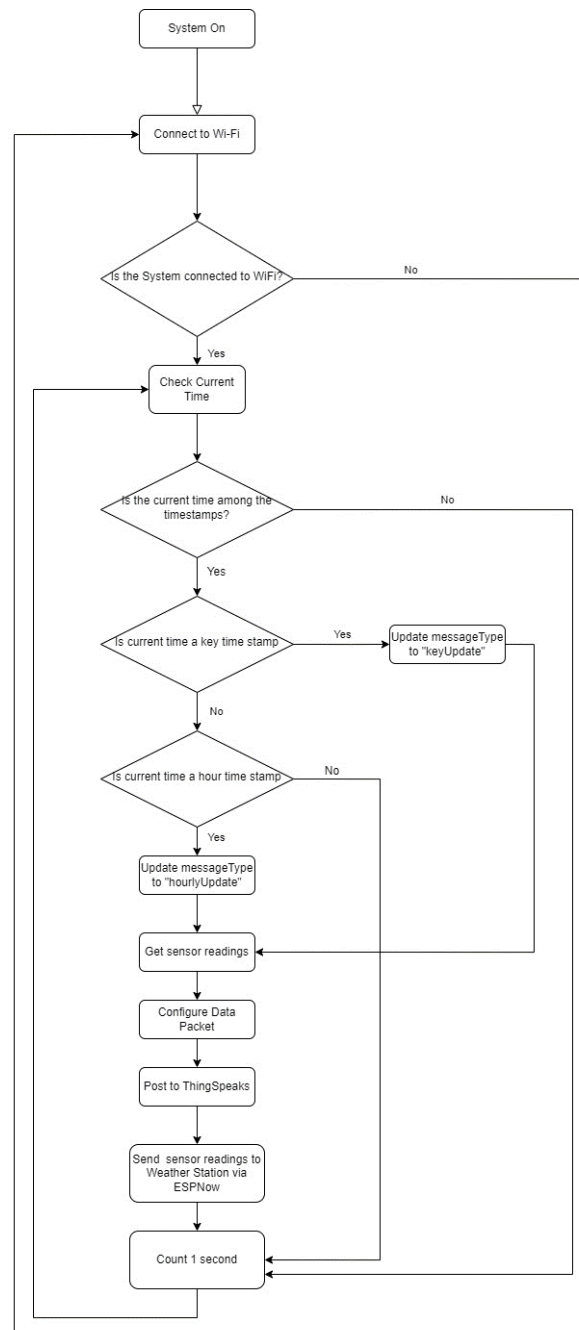


Figure 3 - Flowchart for Sensor Station(s).

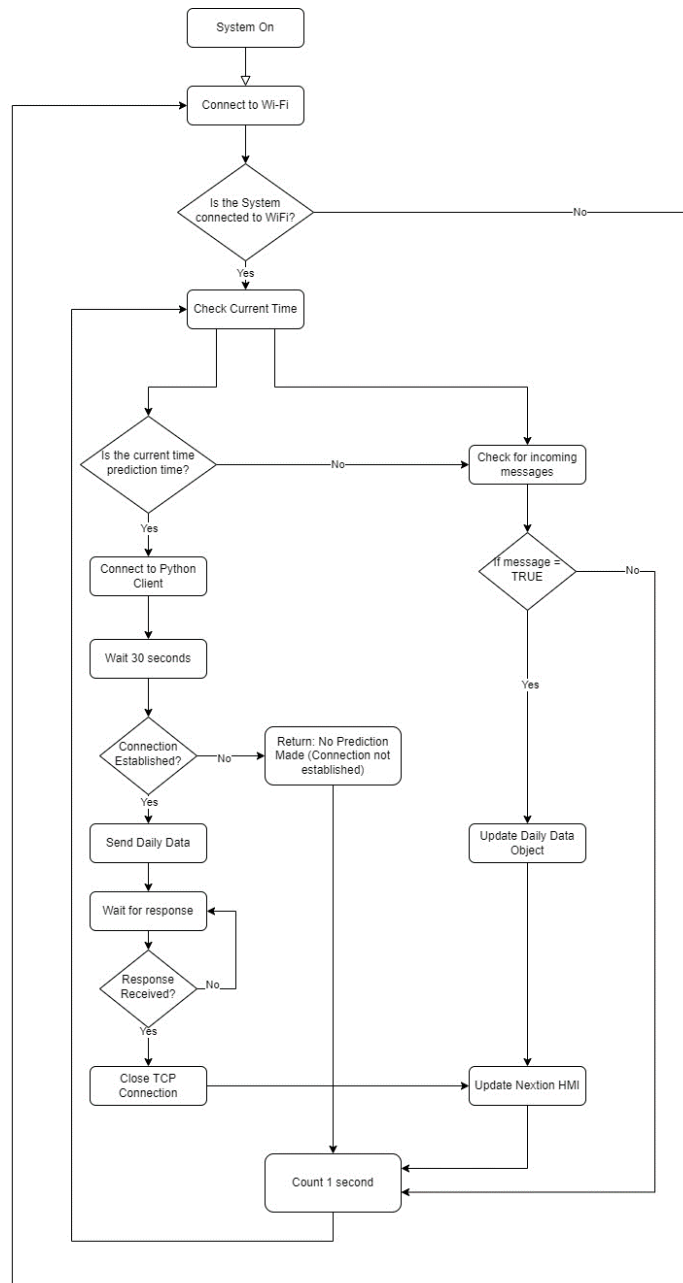


Figure 4 - Flowchart for Weather Station.

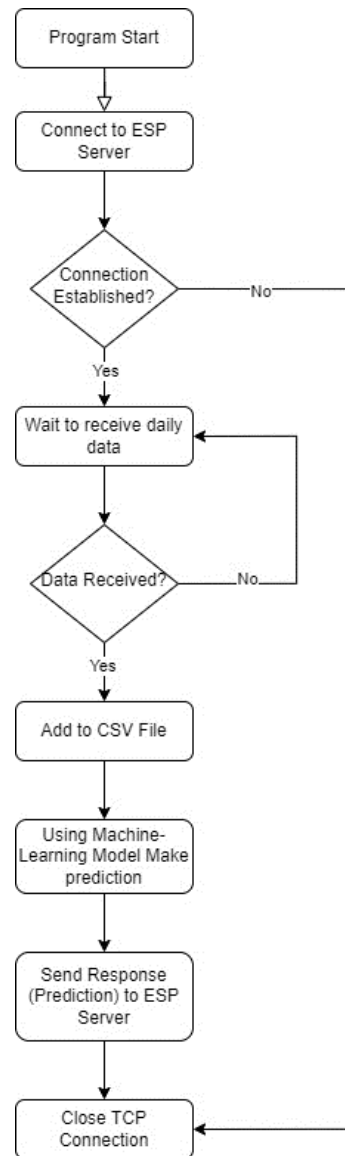


Figure 5 - Flowchart for Python Client

Throughout implementation, testing and validation I used failure injection tests, which would alter the overall processes depicted in my initial flowcharts. The changes implemented are described and validated in the forthcoming chapters.

5.4. GUI Design

The remaining two questions of ***“What is the relevant data to display?”***, and ***“How should the user interface be organized?”*** generally revolved around the design of the GUI. To answer the first of the two remaining, I referred to the research I had carried out, and the environmental parameters that were relevant to my use case, and more specifically weather forecasting. From this I determined the information of relevance to the users (Johns) problem was:

- **Temperature.**
- **Soil moisture.**
- **Pressure.**
- **Humidity.**
- **Wind speed.**
- **The forecast.**

With this information I was able to develop a basic design for the layout of my GUI, subsequently answering the last question “**How should the user interface be organized?**”. Below is an image of my initial sketch [Figure 6]. I tried to ensure that each of the forementioned relevant pieces of information had a visual label on display to the user, as the primary goal was to relay said information in a legible manner.

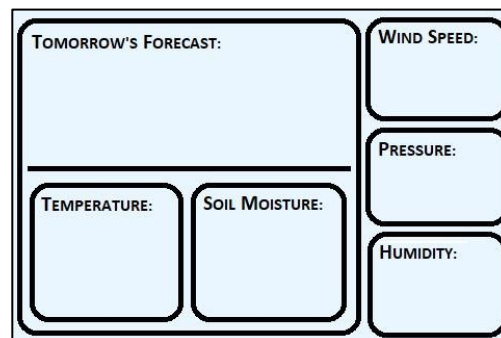


Figure 6 - GUI Sketch (made using Paint)

Using this I then developed the official GUI in the Nextion editor by adding the appropriate labels as depicted in [Figure 7].

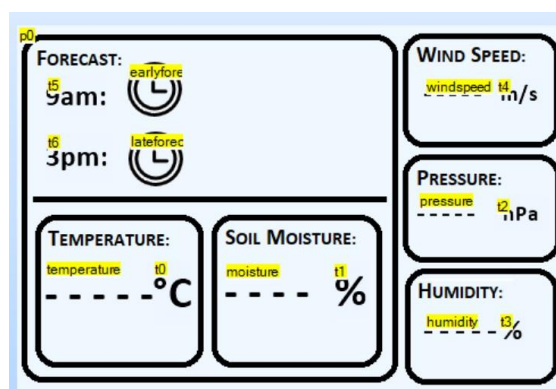








Figure 7 – Nextion HMI GUI

My intention was for each environmental parameter box to be populated with the sensor stations most recent reading value for the corresponding parameter(s), meanwhile the forecast box to be

populated with icons that represent the forecast for the day at 9am and 3pm, in line with the table below depicted:

Table 10 – Forecast symbols and their meanings.

ID number	Icons	Status
1		Awaiting Prediction
2		Settled
3		Less Settled
4		Storm, Heavy Rain
5		Unable to connect to Python Client
6		Rain expected

5.5. Casings CAD Design

With access to the component's datasheets, I took it upon myself to design casings for my physical circuits. Despite this not being one of the key requirements or part of my PDS I felt this could be incorporated as another part of testing. The models of which can be seen in the images below:

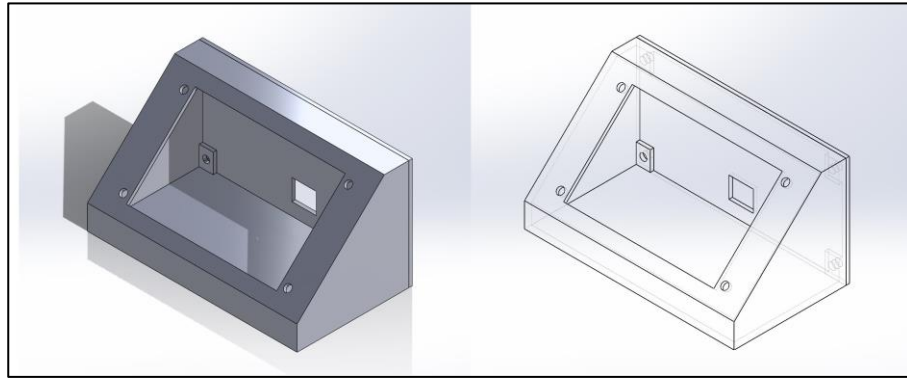


Figure 8 - Nextion HMI Casing Model (Isometric view)

For the Nextion HMI casing model I took the screen dimensions into consideration, along with the holes for mounting it to the casing. Additionally, by using a calliper I decided to add an extruded cut for the micro-USB cables to connect to the ESP32.

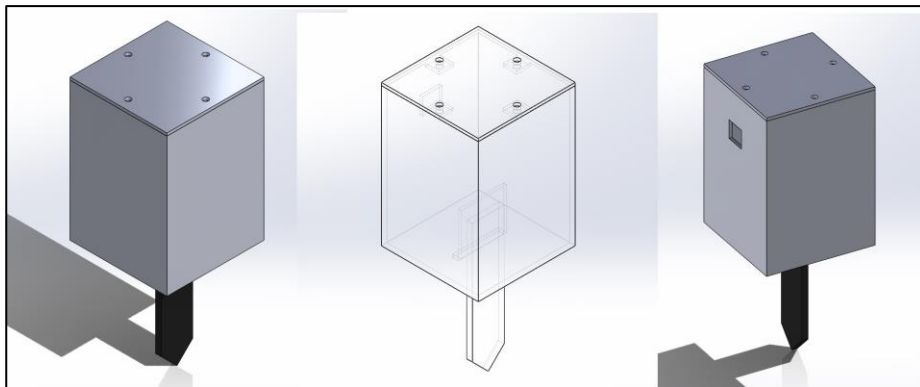


Figure 9 – Capacitive Soil Moisture Sensor Casing Model (Isometric view)

Similarly, I included an extruded cut for the micro-USB cable to be connected to the ESP32 for the Capacitive soil moisture sensor casing. I also added a slot with a wall to mount the sensor itself which I intended achieve by gluing the sensor against the wall.

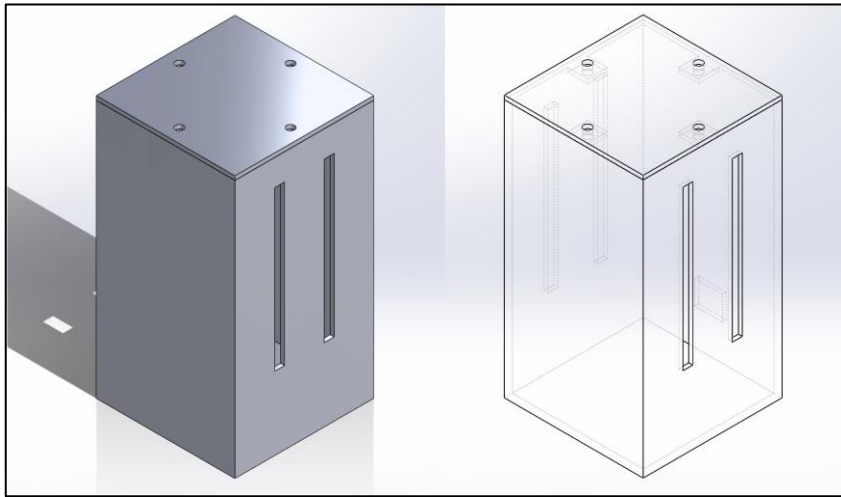


Figure 10 – BME280 Sensor Casing Model (Isometric view)

For the BME280 sensor casing I decided to add slots to each side of the casing to act as ventilation so accurate sensor readings for temperature, pressure and humidity can be taken.

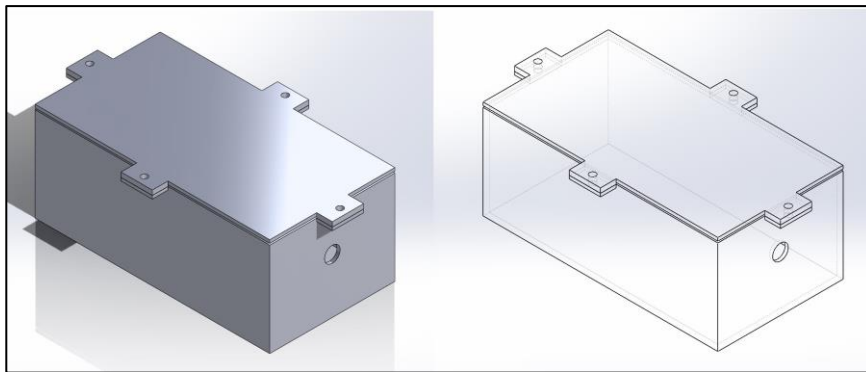


Figure 11 - Adafruit Anemometer Casing Model (Isometric view)

For the Adafruit anemometer casing model I included a hole to exceed the cables diameter, done by measuring it using a calliper. Additionally, I made sure the box could encapsulate both the battery and the mini-breadboard I intended to use. For details surrounding the datasheets, and to see the exact models visit [appendix 1].

5.6. Circuit Design: Schematics

Due to my limited experience in electronics and circuit design, I recognized the necessity of creating individual schematics for each microcontroller station. These schematics aided in understanding and visualizing the system's hardware configuration, facilitating effective communication and documentation. They provided a clear representation of signal paths, component placements, and potential issues, which helped in troubleshooting and ensuring smooth operation of the system.

To accomplish this, I utilized the software program Fritzing, which is designed for creating and sharing electronic schematics, circuit designs, and PCB layouts. It has an intuitive interface and an extensive component library. By using Fritzing, I designed detailed schematics for each microcontroller station, considering the pin-out layout [see appendix 4] for my chosen microcontroller and the specifications of each component.

1.1.1. Sensor Station 1 - BME280 Circuit Schematic

The BME280 circuit schematic (Figure 12 & 13) represents the hardware configuration for Sensor Station 1. This schematic was designed to accommodate the I2C communication protocol used by the BME280 sensor for bidirectional data transfer. The connections here were carefully made with specific considerations in mind:

The SDA (Serial Data) and SCL (Serial Clock) lines were connected to the corresponding I2C pins on the microcontroller. These lines enabled the transmission and reception of data in the I2C communication.

Power supply connections were established by linking the VCC and GND pins of both the sensor and the microcontroller. These connections were responsible for delivering the necessary electrical power for the components to operate effectively.

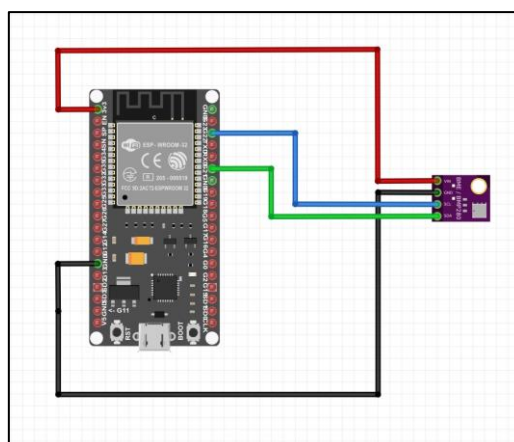


Figure 12 – Fritzing BME280 Circuit

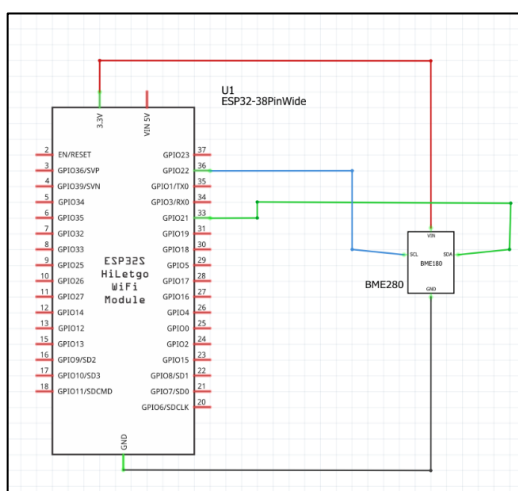


Figure 13 - Fritzing BME280 Circuit Schematic

1.1.2. Sensor Station 2 – Adafruit Anemometer Circuit Schematic

The Adafruit Anemometer circuit schematic (Figure 14 & 15) represents the hardware configuration for Sensor Station 2. As stated, due to the high input voltage required for the anemometer a voltage booster was necessary for voltage regulation.

In my schematic design the anemometer is connected to the ESP32 microcontroller using a DC-to-DC boost converter. The power supply's positive terminal was connected to the boost converter's VIN pin, while the negative terminal was connected to the boost converter's GND pin. This design would ensure a stable power source for the anemometer and allowed for voltage boosting when necessary. In this case the output voltage of the boost converter would be adjusted to match the voltage requirements of the Adafruit anemometer (7.5V) [see table 6]. The boosted voltage would then be connected to the anemometer's V+ or VIN pin for power, while the negative terminal is connected to the anemometer's GND pin. This would provide the anemometer with the required voltage for proper operation.

The signal wire provided by the Adafruit anemometer is then connected to GPIO 35 on the ESP32 microcontroller. My reasons for selecting GPIO 35 was based on pin availability and project requirements. This pin would support, accurate readings of the wind speed via PWM from the anemometer thanks to its digital input/output capabilities and support for interrupt functionality.

To power the ESP32 microcontroller, the appropriate power supply, such as a battery or regulated power source, could be connected to the designated power input pins on the ESP32. This step would provide power to the microcontroller, enabling it to interface with the anemometer and process the received signals.

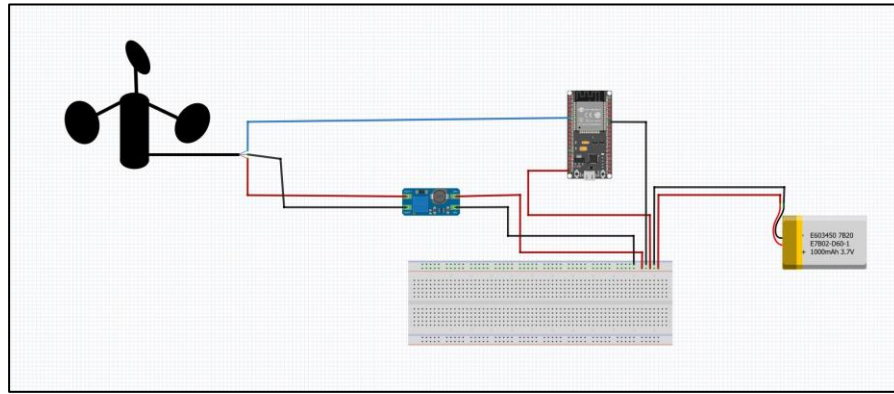


Figure 14 – Fritzing Adafruit Anemometer Circuit

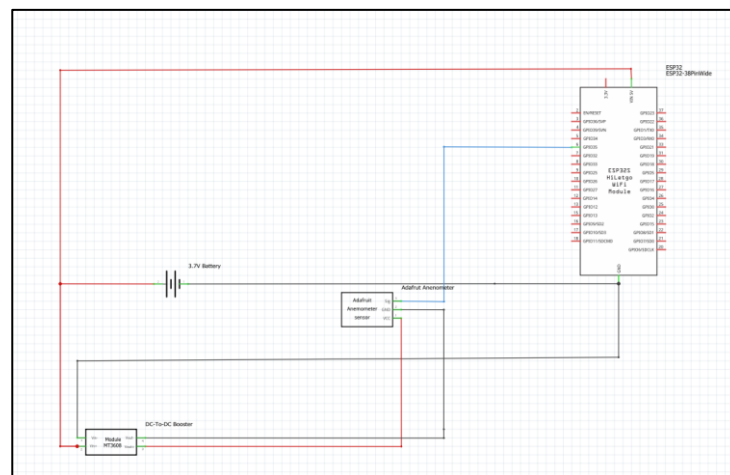


Figure 15 - Fritzing Adafruit Anemometer Circuit Schematic

1.1.3. Sensor Station 3 – Capacitive Soil Moisture Circuit Schematic

The Capacitive Soil Moisture circuit schematic (Figure 16 & 17) represents the hardware configuration for Sensor Station 3. In this schematic, the soil moisture sensor is connected to the ESP32's analog input pin 34. This analog input pin allows for precise measurement of the analog signal provided by the capacitive soil moisture sensor. The analog input pin, along with the Analog-to-Digital Converter (ADC) on the ESP32, enables the conversion of the varying voltage output from the sensor into a corresponding digital value.

By reading the analog input voltage, the ESP32 could accurately determine the soil moisture level based on the signal's magnitude. Analog measurements provide a more accurate representation of soil moisture compared to a simple digital on/off signal, allowing for better control and monitoring of applications reliant on soil moisture data.

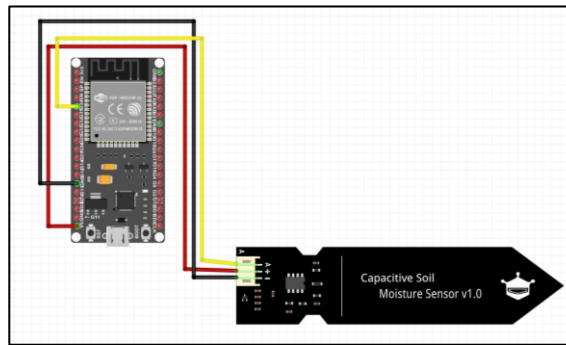


Figure 16 – Fritzing Capacitive Soil Moisture Sensor Circuit

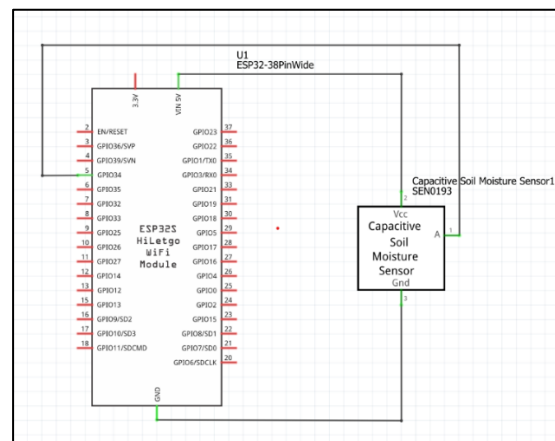


Figure 17 – Fritzing Capacitive Soil Moisture Sensor Circuit Schematic

1.1.4. Weather Station – Nextion HMI Circuit Schematic

For the design of the circuit connecting the Nextion Human Machine Interface (HMI) to the ESP32 microcontroller, careful consideration was given to the choice of GPIO pins to ensure efficient and reliable communication between the two devices. To enable seamless data exchange, UART (Universal Asynchronous Receiver/Transmitter) communication was selected as the preferred protocol. Consequently, GPIO17 on the ESP32 was chosen as the dedicated TX (transmit) pin for UART2 functionality, allowing for the smooth transmission of data from the ESP32 to the Nextion HMI. As my system did not require feedback from the user via the HMI, for the RX (receive) connection, it was deemed unnecessary to select an available GPIO pin on the ESP32 to interface with the Nextion HMI's TX pin. This arrangement ensured one-directional communication, as the ESP32 didn't need to receive data from the HMI as well. Moreover, careful attention was given to voltage compatibility, verifying that both devices operated at compatible voltage levels, which is typically 3.3V. Lastly, to maintain proper signal communication, a common ground connection was established between the ESP32 and the Nextion HMI. These thoughtful wiring choices and configuration considerations were essential in creating a robust and efficient circuit connecting the Nextion HMI to the ESP32 microcontroller,

enabling seamless user interaction with the display and facilitating data exchange for various applications.

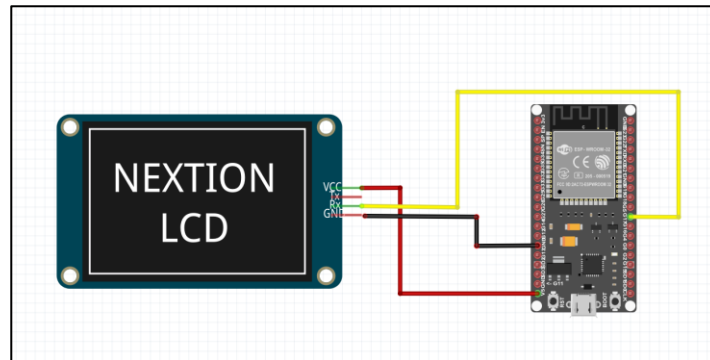


Figure 18 – Fritzing Nextion HMI Circuit

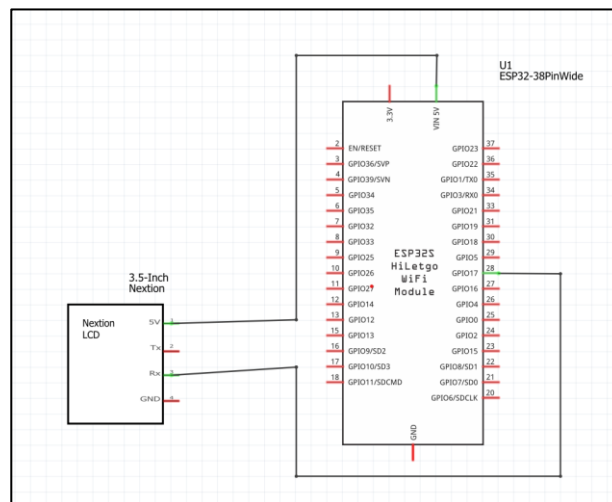


Figure 19 - Fritzing Nextion HMI Circuit Schematic

6. Implementation and Testing

As I delved into the implementation process of my project, integrating both the hardware and software components, I encountered various challenges and made iterative improvements to ensure a robust and integrated system. In this chapter, I will guide you through the steps I took to assemble the circuits, connect to the Wi-Fi network, retrieve sensor readings, and transmit the data to the weather station. Additionally, I will discuss the development of the graphical user interface (GUI) for data visualization. By sharing my experiences and insights, I aim to provide a comprehensive understanding of the implementation and testing aspects of my project.

6.1. Hardware Implementation and Testing

The initial phase of implementation involved assembling the circuits according to the schematics. I carefully followed the pinout layout to ensure proper connectivity and functionality. This step was crucial as it established a solid foundation for the subsequent software implementation, data processing, and user interface development. The assembled hardware components formed a reliable sensor network capable of providing accurate measurements of environmental parameters, including temperature, pressure, humidity, and windspeed.

6.2. Software Implementation and Testing

Once the hardware implementation was completed, I focused on the software implementation phase, where I developed programs and algorithms to facilitate data acquisition, processing, and analysis within the sensor network.

6.2.1. *Connecting to the Wi-Fi network*

Establishing a connection to the Wi-Fi network was crucial for seamless communication between the sensor network stations and the weather station. Using the "getConnection" method within the "ESP32Handler" class, I successfully connected my system to the user's Wi-Fi router. This enabled further testing of software requirements on the sensor network stations, ensuring reliable and efficient data transmission.


```
def getConnection(self, station):  
    ...  
    This method is responsible for establishing a connection to the WiFi router by taking the station as an argument and connecting to it via SSID and Password.  
    After which the senders MAC address is retrieved and officially set along with the host name.  
    ...  
    station.active(True)  
    station.connect("YOUR-SSID", "YOUR-PASSWORD")  
    self.senderMacAddress = station.config('mac')  
    self.hostName = station.ifconfig()[2]
```

Figure 20 – Snippet depicting getConnection function.

```
root = ESP32Handler()  
# Declare WLAN station as variable  
station = network.WLAN(network.STA_IF)  
# If statement checking if the ESP is connected to WiFi  
if station.isconnected() == True:  
    print(" Connected!")  
    # Update sender's MAC address variable  
    root.senderMacAddress = station.config('mac')  
    root.hostName = station.ifconfig()[2]  
else:  
    # Run function to establish a connection and get MAC address  
    root.getConnection(station)  
    # While statement to add "." to  
    while not station.isconnected():  
        print(".", end="")  
        sleep(0.1)  
    print(" Connection Successful!")
```

Figure 21 - Depicting query program used to establish connection (trigger getConnection function).

Troubleshooting Wi-Fi connections

To achieve successful connection to a Wi-Fi network, at first my Wi-Fi router credentials were hard programmed into my system. I soon realised that just like in real life scenarios, sometimes the Wi-Fi connection may fail, but at this point I had no means of notifying the user if this was the case, which could cause confusion to why the system may not be working properly if this was to occur. To combat prevent this potential issue, I decided to implement among my while loop which checks the current time every second (see section 6.2.3.), a query to check if the Wi-Fi connection was still active for both the weather station and sensor station(s). In the case that the Wi-Fi connection was lost at the sensor station(s) a Bluetooth connection to the weather station would be established where a message notifying the loss of connection would be sent.

```
def sendTroubleshootMessage(self):  
    ...  
    This method is responsible for sending a troubleshoot message via bluetooth to the weather station  
    communication protocol.  
    ...  
  
    server_sock = bluetooth.BluetoothSocket(bluetooth.RFCOMM)  
  
    # Enable Bluetooth discoverable mode  
    bluetooth.advertise_service(server_sock, self.bluetooth_name, service_id=bluetooth.UUID("00001101-0000-1000-8000-00805F9B34FB"), advertise=True)  
  
    # Bind the server socket to a port and start listening for incoming connections  
    server_sock.bind(("", bluetooth.PORT_ANY))  
    server_sock.listen(1)  
  
    # Get the server socket port  
    port = server_sock.getsockname()[1]  
  
    #print("Waiting for connection on RFCOMM channel", port)  
  
    # Accept an incoming connection  
    client_sock, client_info = server_sock.accept()  
    print("Accepted connection from", client_info)  
  
    # Send a message to the receiver  
    message = "Wi-Fi Connection Lost"  
    client_sock.send(message)  
  
    # Close the connection  
    client_sock.close()  
    server_sock.close()  
  
    # Update troubleshooting state  
    self.troubleshootState = 1
```

Figure 22 - Troubleshooting Bluetooth function

Meanwhile, if the connection was lost at the weather station or a message sent regarding loss of connection from the sensor station(s) was received a popup notifying the user would be displayed on the HMI screen, that way HCI is maintained, and the user could service the router if need be.

To stop the sensor station from recurringly sending messages stating loss of connection while the system was down, I used a variable representing the system state which changes value once a Wi-Fi loss message is sent and stops another from being sent by being part of the if condition that triggers it being sent to begin with (see Figure 23). My reasoning for choosing Bluetooth as my communication protocol here is due to it not relying on a Wi-Fi network connection to work.

```
if station.isconnected() != True and root.troubleshootState = 0:  
    # Send Wi-Fi troubleshoot message via Bluetooth  
    root.sendTroubleshootMessage()
```

Figure 23 - If statement triggering Bluetooth troubleshoot message (sensor station).

Wi-Fi Network Management

Following implementation of troubleshooting measures, I discovered that hard programming my credentials was not necessarily the best approach mainly due to the following reasons:

1. Security risks: Exposing sensitive information in the code or firmware can compromise network security if unauthorized access occurs.
2. Limited flexibility: Hard-coded credentials make it difficult to change or update Wi-Fi settings without reprogramming the device.
3. Lack of code reusability: Hard-coded credentials hinder the reuse of code in different network environments.

4. Maintenance and version control challenges: Managing multiple devices with different credentials becomes more complex.

Instead, I realised it would be favourable to use a dynamic configuration method such as user input, secure configuration storage, or a Wi-Fi manager library, as these approaches would improve security, flexibility, and ease of use for ESP32 devices when connecting to Wi-Fi networks. So, I decided to develop a system where the two ESP32 boards could securely exchange the Wi-Fi credentials over Bluetooth.

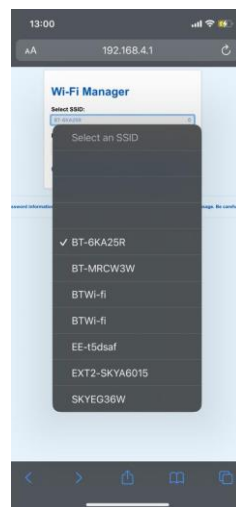


Figure 24 - Wi-Fi Manager Interface (showing nearby networks)

To accomplish this, I used a Wi-Fi Manager library to handle Wi-Fi connection on the sender ESP32. This library allows users to connect to their desired Wi-Fi network by configuring it in an access point mode or a captive portal. Once the Wi-Fi Manager establishes a successful connection, it retrieves the Wi-Fi credentials entered by the user.

To ensure the confidentiality of the credentials during transmission, I incorporated the “cryptolib” library, which provides encryption and decryption functions using a specific encryption algorithm. I utilized a symmetric encryption approach, where the same encryption key is used for both encryption and decryption.

```
import os
import binascii

# Generate a random encryption key
encryption_key = os.urandom(32) # Generate a 256-bit (32-byte) key

# Convert the key to hexadecimal string
hex_key = binascii.hexlify(encryption_key).decode()

# Print the key in hexadecimal format
print(hex_key)
```

Figure 25 – Code used to generate random encryption key.

In the sender code (weather station), after obtaining the Wi-Fi credentials, I encrypted the SSID and password using the encryption key. This process converts the sensitive information into unreadable cipher text. Then, using the Bluetooth module, I established a connection with the receiver ESP32 and sent the encrypted Wi-Fi credentials.

On the receiver side (sensor stations), I configured the Bluetooth module to be discoverable, allowing the sender to locate it. Once a Bluetooth connection was established, my sensor station(s) received the encrypted credentials. Using the encryption key, I decrypted the SSID and password, restoring them to their original form. After decryption, I configured the receiver ESP32 with the obtained Wi-Fi credentials and connected it to the desired Wi-Fi network.

```
# Create a server socket
server_sock = bluetooth.BluetoothSocket(bluetooth.RFCOMM)

# Enable Bluetooth discoverable mode
bluetooth.advertise_service(server_sock, root.bluetooth_name, service_id=bluetooth.UUID("00001101-0000-1000-8000-00805F9B34FB"), advertise=True)

# Wait for an incoming Bluetooth connection
client_sock, client_info = server_sock.accept()

# Receive encrypted Wi-Fi credentials from the sender
received_data = client_sock.recv(1024)
encrypted_ssid, encrypted_password = received_data.decode().split(':')

# Encryption key
encryption_key = root.encryptionKey

# Decrypt the Wi-Fi credentials
ssid = cryptolib.decrypt(encryption_key, encrypted_ssid)
password = cryptolib.decrypt(encryption_key, encrypted_password)

# Run function to establish a connection and get MAC address
root.getConnection(station, ssid, password)

# While statement to add "." to, and display awaiting message
while not station.isconnected():
    print(".", end=" ")
    utime.sleep(0.75)

# Display successful connection message on OLED display
print(" Connection Successful!")

# Close the connection
client_sock.close()
server_sock.close()
```

Figure 26 - Receiver (sensor station) code for connecting to Wi-Fi network after Bluetooth message is received.

This approach of encrypting the Wi-Fi credentials before transmission over Bluetooth provided an additional layer of security. By encrypting the data, I ensured that even if someone intercepted the transmission, they would be unable to read the sensitive information without the correct encryption key.

6.2.2. Eliciting Readings from my Sensor station(s)

In line with my pass test checklist the next step of software implementation was, to retrieve sensor readings and collect accurate and real-time data on the environmental parameters. Accurate and real-

time sensor readings were essential for monitoring environmental parameters within the sensor network. I integrated various sensors into the system and implemented algorithms to retrieve readings from each sensor.

Retrieving sensor readings from the BME280

The BME280 sensor was integrated into the sensor network to collect real-time and accurate data on environmental parameters, including temperature, pressure, and humidity. This section outlines the process of retrieving sensor readings from the BME280 sensor, including library integration, reading acquisition, and formatting.

BME280 Library Integration

The BME280 library was imported and instantiated within the "ESP32Handler" class to enable communication with the BME280 sensor. The following code snippet demonstrates the initialization of the BME280 sensor:

```
# Importing BME280 Library
import BME280

class ESP32Handler():
    def __init__(self):
        # Instantiate BME280 dependables
        self.i2c = I2C(scl=Pin(22), sda=Pin(21), freq=10000)
        self.bme = BME280.BME280(i2c=self.i2c)
```

Figure 27 - Snippet depicting library integration and instantiation of BME280 dependables.

Sensor Reading Acquisition

The "getSensorReadings" method of the "ESP32Handler" class encapsulated the process of obtaining sensor readings. Temperature, pressure, and humidity readings were acquired using the respective functions provided by the BME280 library. Here is the code snippet illustrating the reading acquisition process:

```
def getSensorReadings(self):
    """
    This method is responsible for returning the current sensor reading values
    returns:
        temp: BME280 sensor temperature reading
        pres: BME280 sensor pressure reading
        hum: BME280 sensor humidity reading
    """
    # Handling of BME280 Sensor -----
    bme = self.bme

    temp = bme.temperature
    pres = bme.pressure
    hum = bme.humidity
```

Figure 28 - Snippet depicting BME280 Sensor Reading Acquisition.

Formatting

To ensure consistent formatting, the unit digits were removed from each reading using slicing operations. The purpose of this step was to prepare the readings for further processing and analysis. The code snippet below demonstrates the removal of unit digits from the readings:

```
# Remove unit digits from each string
temp = temp[:-1]
pres = pres[:-3]
hum = hum[:-1]

#-----
return temp, pres, hum
```

Figure 29 – Slicing operations for correct formatting.

Retrieving sensor readings from the Adafruit Anemometer

To capture wind speed measurements within the sensor network, I integrated an Adafruit Anemometer. The following section outlines the key steps involved in obtaining sensor readings from the Adafruit Anemometer:

Analog Input Reading

I began by reading the analog input value from the Adafruit Anemometer using the "ADC" class which is a part of the "machine" module and provides an interface to interact with the ADC hardware. By importing "ADC" from the "machine" module, I could create an instance of the ADC class to read analog input values. To ensure accurate measurements, I set the attenuation ratio to "ADC.ATTN_11DB". Here is the code snippet for reading the analog input value.

```
# Declaring an ADC object
anemometerAnalogInput = ADC(Pin(35))

# Setting the attenuation ratio to full range
anemometerAnalogInput.atten(ADC.ATTN_11DB)

# Reading the analog input value
anemometerValue = float(anemometerAnalogInput.read())
```

Figure 30 - Analog Input Reading

Voltage Calculation

The next step was to calculate the voltage of the anemometer value. Here the analog input value was converted into voltage by dividing it by the maximum value of the analog-to-digital conversion (4095) [25] and multiplying it by the supply voltage (3.3V). The resulting voltage value was stored in the variable voltage. The following equation was utilised to derive the voltage:

$$\left(\frac{\text{anemometerValue}}{4095} \right) \times 3.3$$

```
# Calculating the voltage
voltage = (anemometerValue / 4095) * 3.3
```

Figure 31 - Voltage Calculation

Wind Speed Calculation

Using the obtained voltage value, I calculated the wind speed. If the voltage was below or equal to the minimum voltage threshold ("self.minVoltage"), the wind speed was set to 0.0. Otherwise, I utilized the "map_range" function to map the voltage value within the range of the minimum and maximum voltages of the anemometer to the corresponding wind speed range ("self.minWindSpeed" to "self.maxWindSpeed", see Table 6). Here is the code snippet for wind speed calculation:

```
if voltage <= self.minVoltage:
    windspeed = 0.0
else:
    # Calculating the windspeed by using map_range function
    windspeed = self.map_range(voltage, self.minVoltage, self.maxVoltage, self.minWindSpeed, self.maxWindSpeed)
```

Figure 32 - Query to calculate wind speed.

Rounding and Formatting the Wind Speed

As an additional measure the calculated wind speed value was then rounded to one decimal place using the "round()" function.

```
# Round the result
windspeed = repr(round(windspeed, 1))
windspeed = float(windspeed)
```

Figure 33 - Rounding and formatting the windspeed

The rounded value was initially stored as a string in the windspeed variable and then converted back to a float in a suitable format for further processing and analysis (publishing to the ThingSpeak cloud platform and displaying on my GUI).

Retrieving sensor readings from the Capacitive Soil Moisture Sensor

To monitor soil moisture levels within a sensor network, a Capacitive Soil Moisture Sensor was implemented. This sensor played a crucial role in collecting accurate data on soil moisture. This section outlines the process of retrieving sensor readings from the Capacitive Soil Moisture Sensor.

Sensor Initialization and Configuration

To begin, I initialized the sensor by creating an ADC object called "analogInput" and connecting it to the appropriate pin for analog-to-digital conversion. This setup allowed me to precisely measure soil moisture levels.

```
# Declaring an ADC object
analogInput = ADC(Pin(34))
```

Figure 34 – Capacitive Soil Moisture Sensor Initialization and Configuration

Setting the Attenuation Ratio

For accurate readings, I set the attenuation ratio of the ADC object to "ADC.ATTN_11DB." This configuration provided a wide dynamic range, ensuring accurate measurements across different moisture levels.

```
# Setting the attenuation ratio to full range
analogInput.atten(ADC.ATTN_11DB)
```

Figure 35 - Setting the Attenuation Ratio

Reading Analog Input

The next step was to read the analog input value from the Capacitive Soil Moisture Sensor. Using the "analogInput.read()" function, I obtained the analog input value from the Capacitive Soil Moisture Sensor. This conversion process transformed the analog signal into a digital value, which represented the moisture level in the soil.

```
# Reading the analog input value
soilMoistureValue = analogInput.read()
```

Figure 36 - Reading the Analog Input

Mapping to Soil Moisture Percentage

To better understand the moisture level, I mapped the analog input value to a soil moisture percentage using the "map_range" function. This transformation allowed me to interpret the values on a scale ranging from 0 to 100, with 0 indicating dry soil and 100 representing saturated soil.


```
# Calculating the soil moisture percentage by using map_range function
soilMoisturePercent = self.map_range(soilMoistureValue, self.airValue, self.waterValue, 0, 100)
```

Figure 37 - Mapping to Soil Moisture Percentage

Determining Air and Water Values

To establish accurate calibration, I conducted quantitative tests. By fully submerging the sensor in a cup of water and running the code from the CSMS module [23], I was able to establish the water value = 992, meanwhile by holding the soil moisture sensor in the air and running the code I found the air value = 2700. Despite this, I found the sensor readings to still be slightly inaccurate. Eventually, I found by slightly increasing the air and water value by increments I would establish the optimum values of; 2828 for the air value, and 1011 for the water value.

```
import machine
from src.CSMS import CSMS

adc = machine.ADC(machine.Pin(34))

csms = CSMS(adc)
csms.calibrate()
```

Figure 38 – Code to calibrate the Capacitive Soil Moisture sensor.

Rounding and Storing the Moisture Percentage

To present the data precisely, I rounded the calculated soil moisture percentage to one decimal place using the "round()" function. This allowed for a clearer understanding of the moisture level. The rounded value was then stored in the "moisture" variable for further analysis and processing.

```
# Rounding the result
moisture = repr(round(soilMoisturePercent, 1))
```

Figure 39 - Rounding and Storing the Moisture Percentage.

Analysis of : "Eliciting Readings from my Sensor station(s)"

By effectively integrating the BME280, Adafruit Anemometer, Capacitive Soil Moisture Sensor, and implementing the outlined steps, the sensor network successfully retrieved accurate and real-time environmental data. The BME280 provided precise measurements of temperature, pressure, and humidity, enabling comprehensive monitoring of the surrounding conditions. The Adafruit Anemometer contributed essential wind speed readings and, the Capacitive Soil Moisture Sensor delivered valuable soil moisture data.

6.2.3. Querying Time & Collecting targeted sensor readings (Timestamps)

As part of my design documentation, I desired to collect data at predetermined times. To do so I incorporated a program to query the current time and collect targeted sensor readings based on specific timestamps. This functionality would allow me to capture relevant data at designated intervals for further analysis and reporting.

Declaring Timestamps

Initially, I instantiated the timestamp lists “hourTimeStamps” and “keyTimeStamps” to store the predefined time stamps for triggering specific actions.

```
# Instantiate time-stamp lists
self.hourTimeStamps = []
self.keyTimeStamps = ['09:00:00', '15:00:00']

# For loop for inputting hourly time stamps
for time in range(0,24):
    if time in range(0,10):
        self.hourTimeStamps.append("{}:00:00".format(time))
    else:
        self.hourTimeStamps.append("{}:00:00".format(time))
```

Figure 40 - Creating Timestamps

In the above code snippet, I initialized hourTimeStamps as an empty list and keyTimeStamps with specific time values ('09:00:00' and '15:00:00') for the key updates. Additionally, I used a for loop to populate the hourTimeStamps list with hourly time stamps ranging from '00:00:00' to '23:00:00'. The time stamps were formatted as strings with leading zeros for single-digit hours. By instantiating these time-stamp lists, I had the flexibility to define the desired time points for triggering actions such as sending ESPNow messages or publishing sensor readings to the ThingSpeak API.

Reading the Time

As part of this pass test, I would need to implement a query. This involved comparing the current time with those listed amongst my timestamps. To retrieve the current time, I utilized the “getDateandTime” method, which employed the “utime.localtime()” function. This function provided me with the current time, which I formatted into hours, minutes, and seconds, and stored in the “current_time” variable:

```
def getDateandTime(self):
    """
    This method is responsible for getting the current date and time by using the localtime function, and returning these values as variables.
    returns:
        current_date: the current date
        current_time: the current time
    """
    # Get current local time
    current = utime.localtime()

    # Format current time
    current_time = "{:02d}::{:02d}::{:02d}".format(current[3], current[4], current[5])

    return current_time
```

Figure 41 - Reading the time (getDateandTime function)

Querying the time against timestamps

After establishing the current time, the next step was to determine if an ESPNow message containing the sensor readings in a data packet should be sent to the weather station, and if sensor readings should be published to the ThingSpeak API, this was the framework for my query. Here I incorporated a while loop that checked if the current_time fell within the “hourTimeStamps”. This loop ensured that the ESP32 continuously monitored the current time and triggered the necessary actions when the time matched one of the predefined hour stamps:

```
while current_time in root.hourTimeStamps:

    # Determine message type based on current time
    messageType = root.checkTime(current_time)

    # Get the sensor data for temperature, pressure, humidity, windspeed, and soil moisture
    temperature, pressure, humidity, windspeed, moisture = root.getSensorReadings()

    # Configure data packet
    dataPacket = root.configureDataPacket(messageType, temperature, pressure, humidity, windspeed, moisture, current_time)

    print(dataPacket)

    # Delay before publishing topic
    utime.sleep(5)

    # Publish sensor readings to Thingspeaks API via urequests
    requestStatus = root.updateThingspeaks(temperature, pressure, humidity, windspeed, moisture)

    # Send data packet via ESPNow
    espStatus = root.sendESPMessage(dataPacket, e)

    print(espStatus, requestStatus)

    # Exit while loop to continue as normal
    break
```

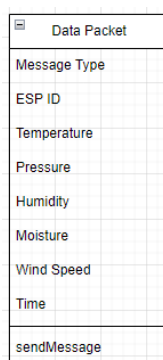
Figure 42 – While loop querying the time against established timestamps.

Within this while loop, my system would call the “getSensorReadings()” function among other methods, subsequently collecting the sensor readings at times that were predetermined, hence targeted sensor readings. Ultimately, my reason for this approach was that I found collecting data at very small intervals ineffective in improving my system, and proved detrimental when it came to data acquisition (see section 6.2.4.).

6.2.4. Data Configuration & Transmission

Sending Data to the Weather Station

As stated in the prior subsection (6.2.3.), the use of a while loop where I queried the “current_time”, checking if it matched one of the predefined time stamps ensured that certain actions only occurred at certain times. When a match was found, sensor readings would be retrieved and then be sent to the weather station, however, some preprocessing would be required to ensure formatting, and data processing ran smoothly. At the point of data configuration, the aim was to create a data packet object as seen in Figure 25. To achieve this a systematic approach was taken:



Data Packet	
Message Type	
ESP ID	
Temperature	
Pressure	
Humidity	
Moisture	
Wind Speed	
Time	
sendMessage	

Figure 43 - Data Packet Object

Determining the Message Type

Firstly, I invoked the “checkTime” method to determine the appropriate messageType based on the current_time. This method compared the current_time with the predefined hour time stamps to determine the type of message to be sent.

Here is the breakdown of the process within the checkTime method:

- The method received the “current_time” as an input parameter.
- Conditional statements were used to compare the “current_time” with the predefined hour time stamps.
- If the “current_time” matched one of the hour time stamps but not the key time stamps, the “messageType” was set to "hourlyUpdate".
- Else If the “current_time” matched both an hour time stamp and a key time stamp, the “messageType” was set to "keyUpdate".
- The determined “messageType” was returned as the output of the method.

This attribute acted as a classifier differentiating message that were sent to the weather station in to two groups.

Getting the Sensor Readings

Next, the “getSensorReadings” method was called to retrieve the most recent sensor data for temperature, pressure, humidity, windspeed, and moisture. This involved reading the relevant sensor values and processing them accordingly which is discussed in section 6.2.2. .

Configuring the Data Packet

Once the sensor readings were obtained, the “configureDataPacket” method was utilized to create a data packet. This data packet contained the collected sensor readings along with other relevant information, such as the “messageType”, “espID”, and “current_time” as depicted in figure 43. The data packet was structured to ensure the seamless interpretation of data at the weather station end. The steps involved in this process are as follows:

Firstly, the method takes the “messageType”, temperature, pressure, humidity, windspeed, moisture, and “current_time” as input parameters.

```
dataPacket = root.configureDataPacket(messageType, temperature, pressure, humidity, windspeed, moisture, current_time)
```

Figure 44 - Inputting data packet parameters

Next, the method creates a list called “dataList” to hold the collected data and other information. This list is initialized with the relevant values. The values are appended to the “dataList” in a specific order that corresponds to their respective fields in the data packet.

```
dataList = [messageType, espID, temperature, pressure, humidity, windspeed, moisture, current_time]
```

Figure 45 - Parameter conversion to list

The “dataList” is then converted into a string format by using list comprehension and joining the elements with a space separator before the data packet is returned.

```
# Using list comprehension to convert data packet list to string
dataPacket = ' '.join([str(elem) for elem in dataList])

return dataPacket
```

Figure 46 - List conversion to string for ESPNow message

Sending The Message

Finally, the `sendESPMessage` method was employed to transmit the prepared data packet via ESPNow to the weather station. This involved establishing a communication channel with the recipient and sending the data packet, ensuring reliable and efficient transmission. Upon successful transmission, it would return a status indicating the success of the message.

```
def sendESPMessage(self, dataPacket, e):
    """
    This method is responsible for taking the data packet and instance of the espnow module and sending the data packet in a message via the espnow
    communication protocol. The status of the message is subsequently returned as either success or failure.
    params:
        dataPacket: configured data packet containing recent sensor readings (string)
    returns:
        status: status of espnow message (success or failure)
    """
    utime.sleep(8)

    try:
        # Send data to ESPNow Peer
        e.send(None, dataPacket, False)
        status = "ESPNow message successfully sent"
    except:
        status = "FAIL"
    return status
```

Figure 47 - Sending the ESPNow Message (*sendESPMessage* function)

Extra Measures

To improve reliability in data packet transmission via the sensor station, a backup communication channel was implemented using a TCP socket server on the sensor station side. If the ESPNow message failed to send, the socket server method was invoked to send the data packet. This workaround resolved transmission issues, providing a dependable means of transferring sensor readings.

```
def sendViaServerSocket(self, dataPacket):
    """
    This method is responsible for taking the retrieved sensor readings -data packet (temperature, pressure, humidity etc.) and sending it to the
    weather station by establishing a server socket
    params:
        dataPacket:
    """
    # Get instance
    server_socket = socket.socket()

    server_address = socket.getaddrinfo(self.hostName, self.port)[0][-1]

    # Bind the host address and port together
    server_socket.bind(server_address)

    # Configure how many clients the server can listen to simultaneously
    server_socket.listen(1)

    # Set "time out" waiting time for server-client to establish connection (acts as countdown)
    server_socket.settimeout(60)

    try:
        # Accept connection
        conn, address = server_socket.accept()

        # Disable countdown
        server_socket.settimeout(None)

    except:
        # No connection established after 60 second countdown
        return

    print("Connection from: " + str(address) + " established")

    data = dataPacket

    # Send data to the client
    conn.send(data.encode())

    conn.close() # close the connection
```

Figure 48 - Backup communication channel (server side)

On the client side (weather station), a listening method was implemented to receive the data packet from the sensor station. It involved creating a client socket that waited for incoming connections. The

method was scheduled to run simultaneously with the timestamp time(s) to allow time for the sensor station to attempt ESPNow transmission or establish the socket server. This scheduling was achieved using the same practice explained in section 6.2.3. To prevent prolonged waiting, a timeout feature cancelled the connection if the sensor station's server method was not called within a specified time frame. This approach ensured synchronized communication between the sensor station and weather station, enabling reliable data packet transfer.

```
def getClientServerMsg(self):
    ''' This method is responsible for establishing a connection with the socket server (sensor station) to receive the data packet sent via it
    ...

    client_socket = self.client_socket
    client_socket.connect((self.host, self.port)) # Connect to server

    # Receive response
    data = client_socket.recv(1024).decode()

    # Convert data string to list
    data = list(data.split(" "))

    root.checkMessage(data)
```

Figure 49 - Backup communication channel (client side)

Furthermore, I found by performing garbage collect to remove unnecessary data occupying the ESP's memory after sending ESPNow messages I could mitigate the chances of failure when sending ESPNow messages.

Publishing topics to the ThingSpeak API

This part of the project involved me publishing topics to the ThingSpeak API. By utilizing the uRequests library, I was able to establish a connection to the ThingSpeak API and securely transmit the sensor readings in the form of a JSON payload. This integration enabled real-time updates of temperature, pressure, humidity, windspeed, and moisture data, ensuring the availability of accurate and up-to-date information on the ThingSpeak platform.

```
def updateThingspeaks(temperature, pressure, humidity, windspeed, moisture):
    ...
    This method is responsible for taking the sensor readings as an argument and organising a json topic to be then trying to publish (posting)
    it via the urequests module to the Thingspeak API. The status of the request is subsequently return as either success or failure.
    params:
        temperature: most recent temperature readings
        pressure: most recent pressure readings
        humidity: most recent humidity readings
        moisture: most recent soil moisture readings
    returns:
        status: status of request (success or failure)
    ...

    # Organise json
    sensor_readings = {'field1':temperature, 'field2':pressure, 'field3':humidity, 'field4':windspeed, 'field5':moisture}

    # Try to send readings to Thingspeaks API - only works if WiFi still connected
    try:
        # Post readings (topic) to thingspeak API
        request = urequests.post('http://api.thingspeak.com/update?api_key=' + thingspeakAPIKey, json = sensor_readings, headers = http_header )
        request.close()
        status = "Thingspeaks request (topic) successfully published"
    except Exception as e:
        status = e
    return status
```

Figure 50 – Publishing topics to the ThingSpeaks Cloud Platform

Conclusion and Analysis of : "Data Configuration & Transmission"

During data transmission testing, multiple sensor stations caused significant message-sending delays, leading to data loss and system performance issues. The initial code structure struggled with simultaneous processing of incoming messages, updating the HMI screen, and publishing topics. To overcome this, I researched and implemented the "uasyncio" module, enabling asynchronous programming. This allowed the microcontroller(s) to initiate tasks and run background processes concurrently, improving efficiency and reducing information overload. However, publishing requests still posed problems due to ThingSpeak's rate limit. As an alternative, I attempted using SMTP emails for remote data communication but encountered time-consuming issues. Eventually, I discovered that reducing the number of sensor station microcontrollers to one, with each sensor connected, minimized requests to the IoT cloud platform, resolving the issue.

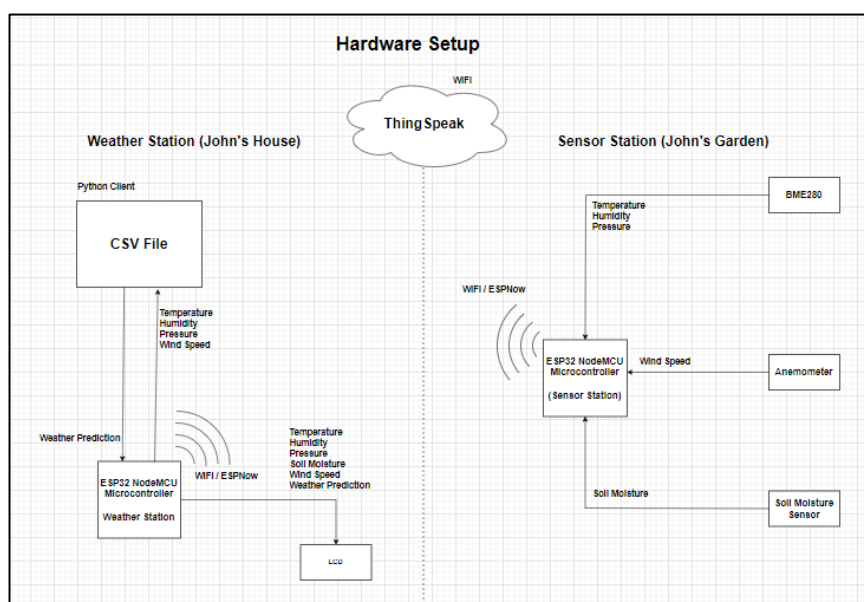


Figure 51 – Final Iteration of the hardware Setup

6.2.5. Interpreting the Data Packet

The image below depicts an instance of a data packet object once received by the weather station.

Data Packet	
Message Type	
ESP ID	
Temperature	
Pressure	
Humidity	
Moisture	
Wind Speed	
Time	
checkMessage	

Figure 52 - Depicting an instance of a data packet object.

By changing the hardware setup, I ultimately changed the way in which the data would be interpreted at the weather station. Instead of querying the data based on the espID and messageType, because all data packets were now coming from one place the only attribute that was of importance as a classifier was now the message type, to differentiate each message.

Checking the Message

Once a new message is received the “checkMessage” method is responsible in processing messages received from the sensor station in an asynchronous manner. Upon receiving a message, the method extracts the message type from the message list and performs the following actions based on the message type:

For messages of type "hourlyUpdate":

- The temperature value is extracted from the message list and stored in the temp variable.
- The temperature value is appended to the “hourlyTemp” list, which keeps track of hourly temperature readings.
- The Nxtion display is updated asynchronously using the “nextionUpdate” method, providing the updated temperature, pressure, humidity, wind speed, and moisture values.

For messages of type "keyUpdate":

- The temperature, pressure, humidity, wind speed, and moisture values are extracted from the message list and assigned to their respective variables.
- The temperature value is appended to the “hourlyTemp” list.
- The timestamp is obtained from the message list, and the hour portion is extracted.
- Depending on the hour value, either the 9 am or 3 pm data values are updated in the “dailyDataDict”, a dictionary used to store daily data.

- The Nextion display is asynchronously updated using the “nextionUpdate” method, providing the updated temperature, pressure, humidity, wind speed, and moisture values.

```

async def checkMessage(self, msgList):
    msgType = msgList[0]
    if msgType == "hourlyUpdate":
        temp = msgList[2]
        # Append variable value to hourly list
        self.hourlyTemp.append(temp)
        # Update Nextion Temperature, Pressure and Humidity, WindSpeed and Moisture Labels
        await self.nextionUpdate(temp, press, hum, windspeed, moisture)
    elif msgType == "keyUpdate":
        temp, press, hum, windspeed, moisture = msgList[2], msgList[3], msgList[4], msgList[5], msgList[6]
        # Append variable values to hourly list
        self.hourlyTemp.append(temp)
        timeStamp = msgList[7]
        timeStampHour = timeStamp[0:2]
        if timeStampHour == "09":
            # Update daily data dictionary
            self.dailyDataDict.update(Temp9am = temp)
            self.dailyDataDict.update(Pressure9am = press)
            self.dailyDataDict.update(Humidity9am = hum)
            self.dailyDataDict.update(WindSpeed9am = windspeed)
        else:
            # Update daily data dictionary
            self.dailyDataDict.update(Temp3pm = temp)
            self.dailyDataDict.update(Pressure3pm = press)
            self.dailyDataDict.update(Humidity3pm = hum)
            self.dailyDataDict.update(WindSpeed3pm = windspeed)
        # Update Nextion HMI Labels
        await self.nextionUpdate(temp, press, hum, windspeed, moisture)
    else:
        pass
    
```

Figure 53 - Checking the message (checkMessage function)

This approach allowed for efficient processing of different types of messages and ensured that the Nextion display, and internal data structures were updated accordingly. By leveraging asynchronous programming, the system could handle incoming messages without blocking other tasks, contributing to improved responsiveness and performance.

6.2.6. Displaying the data – GUI development

This part of implementation was done with the objective of supplying the user with a means of perceiving the acquired data. As a guide in achieving this I decided to use my product design specification and pass test checklists. From the design documents, I established the following requirements as paramount for my GUI(s):

- The data is displayed clearly.
- The data trend is displayed.

The objective of displaying data can be categorised into two parts, displaying data via my Nextion HMI, and displaying data via the ThingSpeaks cloud platform.

Displaying data via the Nextion HMI

Initially, I encountered issues when attempting to communicate with the Nextion display using UART1. However, after reassessing the pin mappings and making necessary adjustments, I switched to UART2. By referring to resources such as the "ESP32 Pinout Reference" from Upesy [21] and the "MicroPython UART Tutorial" from Engineers Garage [22], I gained an understanding of the available UART interfaces

and their pin mappings. This decision was made based on UART2's compatibility with the GPIO17 (TX) and GPIO16 (RX) pins on the ESP32, which allowed for seamless communication with the Nextion HMI.

In my code, the “nextionUpdate” method was responsible for updating the Nextion display with sensor data. Using UART2, I constructed commands as strings to set the values of different components on the HMI, such as temperature, pressure, humidity, windspeed, and moisture. These commands were sent using the “self.uart.write()” method.

```
async def nextionUpdate(self, temp, press, hum, windspeed, moisture):
    """
    This method is responsible for taking the Sensor readings as arguments and writing them to
    the nextion display
    params:
        temp: temperature reading recieved from BME280
        press: pressure reading recieved from BME280
        hum: humidity reading recieved from BME280
        windspeed: windspeed reading recieved from Anemometer
        moisture: moisture reading recieved from soil moisture station
    """
    cmd = "temperature.txt=\""+temp+"\""
    self.uart.write(cmd)
    self.uart.write(self.end_cmd)
    utime.sleep_ms(1000)
    self.uart.read()

    cmd = "pressure.txt=\""+press+"\""
    self.uart.write(cmd)
    self.uart.write(self.end_cmd)
    utime.sleep_ms(1000)
    self.uart.read()

    cmd = "humidity.txt=\""+hum+"\""
    self.uart.write(cmd)
    self.uart.write(self.end_cmd)
    utime.sleep_ms(1000)
    self.uart.read()

    cmd = "windspeed.txt=\""+windspeed+"\""
    self.uart.write(cmd)
    self.uart.write(self.end_cmd)
    utime.sleep_ms(1000)
    self.uart.read()

    cmd = "moisture.txt=\""+str(moisture)+"\""
    self.uart.write(cmd)
    self.uart.write(self.end_cmd)
    utime.sleep_ms(1000)
    self.uart.read()

    await asyncio.sleep(0)
```

Figure 54 - Updating the Nextion HMI

Through careful configuration and utilization of UART2, I successfully established a reliable communication channel with the Nextion display, enabling the seamless display of data on the HMI screen. This approach allowed for a user-friendly and informative interface, enhancing the overall user experience of the weather monitoring system.

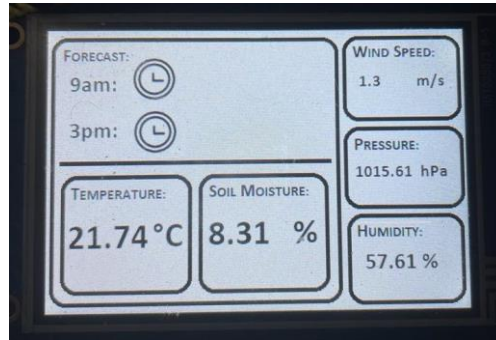


Figure 55 - Nextion HMI Displaying Live Readings

Displaying data via ThingSpeaks

Naturally, by publishing data (topics) to my ThingSpeaks channel the data acquired would be added amongst the historic data to produce a graph for each field. This meant that the data trend for each field could be visualised by anyone with access to the channel. As an extra measure, I decided to incorporate numeric widgets to my channel. By adding numeric widgets to each field in my ThingSpeak channel, I could provide a clear and concise representation of the current values. These widgets offer an easily accessible and straightforward display of the data, allowing users to quickly understand the current state without needing to interpret graphs or analyse historical trends. This addition enhanced the usability of the channel, by also catering to users who prefer a numerical representation of the data. By including numeric widgets alongside the graphs, I could ensure a comprehensive and user-friendly data visualization experience.

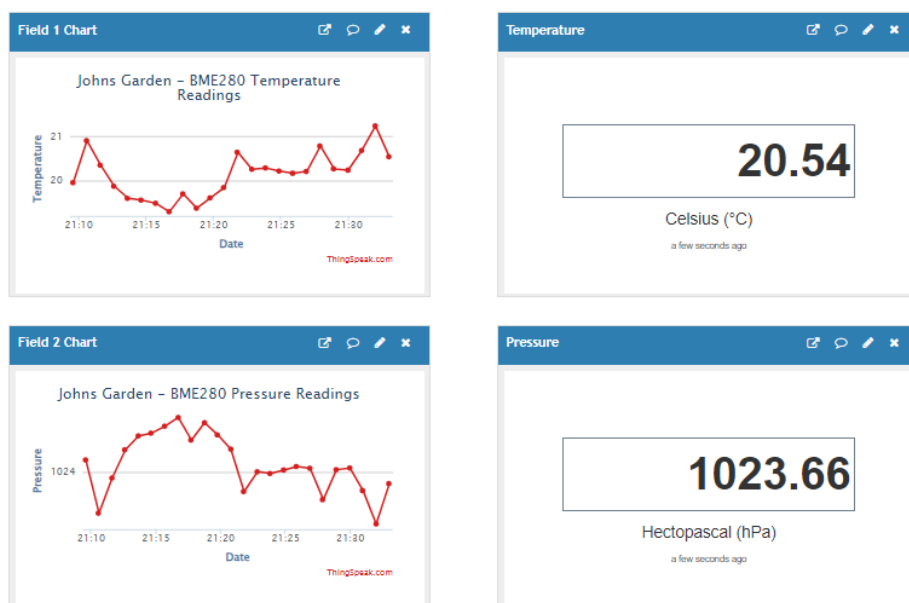


Figure 56 – Image Depicting ThingSpeaks Channel with Numeric widgets incorporated.

6.2.7. Machine learning: Forecasting tomorrow.

After collecting the weather data, I embarked on determining a prediction for the upcoming day. Employing a systematic approach, I leveraged machine learning models and data analysis techniques to forecast the weather parameters for the next day. The process involved data preprocessing, training, and evaluating multiple machine-learning models, selecting the best model based on performance, and making predictions using the chosen model. Additionally, I incorporated rolling averages to capture long-term trends and utilized Zambretti values to assess expected weather conditions based on atmospheric pressure. Below, I outline the detailed step-by-step breakdown, highlighting the rationale behind each step and the methods I employed to ensure accurate weather predictions.

Data Preprocessing

To begin, I imported the necessary libraries for data manipulation, including pandas for data handling, joblib for saving trained models, and various modules from the scikit-learn library for machine learning modelling, data preprocessing, and evaluation metrics. By leveraging these existing libraries, I saved time and benefited from their optimized implementations.

```
1 import warnings
2 warnings.filterwarnings("ignore")
3 import pandas as pd
4 import joblib
5 from sklearn.model_selection import train_test_split
6 from sklearn.linear_model import LinearRegression
7 from sklearn.ensemble import RandomForestRegressor
8 from sklearn.neighbors import KNeighborsRegressor
9 from sklearn.tree import DecisionTreeRegressor
10 from sklearn.metrics import mean_squared_error
```

Figure 57 – Snippet importing the necessary libraries for data manipulation.

Next, I read the weather data from a CSV file into a pandas DataFrame, ensuring a chronological analysis by sorting the dataset based on the date column in ascending order. This allowed me to better understand the temporal patterns and trends in the weather data.

```
92 # Load the dataset
93 data = pd.read_csv('weatherData.csv')
94
95 # Remove rows with at least one Nan value (Null value)
96 data = data.dropna()
97
98 # Sort the dataset by date in ascending order
99 data = data.sort_values('Date')
```

Figure 58 – Snippet reading and chronologically ordering data.

To identify long-term trends and smooth out short-term fluctuations in the data, I calculated the rolling average across the last 30 data entries. This approach provided a clearer representation of the underlying patterns in the data over time. Then, to prepare the data for machine learning modelling, I defined two lists: features and targets. The features list included the columns related to weather parameters, while the targets list included the same weather parameters. This separation allowed me to train the models to predict the target variables based on the provided input features.

```
119 # Calculate rolling average across the last 30 data entries
120 window_size = 30
121 rolling_average_data = data.rolling(window_size).mean().dropna()
122
123 # Define the input features and target variables
124 features = ['MinTemp', 'MaxTemp', 'WindSpeed9am', 'WindSpeed3pm', 'Humidity9am', 'Humidity3pm']
125 targets = ['MinTemp', 'MaxTemp', 'WindSpeed9am', 'WindSpeed3pm', 'Humidity9am', 'Humidity3pm']
```

Figure 59 – Snippet Calculating averages and defining features and targets.

Model Selection and Evaluation

To evaluate the performance of the machine learning models, I split the dataset into training and testing sets using the `train_test_split()` function. This unbiased evaluation assessed the models' ability to generalize to new, unseen data by testing their performance on a separate testing set.

```

130 X = rolling_average_data[features]
131 y = rolling_average_data[targets]
132
133 # Split the dataset into training and testing sets
134 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

```

Figure 60 – Snippet splitting training and testing data.

To identify the model that provided the most accurate predictions, I initially used the K-Nearest Neighbours (KNN) Regressor. However, upon further experimentation and following advice from my supervisor, I decided to explore the performance of additional models. The rationale behind incorporating multiple models was to leverage the unique strengths and characteristics of each model, potentially yielding improved prediction accuracy. To evaluate the performance of the models, I compared the mean squared error (MSE) values. The MSE represents the average squared difference between the predicted and actual values. By selecting the model with the smallest MSE, I ensured that it had the least average squared difference, indicating better predictive performance.

```

136 # Define the models to compare
137 models = [
138     LinearRegression(),
139     RandomForestRegressor(),
140     KNeighborsRegressor(),
141     DecisionTreeRegressor()
142 ]
143
144 # Train and evaluate each model
145 model_mses = []
146 for model in models:
147     model.fit(X_train, y_train)
148     joblib.dump(model, 'weather_predictor.sav')
149     y_pred = model.predict(X_test)
150     mse = mean_squared_error(y_test, y_pred)
151     model_mses.append(mse)
152
153 # Find the model with the smallest MSE
154 best_model_index = model_mses.index(min(model_mses))
155 best_model = models[best_model_index]
156
157 print("Best Model MSE:", min(model_mses))
158 print("Best Model:", best_model)

```

Figure 61 - Snippet establishing best machine learning model.

Visualization of Results

To gain further insights, I visualized the results generated by each model for a chosen feature, such as pressure. This visualization allowed for a direct comparison of the predicted pressure values from each model against the real pressure values. The plot showcased the real pressure values in red and the predicted pressure values in blue. This visual representation provided a clear illustration of the variations in the predictions generated by each model and helped me assess their respective performances for the chosen feature.

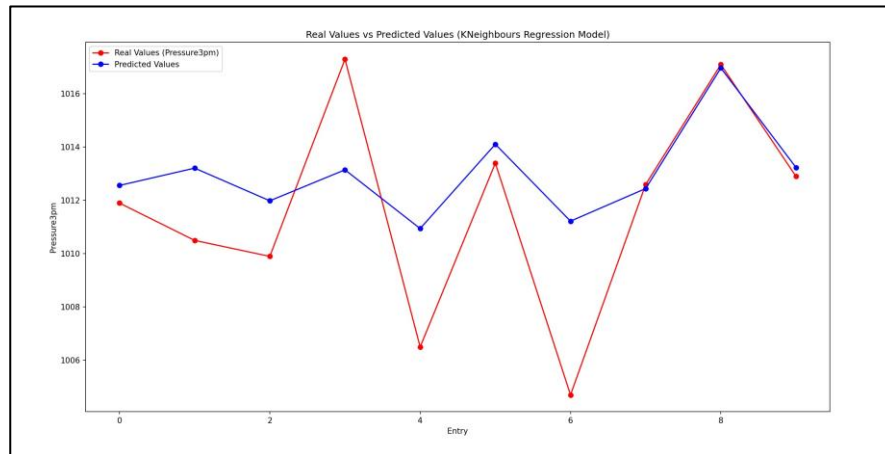


Figure 62 – Graph showing the discrepancy between real and predicted values when utilising the “KNeighborsRegressor” model.

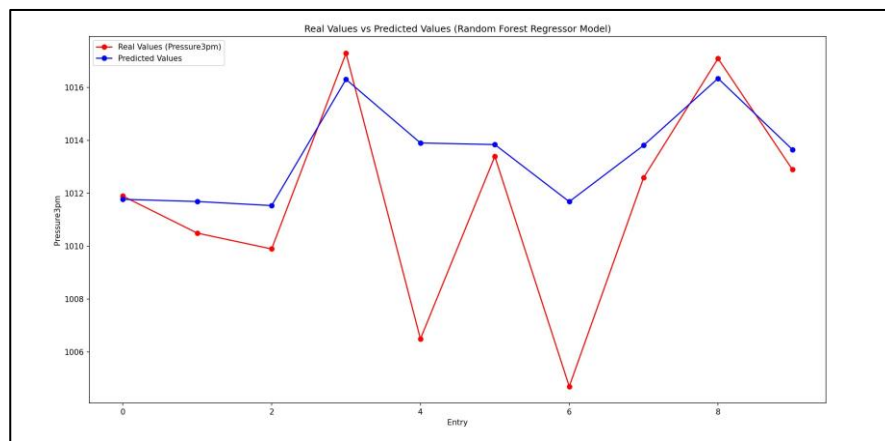


Figure 63 - Graph showing the discrepancy between real and predicted values when utilising the “RandomForestRegressor” model.

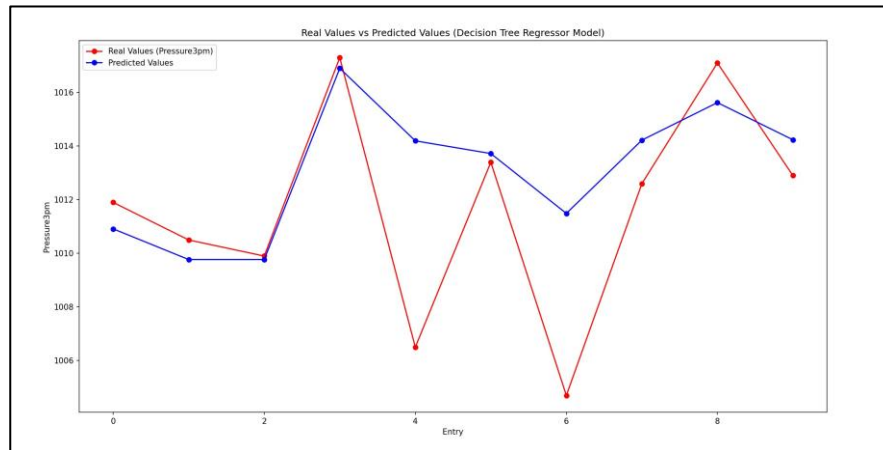


Figure 64 - Graph showing the discrepancy between real and predicted values when utilising the "DecisionTreeRegressor" model.

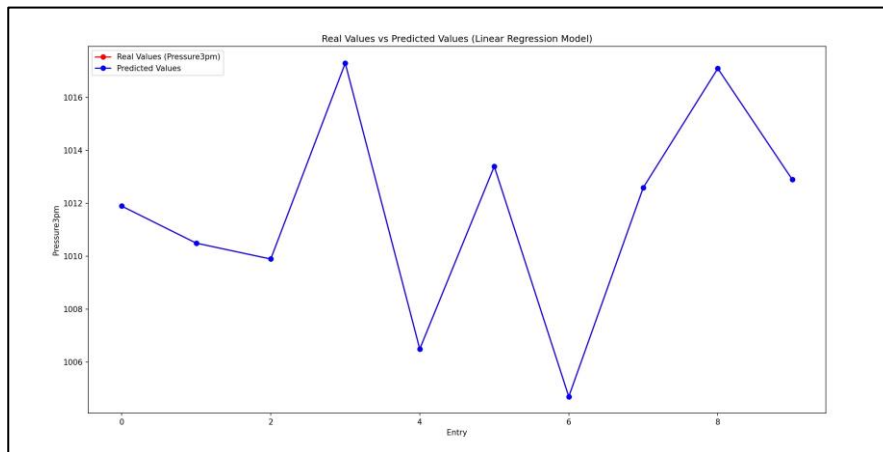


Figure 65 - Graph showing the discrepancy between real and predicted values when utilising the "LinearRegression" model.

As seen in the images above the LinearRegression model was found to be the model that generated the smallest discrepancy, where the difference between the real values and the prediction values were so small it was difficult to view the line representing the real values (red line) amongst its graphs. Nonetheless, I still decided to keep the comparison query amongst my code as part of my quantitative testing.

```
Best Model MSE: 2.119427608308035e-27
Best Model: LinearRegression()
Last 10 Day's Pressure at 3pm Real values: [1011.9, 1010.5, 1009.9, 1017.3, 1006.5, 1013.4, 1004.7, 1012.6, 1017.1, 1012.9]
Last 10 Day's Pressure at 3pm Predictions: [1011.9, 1010.5, 1009.9, 1017.3, 1006.5, 1013.4, 1004.7, 1012.6, 1017.1, 1012.9]
```

Figure 66 – Output shell showing the best models MSE value, best models name, and real values alongside predictions for Pressure at 3pm over the last 10 days.

Prediction for the Next Day

With the best model selected, I then prepared the input features for the next day's weather prediction. I extracted the most recent 30 days' data entries and calculated the rolling average. This approach aimed to capture the recent trends in the weather conditions, which play a crucial role in determining the next day's weather.

```
182 # Prepare input features for the next day (assuming you have data for the next day)
183 last_30_days_data = data.tail(window_size) # Select the last 30 days' data entries
184 next_day_rolling_average = last_30_days_data.mean() # Calculate the rolling average
185 next_day_features = next_day_rolling_average[features].values.reshape(1, -1) # Reshape the features
```

Figure 67 - Snippet preparing input features for the next day.

Using the selected model, I made predictions for the next day's weather parameters based on the input features prepared from the rolling average data. Leveraging the model's learned patterns and relationships, these predictions provided insights into the expected weather conditions for the next day.

```
187 # Make predictions for the next day's weather parameters using the best model
188 next_day_predictions = best_model.predict(next_day_features)
189 next_day_predictions = pd.DataFrame(next_day_predictions, columns=targets)
190
191 print("\nNext Day's Weather Predictions:")#, predictions
192 print(next_day_predictions.to_string(index=False))
```

Figure 68 – Snippet making predictions for the next day's weather parameters.

MinTemp	MaxTemp	WindSpeed9am	WindSpeed3pm	Humidity9am	Humidity3pm	Pressure9am	Pressure3pm	Temp9am	Temp3pm
16.563333	29.97	15.833333	20.033333	53.233333	36.833333	1011.863333	1009.833333	22.566667	28.146667

Figure 69 – Output shell depicting next day's predictions

To present the predicted values in a readable format, I rounded them to two decimal places and stored them in a list. This formatting approach enhanced the clarity and precision of the predicted values, making them easily interpretable and communicable.

In conclusion, the implemented weather forecasting system, incorporating machine learning techniques, has demonstrated its effectiveness in predicting weather parameters for the upcoming day. The results achieved through rigorous testing and evaluation indicate the system's potential for

real-world applications in weather forecasting and decision-making. Further refinements and optimizations can be explored to improve the system's performance and expand its capabilities in the future.

6.2.8. *Zambretti Forecast.*

Additionally, I made the decision to incorporate the Zambretti algorithm due to its unique capability to classify expected weather conditions based on atmospheric pressure changes. By leveraging this algorithm, I was able to assign specific weather icons, such as sun, clouds, or rain, to the predicted weather and display them on the Nextion HMI appropriately by calling the method depicted below.

```
async def updateForecast(self, forecasts):
    ...
    This method is responsible for taking the zambretti predictions as arguments and writing them to
    the nextion display
    params:
    ... forecasts: zambretti prediction list
    ...
    cmd= "earlyforecast.pic={}".format(zambretti[0])
    self.uart.write(cmd)
    self.uart.write(self.end_cmd)
    utime.sleep_ms(100)
    self.uart.read()

    cmd= "lateforecast.pic={}".format(zambretti[1])
    self.uart.write(cmd)
    self.uart.write(self.end_cmd)
    utime.sleep_ms(100)
    self.uart.read()

    await asyncio.sleep(0)
```

Figure 70 - Updating forecast with Zambretti predictions.

This approach greatly enhanced the visual representation of the forecast, allowing users to understand the expected weather quickly and intuitively immediately. By combining the power of the Zambretti algorithm with other predictive models, I aimed to improve the accuracy and reliability of the weather classification, enabling users to make more informed decisions based on the expected weather conditions. This incorporation of the Zambretti algorithm not only added a valuable feature to the weather prediction system but also enhanced its usability and user experience.

As part of this process, I compared the atmospheric pressure prediction with the actual atmospheric pressure data (most recent). If the pressure was rising, the algorithm assigned a Zambretti value that indicated clear or improving weather. If the pressure was falling, it suggested deteriorating weather conditions. For steady pressure, the algorithm assigned a value that represented a stable weather.

To derive the atmospheric pressure, I used the following formula:

$$P_0 = P \left(1 - \frac{0.0065h}{T + 0.0065h + 273.15} \right)^{-5.257},$$

Figure 71 - Sea Level Pressure Formula

In this formula, 'P' represents the pressure at a specific time, 'h' denotes the altitude of the location, and 'T' signifies the temperature at that time. By incorporating these variables into the equation, I was able to calculate the atmospheric pressure reduced to sea level.

The ID values are then sent back to the weather station in the form of a list and extracted via a forecast method to display on the HMI.

7. Further Developments

During the project, I started designing casings for each sensor station using SolidWorks to provide physical protection and organization for the components. To achieve these models, I referenced the data sheets for each of my external components.

However, when I decided to change my hardware setup, the previously established casings for the sensor stations became obsolete. To address this, for future developments, I intend to design a new enclosure that can accommodate all the sensors currently onboard. This unified casing will provide a compact and streamlined solution for housing the components. Additionally, as part of the Nextion HMI GUI I plan to incorporate a Wi-Fi network manager into my system where nearby network will be displayed, SSID's can be selected on, and passwords entered to establish a connection on the Nextion screen instead of a phone or tablet.

8. Conclusions

In conclusion, it is important to highlight that this project was undertaken as an individual effort rather than aiming to develop a fully commercialized system. The primary focus was to address a specific problem by exploring the potential of sensor-based solutions in collecting and abstracting environmental data. The goal was to provide users with easily understandable abstractions of environmental insights. Additionally, the project involved the application of data science techniques to enable predictions pertinent to precision agriculture. By integrating electronic sensors, hardware components, machine learning algorithms, and networking infrastructure, the IoT system developed for environmental data collection showcases a promising foundation for further advancements.

Despite being able to achieve the main project requirements, I believe there are areas where I could have improved my approach. Specifically, I feel that establishing a contingency plan to address time management issues would have been beneficial. Towards the end of the project, I found myself rushing and making significant changes that ultimately had an impact on the outcome. It highlighted the importance of having a well-thought-out plan and allowing sufficient time for each stage of the project. Moving forward, I will prioritize effective time management and establish contingency plans to ensure smoother project execution and better outcomes.

Bibliography

- [1] M. R. M. Kassim, "IoT Applications in Smart Agriculture: Issues and Challenges," 2020 IEEE Conference on Open Systems (ICOS), Kota Kinabalu, Malaysia, 2020, pp. 19-24, doi: 10.1109/ICOS50156.2020.9293672.
- [2] Zhang, S., Chen, X., & Wang, S., "Research on the monitoring system of wheat diseases, pests and weeds based on IoT" in Proceedings of 9th International Conference on Computer Science & Education, 2014, 981-985.
- [3] Goap, A., Sharma, D., Shukla, A. K., & Krishna, C. R., "An IoT based smart irrigation management system using Machine learning and open-source technologies", Computers and electronics in agriculture, 155, 2018, 41-49.
- [4] Nawandar, N. K., & Satpute, V. R., "IoT based low cost and intelligent module for smart irrigation system". Computers and electronics in agriculture, 162, 2019, 979- 990.
- [5] Jawad, H., Nordin, R., Gharghan, S., Jawad, A., & Ismail, M., "Energy-efficient wireless sensor networks for precision agriculture: A Review", Sensors, 17(8), 2017, 1781
- [7] M. Asikainen, K. Haataja and P. Toivanen, "Wireless indoor tracking of livestock for behavioral analysis," 2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC), Sardinia, Italy, 2013, pp. 1833-1838, doi: 10.1109/IWCMC.2013.6583835.
- [8] M. S. Farooq, O. O. Sohail, A. Abid and S. Rasheed, "A Survey on the Role of IoT in Agriculture for the Implementation of Smart Livestock Environment," in IEEE Access, vol. 10, pp. 9483-9505, 2022, doi: 10.1109/ACCESS.2022.3142848.
- [9] Trick, C. (2022) What is an IP67 rating? Trusted Computing Innovator. Trenton Systems, Inc. Available at: <https://www.trentonsystems.com/blog/ip67-rating>.
- [10] Ojha, T., Misra, S., & Raghuwanshi, N. S., "Wireless sensor networks for agriculture: The state-of-the-art in practice and future challenges", Computers and electronics in agriculture, 118, 2015, 66-84.
- [11] M. Hossain, M. Fotouhi and R. Hasan, "Towards an Analysis of Security Issues, Challenges, and Open Problems in the Internet of Things," in 2015 IEEE World Congress on Services (SERVICES), New York City, NY, USA, 2015, 21-28.
- [12] Maanak Gupta et al., "Security and Privacy in Smart Farming: Challenges and Opportunities", IEEE Access, Vol. 8, 2020.
- [13] Chris (2021) Arduino with WIFI: A how-to guide for adding and using, Chip Wired. Chip Wired. Available at: <https://chipwired.com/arduino-board-wifi-guide/>
- [14] Kpower (2022) Uno R3 + WIFI ESP8266 + CH340G arduino and WIFI Single Board, Hackster.io. Available at: <https://www.hackster.io/umpheki/uno-r3-wifi-esp8266-ch340g-arduino-and-wifi-single-board-eed9f6>
- [15] Arduino Uno WIFI REV2 (no date) Arduino Official Store. Available at: <https://store.arduino.cc/products/arduino-uno-wifi-rev2>.
- [16] Fahad, E. (2022) Esp32 vs esp8266 NODEMCU, esp32 vs Nodemcu, ESP8266 vs ESP32, Electronic Clinic. Available at: <https://www.electronicclinic.com/esp32-vs-esp8266-nodemcu-esp32-vs-nodemcu->

[eps8266-vs-esp32/#:~:text=%20ESP32%20is%20better%20than%20ESP8266%20because%20ESP32,touch%20and%20can%20be%20used%20to%20trigger%20events.](#)

[17] Cdaviddav et al. (2021) Arduino vs ESP8266 vs ESP32 microcontroller comparison, DIYIOT. Available at: <https://diyi0t.com/technical-datasheet-microcontroller-comparison/>

[18] ESPNow ESP. Available at: <https://www.espressif.com/en/solutions/low-power-solutions/esp-now>

[19] Anemometer wind speed sensor w/analog voltage output (no date) The Pi Hut. Available at: https://thepihut.com/products/anemometer-wind-speed-sensor-w-analog-voltage-output?variant=27739587537&cy=GBP&utm_medium=product_sync&utm_source=google&utm_content=sag_organic&utm_campaign=sag_organic&msclkid=3c8b5aebc5dd1262bec139730d3d9879&utm_term=4585375810283320.

[20] Gupta, D. (2018) Capacitive V/S resistive soil moisture sensor, Hackster.io. Available at: <https://www.hackster.io/devashish-gupta/capacitive-v-s-resistive-soil-moisture-sensor-e241f2>.

[21] Upesy. "ESP32 Pinout Reference: GPIO Pins Ultimate Guide." Available at: <https://www.upesy.com/blogs/tutorials/esp32-pinout-reference-gpio-pins-ultimate-guide>.

[22] Engineers Garage. "MicroPython UART Tutorial with ESP8266/ESP32." Available online: Link.

[23] MicroPython capacitive soil moisture sensor GitHub. Available at: <https://github.com/ashleywm/micropython-capacitive-soil-moisture-sensor/blob/master/README.md>.

[24] Nwankwo-Igwe, C. What is the difference between python and MicroPython?, Educative. Available at: <https://www.educative.io/answers/what-is-the-difference-between-python-and-micropython>.

[25] Juodvalkis, R. et al. (2019) ESP32/ESP8266 analog readings with MicroPython, Random Nerd Tutorials. Available at: <https://randomnerdtutorials.com/esp32-esp8266-analog-readings-micropython/>.

Appendix

Appendix 1 - Project Blog: [Home | My Site \(pcnkwo.wixsite.com\)](https://pcnkwo.wixsite.com)

Appendix 2 - GitHub Repository: [cnkwo/IoT_Environmental_Data_Collection \(github.com\)](https://github.com/cnkwo/IoT_Environmental_Data_Collection)

Appendix 3 – Bill of Materials Table

Component Name	Model	Quantity	Price
ESP32-DevKitC NodeMcu	AZDelivery NodeMCU32	1	£25.59
BME280	AZDelivery GY-BME280	1	£13.99
Nextion 3.5-inch HMI Display	DIYmalls NX4832T035	1	£53.90
3.7V 820mAh 653042 Li-Ion Polymer Battery	EEMB LP 653042 Battery	1	£7.29
HUAREW Breadboard and Jumper Wires Kit	HR-DP-5V124P	1	£12.99
Anemometer Wind Speed Sensor w/Analog Voltage Output	Adafruit ADA1733	1	£44.80
20 x Micro JST PH 2.0 2-Pin Connector Plug Male and 10cm Red and Black Silicone Cable Wire with Female Connector	Micro JST PH 2.0	1	£6.49
Boost Converter DC-DC Step Up 2A Voltage regulator With USB	BCU2AU	1	£2.50

